Elementary Graph Algorithms

Graphs

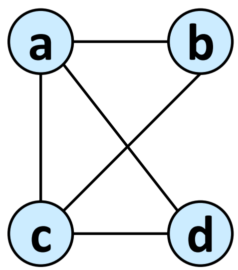
- *Graph $G = (V, E)$*
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \rightarrow v$. Self loops are allowed.
  - Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbf{R}$.
  - Dense: $|E| \approx |V|^2$.
  - Sparse: $|E| << |V|^2$.
- $|E| = O(|V|^2)$

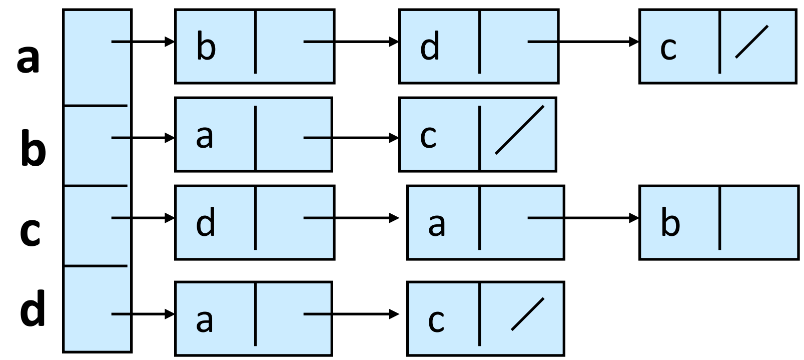
Graphs

- If $(u, v) \in E$, then vertex v is adjacent to vertex u.
- Adjacency relationship is:
  - Symmetric if G is undirected.
  - Not necessarily so if G is directed.
- If G is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then G is a tree.

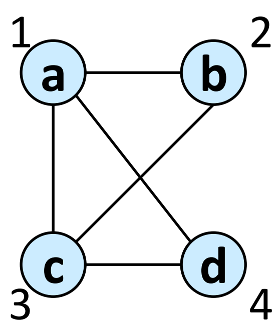
Representation of Graphs

- Two standard ways.
  - Adjacency Lists.





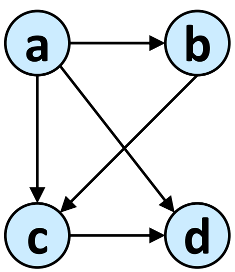
  - Adjacency Matrix.

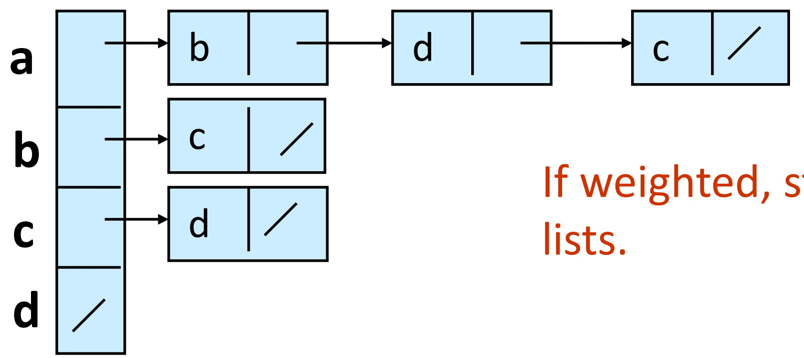


	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

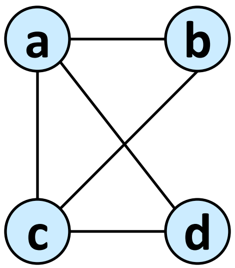
Adjacency Lists

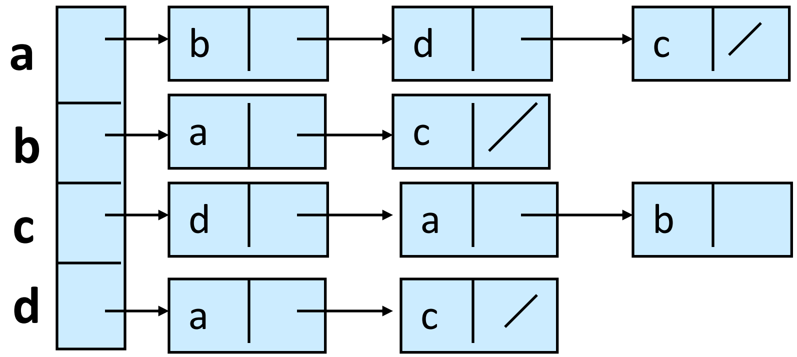
- Consists of an array *Adj* of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to u.





If weighted, store weights also in adjacency lists.





Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{out-degree}(v) = |E|$$

    No. of edges leaving v
  - Total storage: $\Theta(V+E)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is

    $$\sum_{v \in V} \text{degree}(v) = 2|E|$$

    No. of edges incident on v. Edge (u,v) is incident on vertices u and v.
  - Total storage: $\Theta(V+E)$

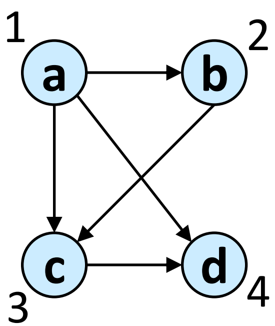
Pros and Cons: adj list

- Pros
  - Space-efficient, when a graph is sparse.
- Cons
  - Determining if an edge $(u,v) \in G$ is not efficient.
    - Have to search in u's adjacency list. $\Theta(\text{degree}(u))$ time.

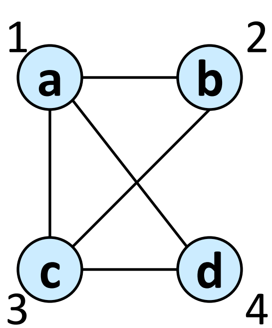
Adjacency Matrix

- $|V| \times |V|$ matrix A.
- Number vertices from 1 to $|V|$ in some arbitrary manner.
- A is then given by:

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



	1	2	3	4
1	0	1	1	1
2	0	0	1	0
3	0	0	0	1
4	0	0	0	0



	1	2	3	4
1	0	1	1	1
2	1	0	1	0
3	1	1	0	1
4	1	0	1	0

$A = A^{\mathrm{T}}$ for undirected graphs.

Space and Time

- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **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.

Graph-searching Algorithms

- Searching a graph:
  - Systematically follow the edges of a graph to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

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, or shortest path) from s to v, for all $v \in V$. $d[v] = \infty$ if v is not reachable from s.
  - $\pi[v]$ = u such that (u, v) is last edge on shortest path $s \rightsquigarrow v$.
    - u is v's predecessor.
  - Builds breadth-first tree with root s that contains all reachable vertices.

Definitions:

Path between vertices u and v: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u=v_1$ and $v = v_k$, and $(v_i, v_{i+1}) \in E$, for all $1 \leq i \leq k-1$.

Length of the path: Number of edges in the path.

Path is simple if no vertex is repeated.

Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
  - A vertex is "discovered" the first time it is encountered during the search.
  - A vertex is "finished" if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
    - Colors are required only to reason about the algorithm. Can be implemented without colors.

BFS(G,s)

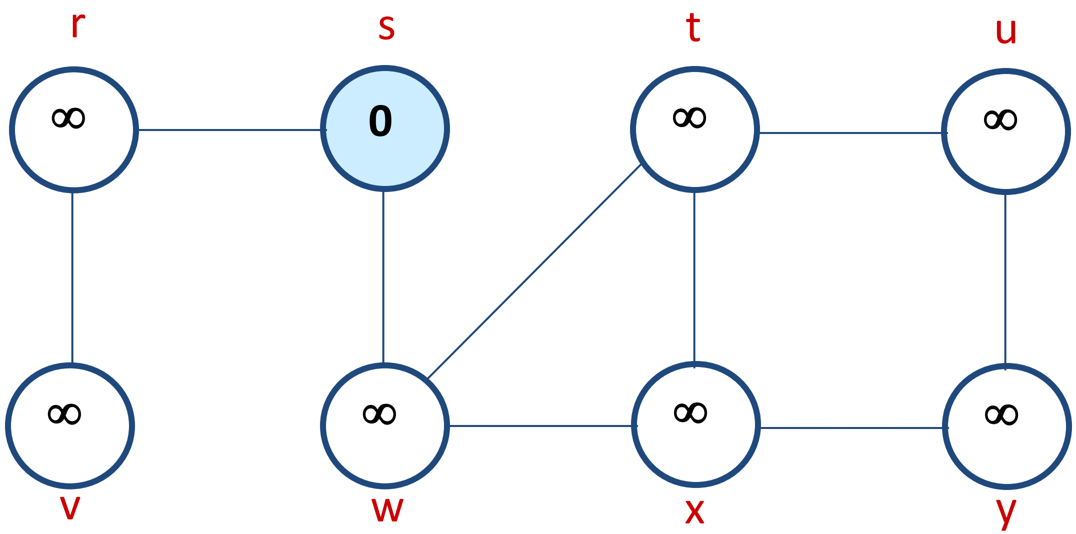
```
for each vertex u in V[G] – {s}
        do color[u] ← white
            d[u] ← ∝
            π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
                for each v in Adj[u]
                        do if color[v] = white
                                then color[v] ← gray
                                    d[v] ← d[u] + 1
                                    π[v] ← u
                                    enqueue(Q,v)
                color[u] ← black
```

white: undiscovered
gray: discovered
black: finished

Q: a queue of discovered vertices
color[*v*]: color of v
d[*v*]: distance from s to v
π[*u*]: predecessor of v

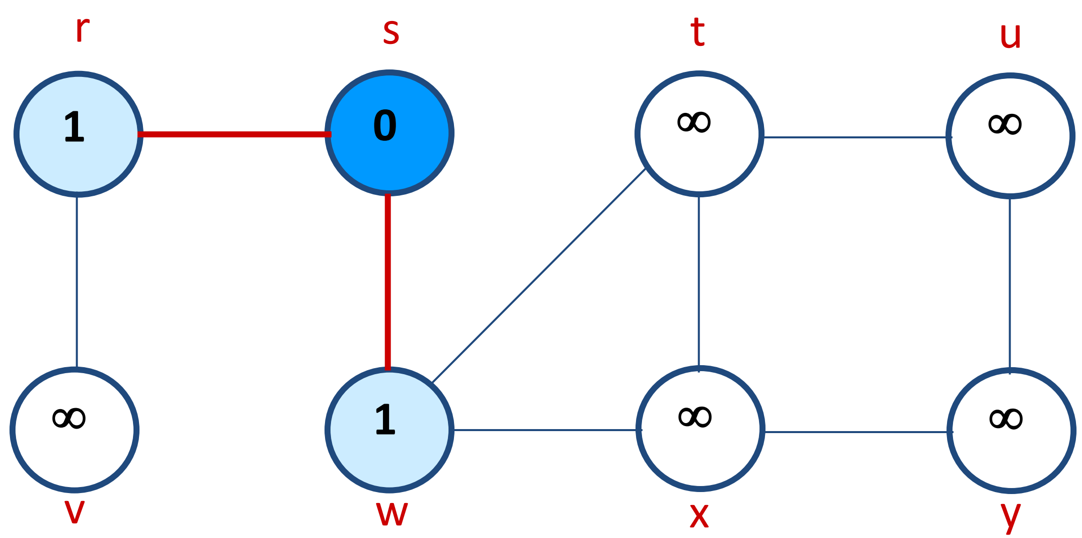
Example: animation.

Example (BFS)



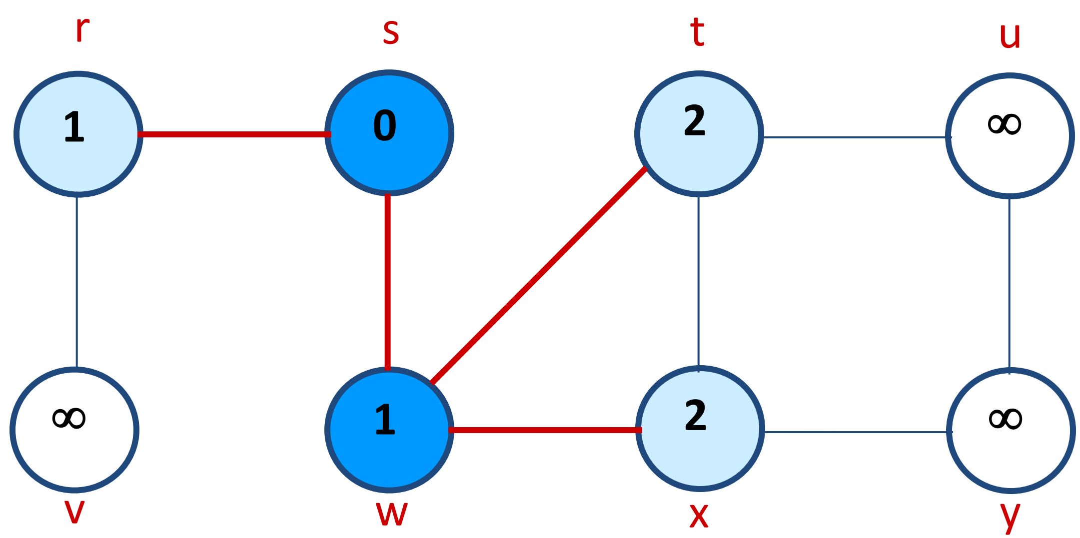
Q: s
0

Example (BFS)



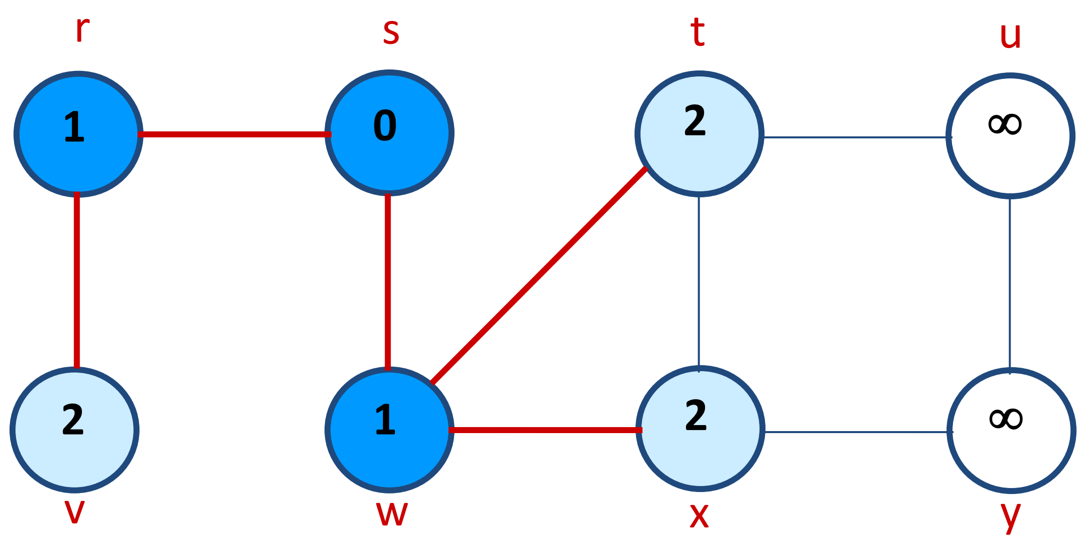
Q: w r
1 1

Example (BFS)



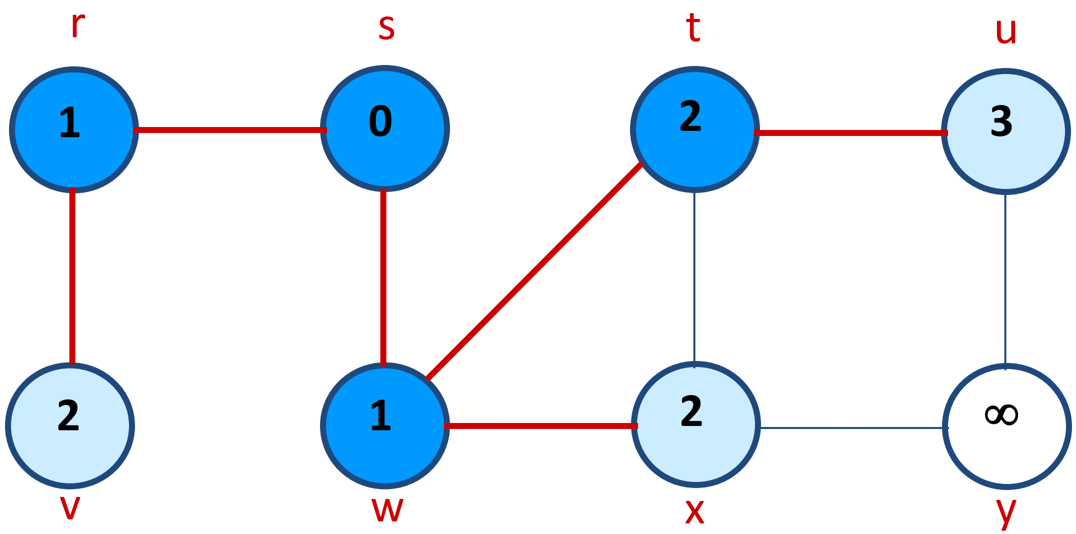
Q: r t x
1 2 2

Example (BFS)



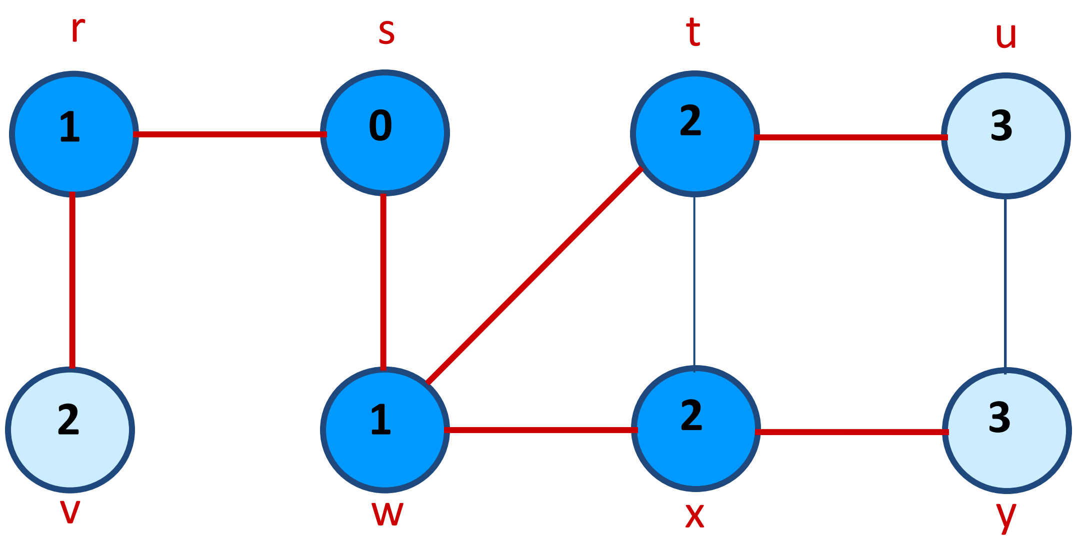
Q: t x v
2 2 2

Example (BFS)



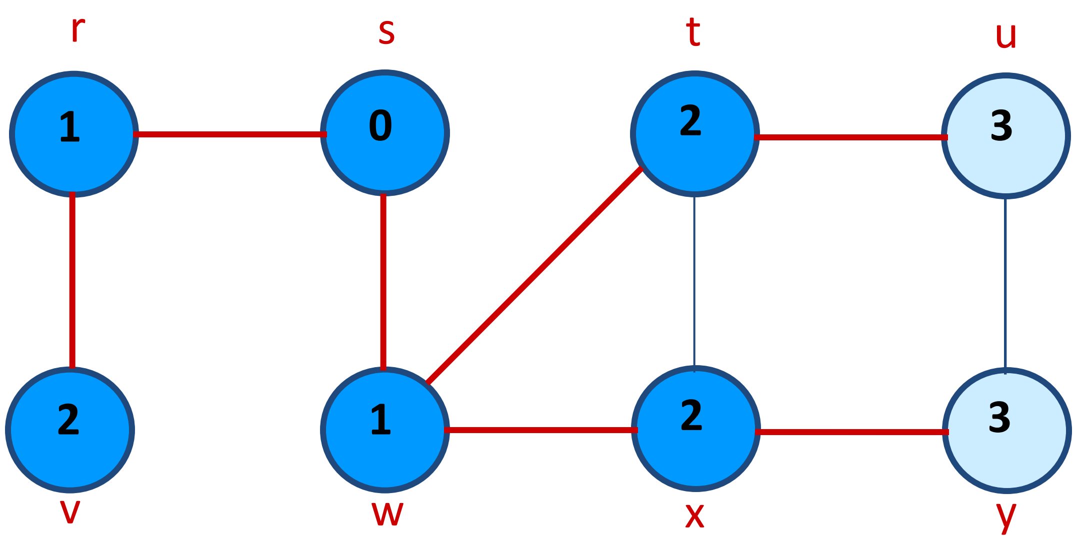
Q:	x	v	u
	2	2	3

Example (BFS)



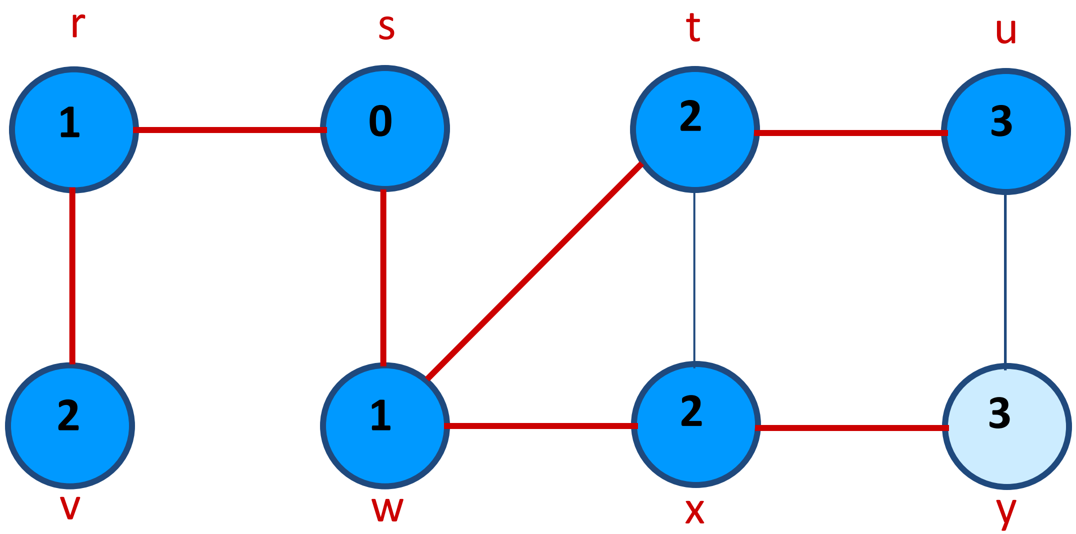
Q: v u y
2 3 3

Example (BFS)



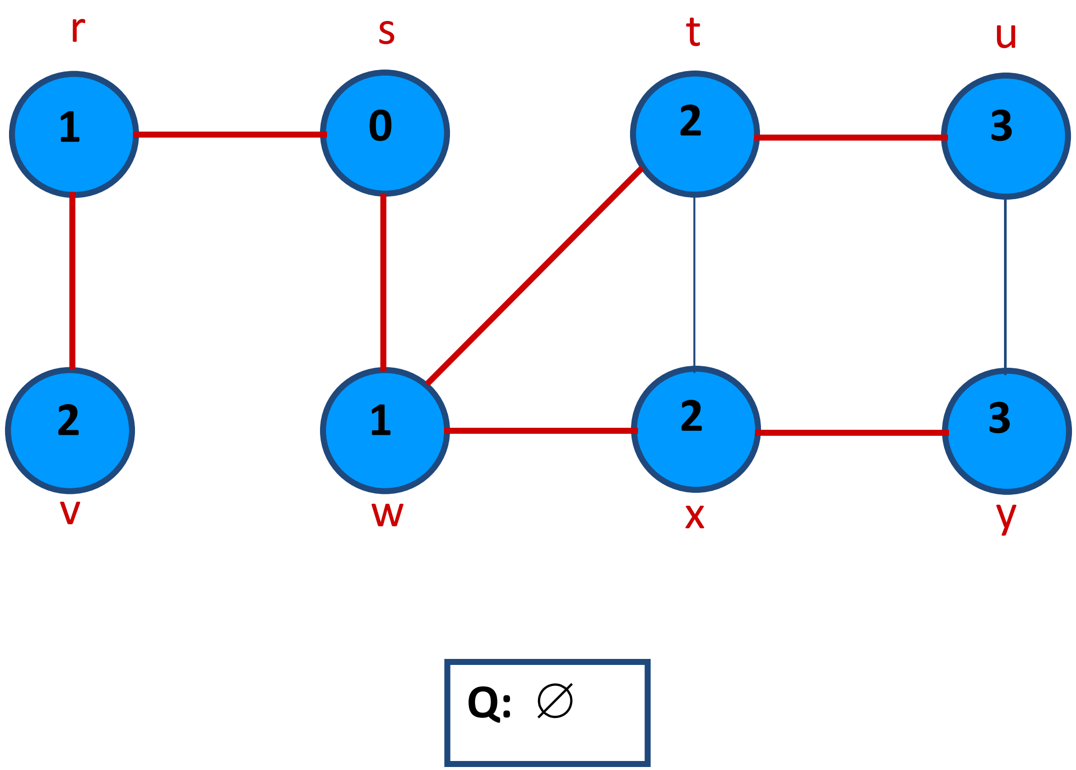
Q:	u	y
	3	3

Example (BFS)

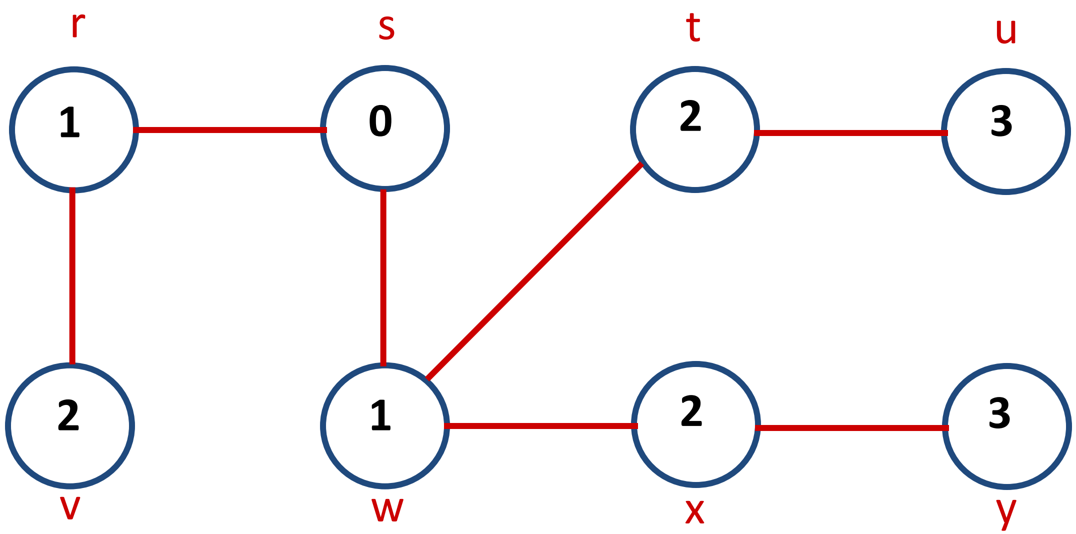


Q: y
3

Example (BFS)



Example (BFS)



BF Tree

Analysis of BFS

- Initialization takes $O(V)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices => total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.
- .

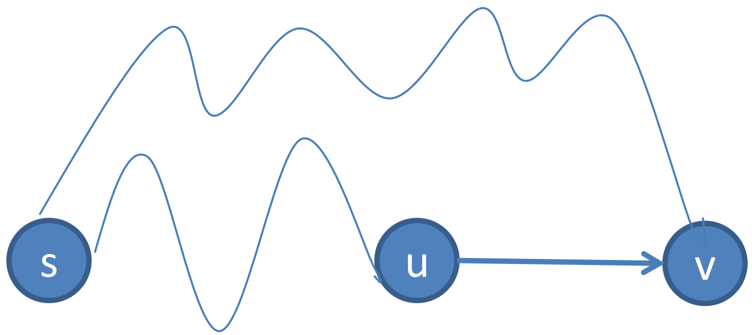
BFS(G,s)

```
for each vertex u in V[G] – {s}
        do color[u] ← white
            d[u] ← ∝
            π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
                for each v in 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
```

Lemma 1

Let $G = (V, E)$ be a directed or undirected graph, and let $s \in V$ be an arbitrary vertex. Then, for any edge $(u, v) \in E$,

$$\delta(s, v) \le \delta(s, u) + 1 .$$



Lemma 2

Let G = (V,E) be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex s ∈ V . Then upon termination, for each vertex ∈ V , the value v.*d* computed by BFS satisfies v.*d* *>=δ(s,v)*.

```
BFS(G,s)
for each vertex u in V[G] – {s}
        do color[u] ← white
           d[u] ← ∝
           π[u] ← nil
color[s] ← gray
d[s] ← 0
π[s] ← nil
Q ← Φ
enqueue(Q,s)
while Q ≠ Φ
        do u ← dequeue(Q)
                for each v in 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
```

- For initial case,
  - $\delta(s,s)=0$
  - $s.d=0$
  - $s.d>=\delta(s,s)$
  - $v.d=\infty\ for\ all\ v\ \in V-s$
  - $v.d>=\delta(s,v)$

- We are modifying distance of a node when it is explored for the first time
- Assuming node v is being explored through node u and for node u, $u.d >= \delta(s,u)$, we have to show it is true for v as well
- $v.d=u.d+1>= \delta(s,u)+1>= \delta(s,v)$

Lemma 3

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \ldots, r - 1$.

- After initialization, there is only s in Q, so there is no violation
- Assuming at certain stage there are $v_1, v_2, \ldots, v_r$ are there in Q and it satisfies the constraints. We have to show dequeue and enqueue does not violate this condition.
- Vr.d<=v1.d+1<=v2.d+1
- Let's say v_{r+1} node is going to be enqueued through node u.
- u.d <=v_1.d
- V_{r+1}.d=u.d+1<=v_1.d+1

Lemma 4

Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \leq v_j.d$ at the time that v_j is enqueued.

Lemma 5

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ for all $v \in V$. Moreover, for any vertex $v \neq s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

- Assume v is assigned a distance which is not equal to its shortest path distance and v is such node with shortest $\delta(s,v)$.
- As we have already shown $v.d >= \delta(s,v)$, it implies that $v.d > \delta(s,v)$
- Let u is the preceding node in shortest path from s to v. However, $u.d = \delta(s,u)$
- $v.d > \delta(s,v) = \delta(s,u)+1 = u.d+1$

- v is white:
  - V.d=u.d+1
- v is black:
  - v is already dequeued and hence as per lemma 4, v.d<=u.d
- v is grey:
  - Assume v is colored grey when node w was dequeued. w.d=v.d-1
  - w.d<=u.d
  - v.d-1<=u.d
  - V.d<=u.d+1

Breadth-first Tree

- For a graph $G = (V, E)$ with source s, the **predecessor subgraph** of G is $G_\pi = (V_\pi, E_\pi)$ where
  - $V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$
  - $E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$
- The predecessor subgraph G_π is a **breadth-first tree** :
  - V_π consists of the vertices reachable from s and
  - for all $v \in V_\pi$, there is a unique simple path from s to v in G_π that is also a shortest path from s to v in G.
- The edges in E_π are called **tree edges**.
  $|E_\pi| = |V_\pi| - 1$.

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex *v*.
- When all edges of *v* have been explored, backtrack to explore other edges leaving the vertex from which *v* was discovered (its *predecessor*).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.

Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 **timestamps** on each vertex. Integers between 1 and 2|V|.
    - $d[v]$ = ***discovery time*** (v turns from white to gray)
    - $f[v]$ = ***finishing time*** (v turns from gray to black)
  - $\pi[v]$: predecessor of $v = u$, such that v was discovered during the scan of u's adjacency list.
- Uses the same coloring scheme for vertices as BFS.

Pseudo-code

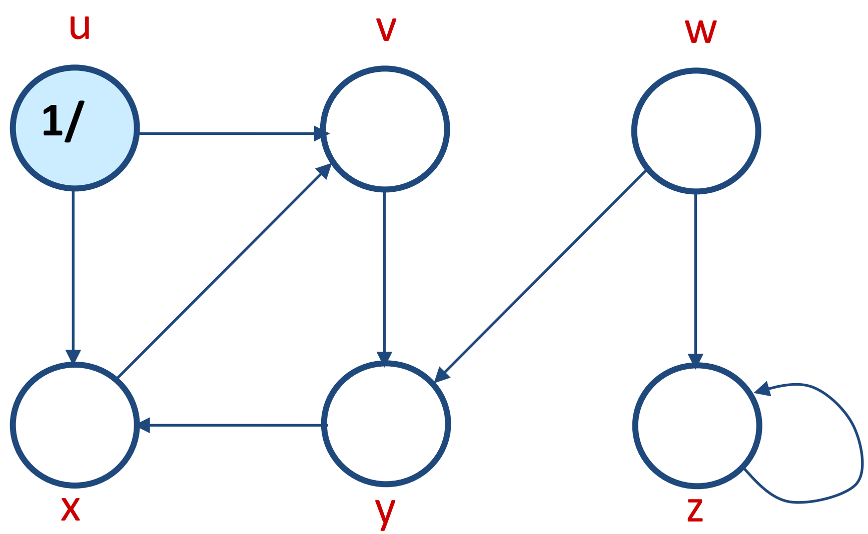
```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← white
        π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = white
        then DFS-Visit(u)
```

```
DFS-Visit(u)
color[u] ← GRAY  ∇ White vertex u has been discovered
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
        then π[v] ← u
            DFS-Visit(v)
color[u] ← BLACK   ∇ Blacken u; it is finished.
f[u] ← time ← time + 1
```

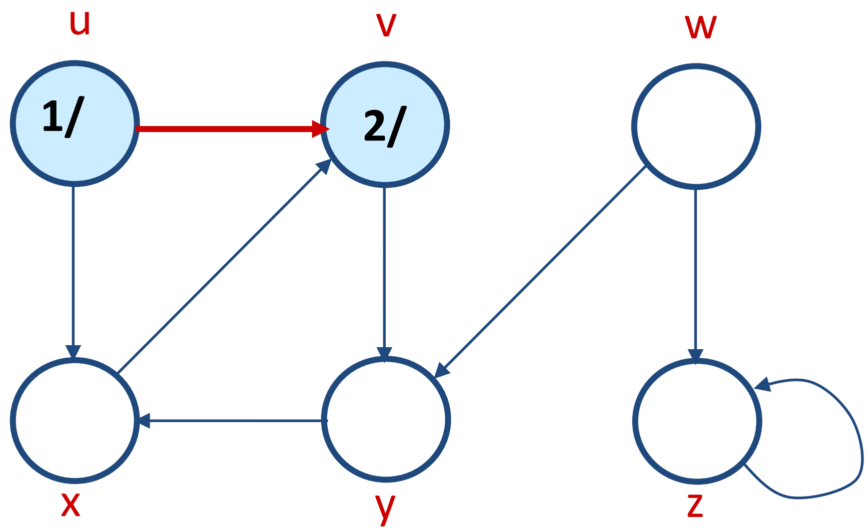
Uses a global timestamp ***time***.

Example: animation.

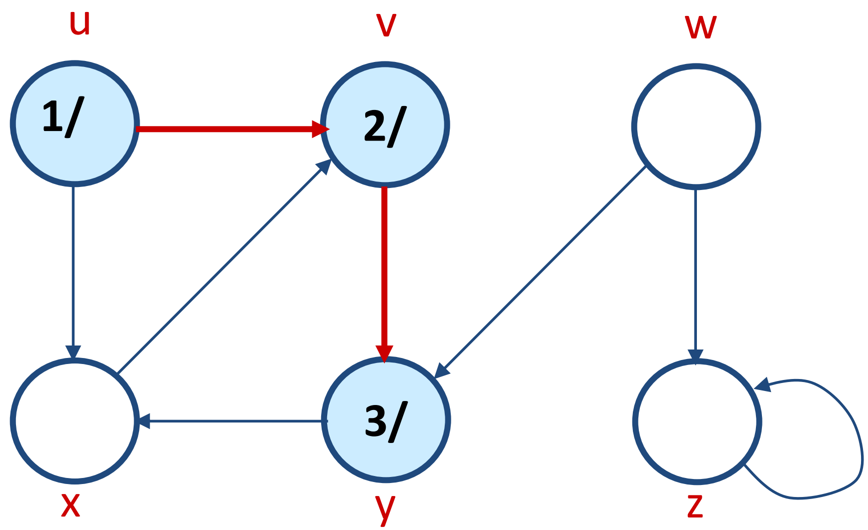
Example (DFS)



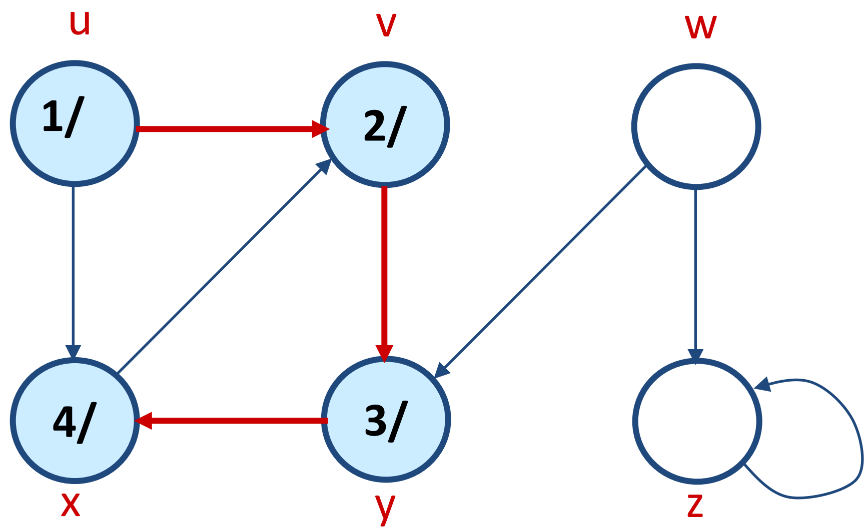
Example (DFS)



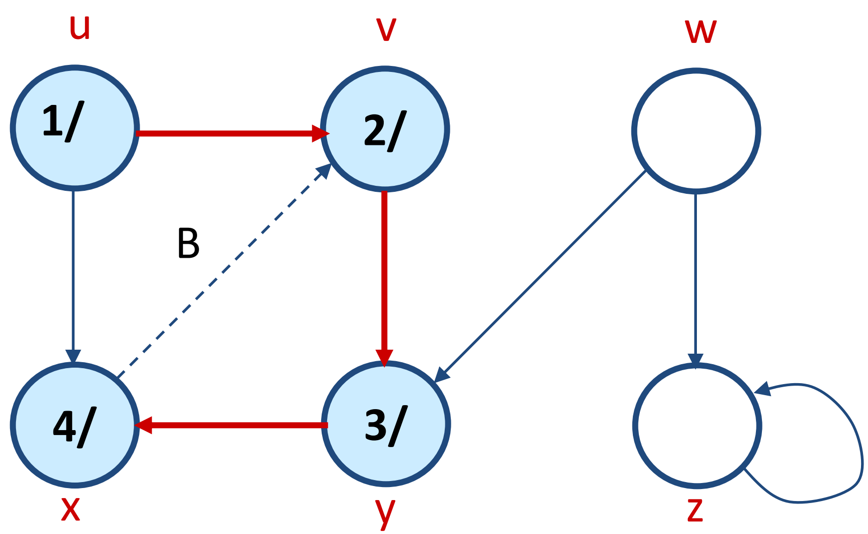
Example (DFS)



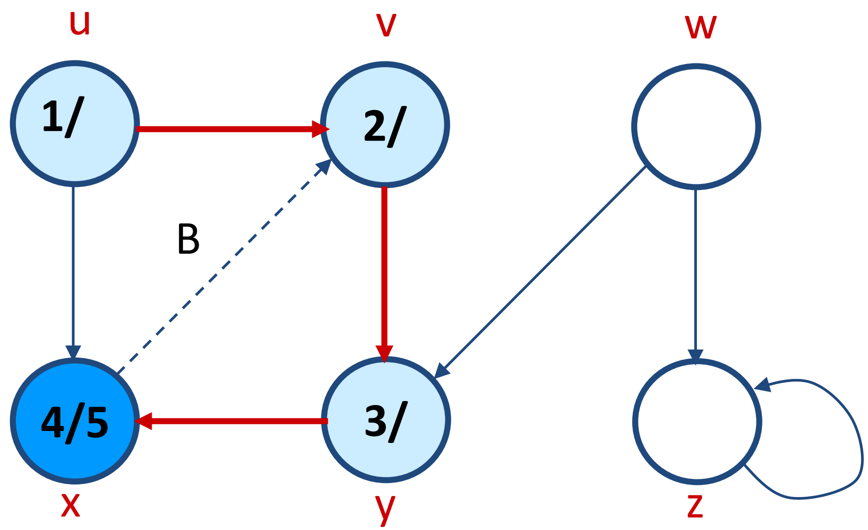
Example (DFS)



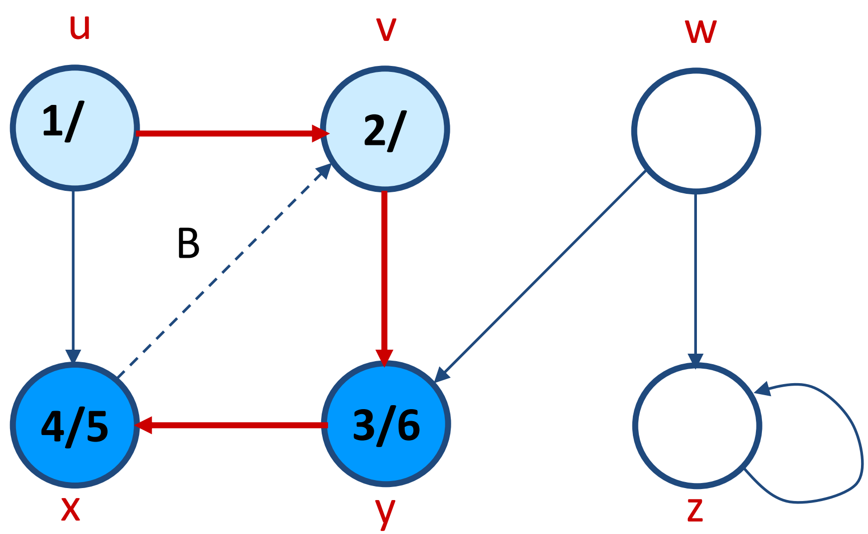
Example (DFS)



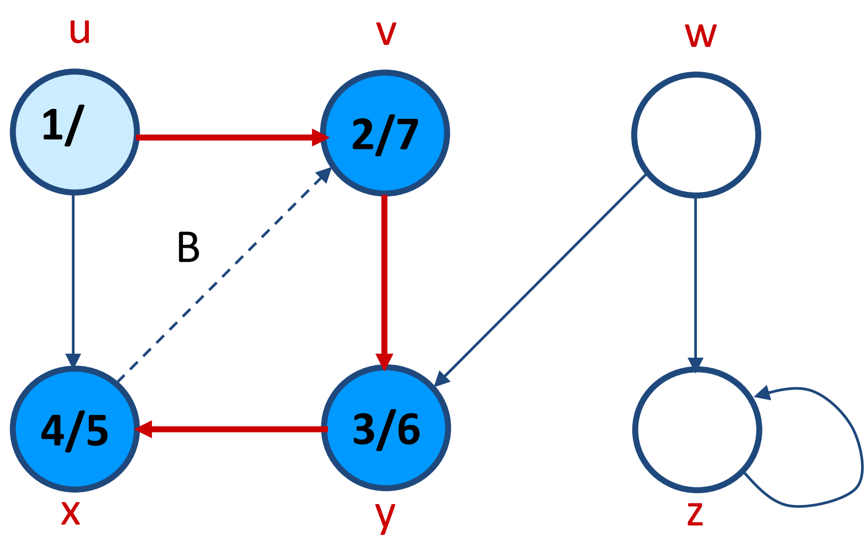
Example (DFS)



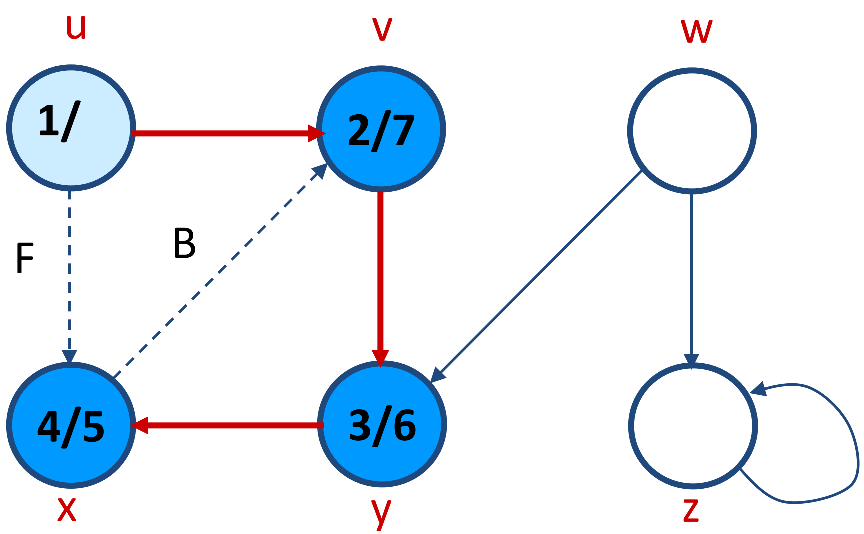
Example (DFS)



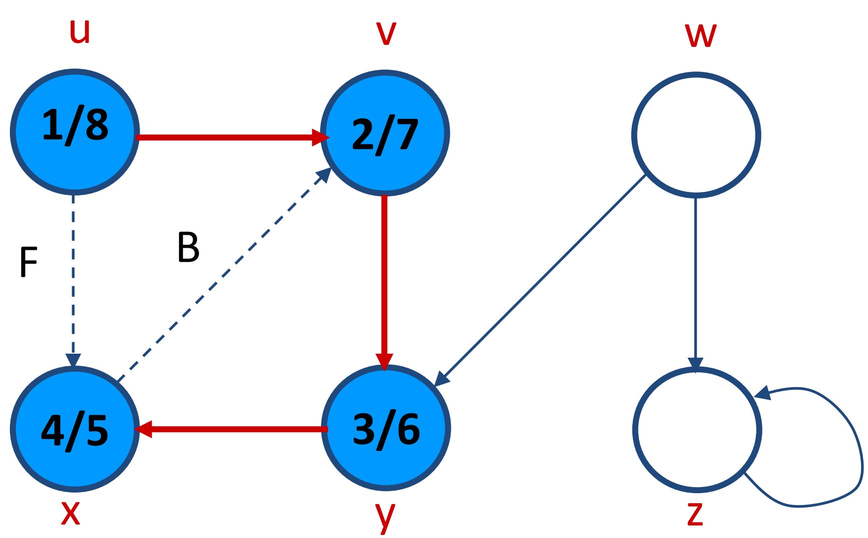
Example (DFS)



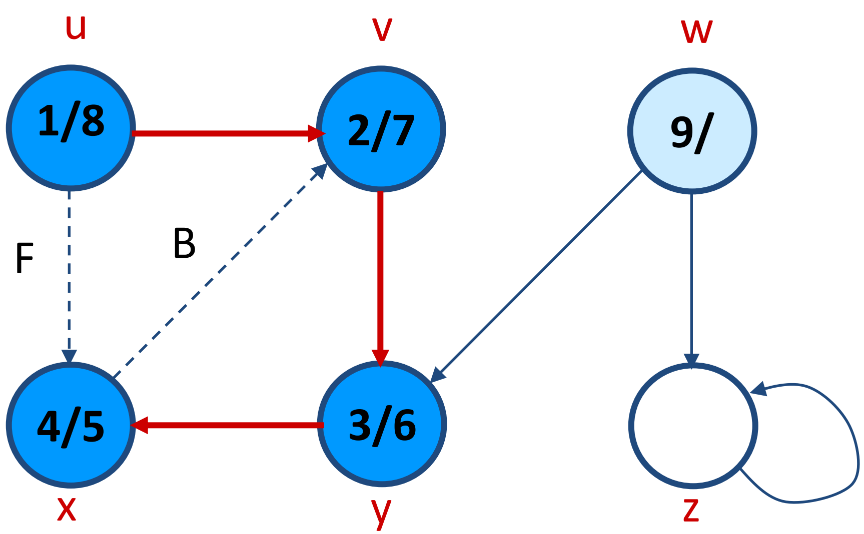
Example (DFS)



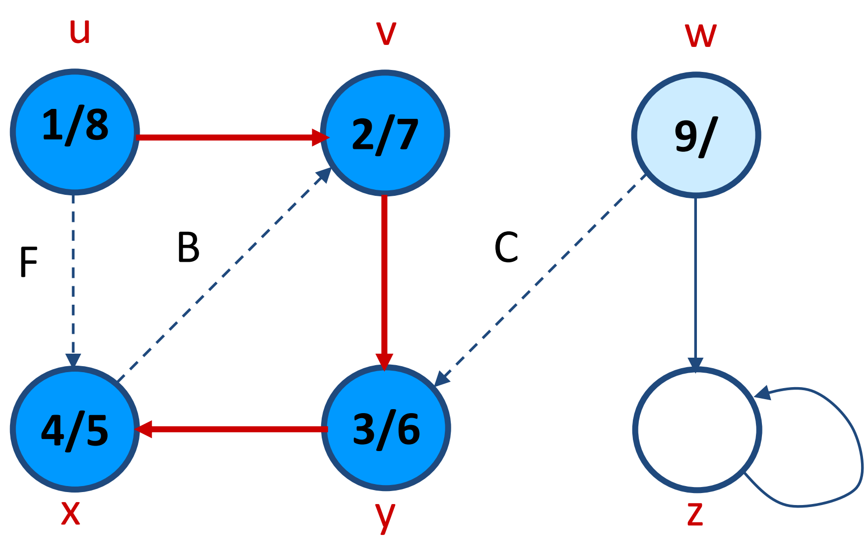
Example (DFS)



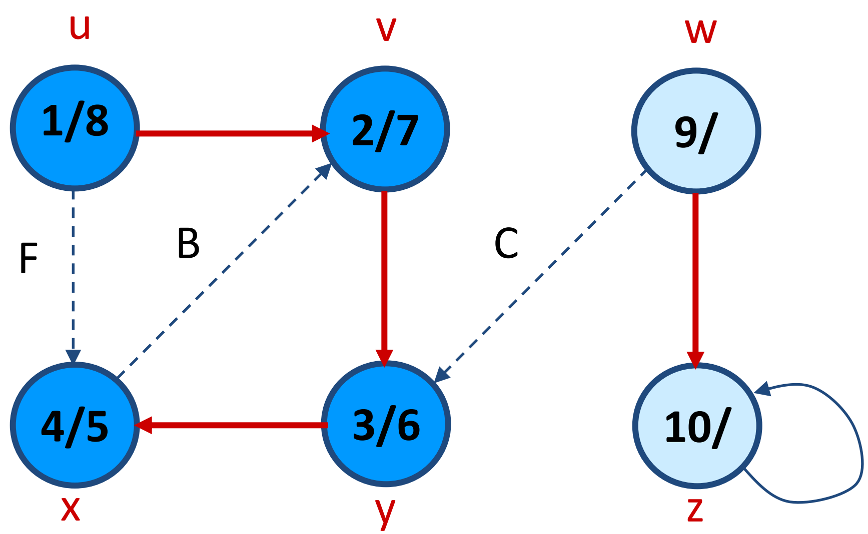
Example (DFS)



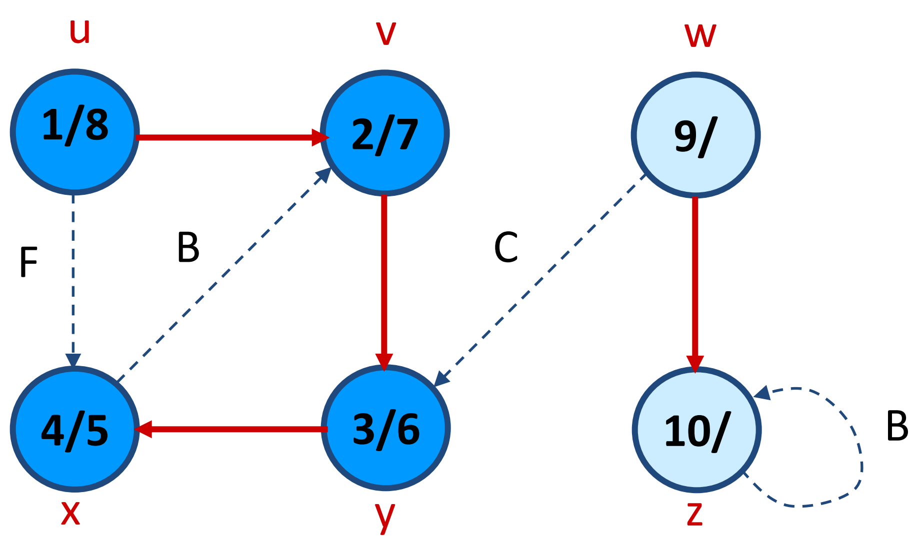
Example (DFS)



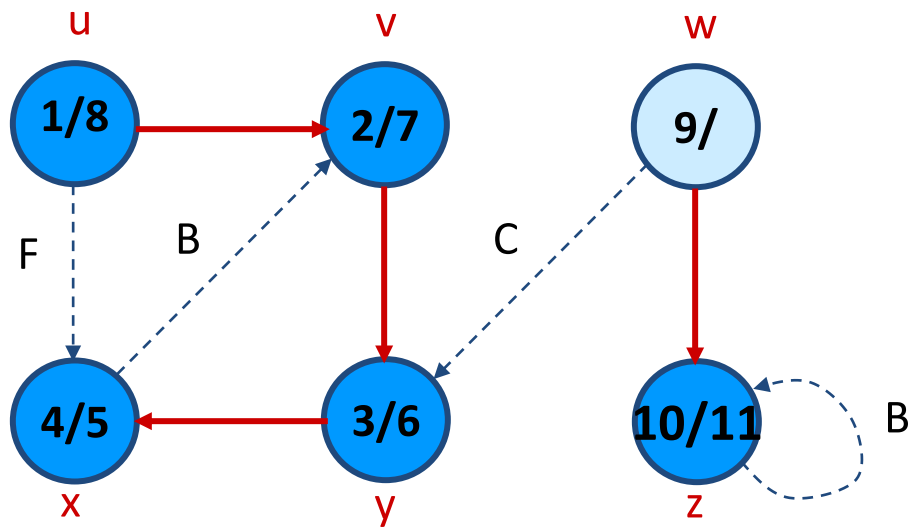
Example (DFS)



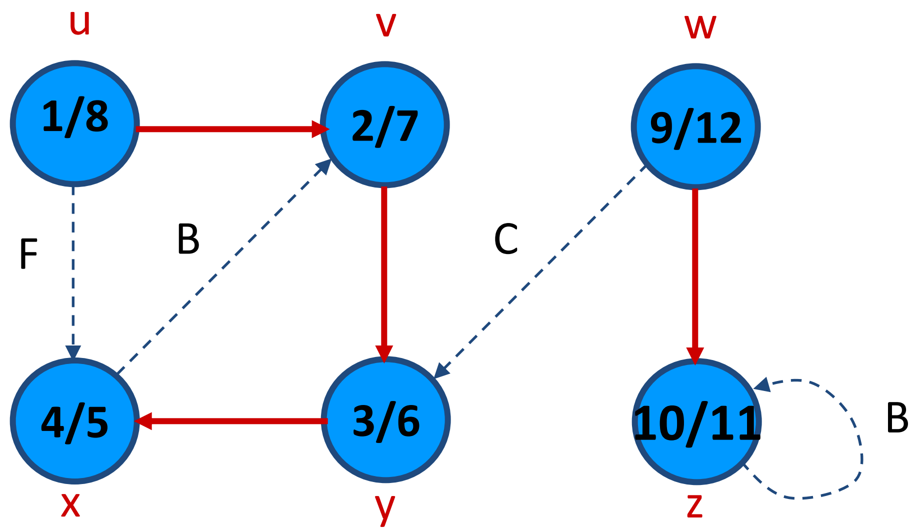
Example (DFS)



Example (DFS)



Example (DFS)



Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it's painted gray the first time. Lines 4-7 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$
- Total running time of DFS is $\Theta(V+E)$.

<u>DFS(*G*)</u>

```
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)
```

DFS-Visit(u)

```
1. color[u] ← GRAY   ∇ White vertex
         u has been discovered
2. time ← time + 1
3. d[u] ← time
4. for each v ∈ Adj[u]
5.     do if color[v] = WHITE
6.          then π[v] ← u
7.               DFS-Visit(v)
8. color[u] ← BLACK     ∇ Blacken u;
         it is finished.
9.  f[u] ← time ← time + 1
```

Parenthesis Theorem

Theorem 22.7

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

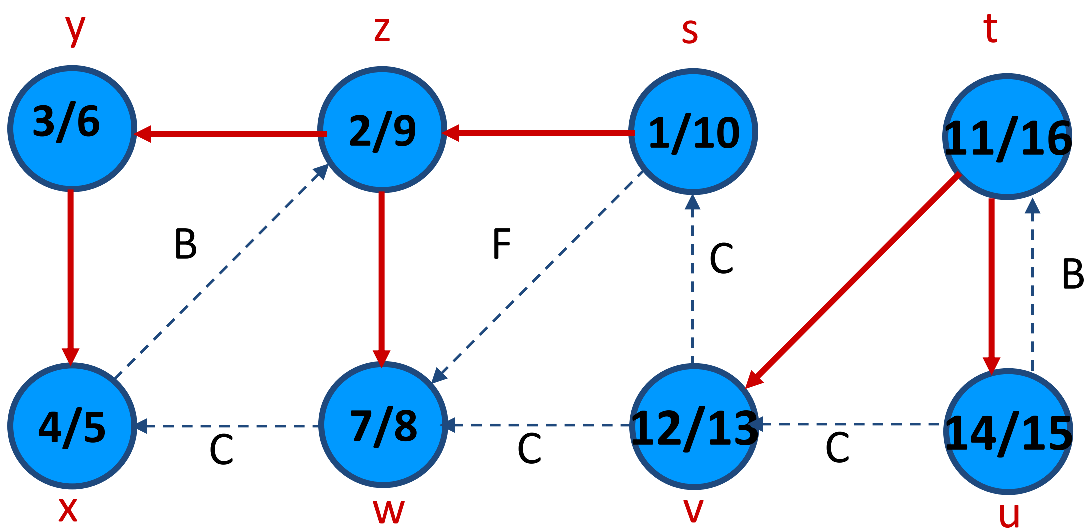
1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other. => ()[] or [] ()
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.=>([])
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.=>[()]

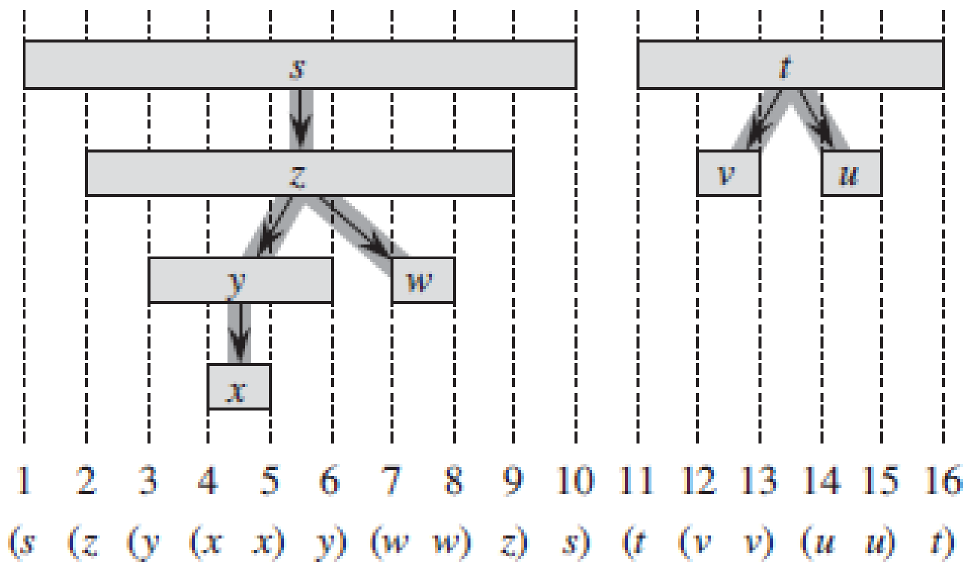
- So $d[u] < d[v] < f[u] < f[v]$ *cannot* happen.
- Like parentheses:
  - OK: ( ) [ ] ([ ]) [( )]
  - Not OK: ([ )] [( ])

Corollary

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

Example (Parenthesis Theorem)





Depth-First Trees

- Predecessor subgraph defined slightly different from that of BFS.
- The predecessor subgraph of DFS is $G_\pi = (V, E_\pi)$ where $E_\pi = \{(\pi[v], v) : v \in V \text{ and } \pi[v] \neq \text{NIL}\}$.
  - How does it differ from that of BFS?
  - The predecessor subgraph G_π forms a *depth-first forest* composed of several *depth-first trees*. The edges in E_π are called *tree edges*.

Definition:

Forest: An acyclic graph G that may be disconnected.

White-path Theorem

Theorem 22.9

v is a descendant of u if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ 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 (in the depth-first tree).
- **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 same depth-first tree or in different depth-first trees.

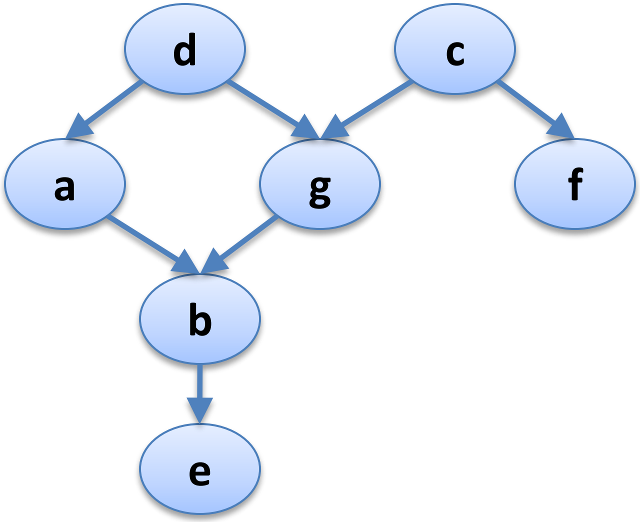
Theorem:
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Topological sort

- We have a **set of tasks** and a **set of dependencies (precedence constraints)** of form “task A must be done before task B”
- **Topological sort**: An ordering of the tasks that conforms with the given dependencies
- **Goal**: Find a topological sort of the tasks or decide that there is no such ordering

Examples

- **Scheduling**: When scheduling *task graphs* in distributed systems, usually we first need to sort the tasks topologically
  ...and then assign them to resources
- Or during compilation to order modules/libraries



Examples

- **Resolving dependencies**: *apt-get* uses topological sorting to obtain the admissible sequence in which a set of Debian packages can be installed/removed

Topological sort more formally

- Suppose that in a **directed** graph **G = (V, E)** vertices **V** represent tasks, and each edge (**u**, **v**)∈**E** means that task **u** must be done before task **v**

- What is an ordering of vertices 1, ..., |**V**| such that for every edge (**u**, **v**), **u** appears before **v** in the ordering?

- Such an ordering is called a **topological sort of G**
- Note: there can be multiple topological sorts of G

Topological sort more formally

- Is it possible to execute all the tasks in **G** in an order that respects all the precedence requirements given by the graph edges?
- The answer is "**yes**" *if and only if* the directed graph **G** has **no cycle**!

  (otherwise we have a **deadlock**)
- Such a **G** is called a Directed Acyclic Graph, or just a **DAG**

Algorithm for TS

- TOPOLOGICAL-SORT(**G**):
  1) call DFS(G) to compute **finishing** times **f**[**v**] for each vertex **v**
  2) as each vertex is finished, insert it onto the **front** of a linked list
  3) return the linked list of vertices

- Note that the result is just a list of vertices in order of **decreasing** finish times **f**[]

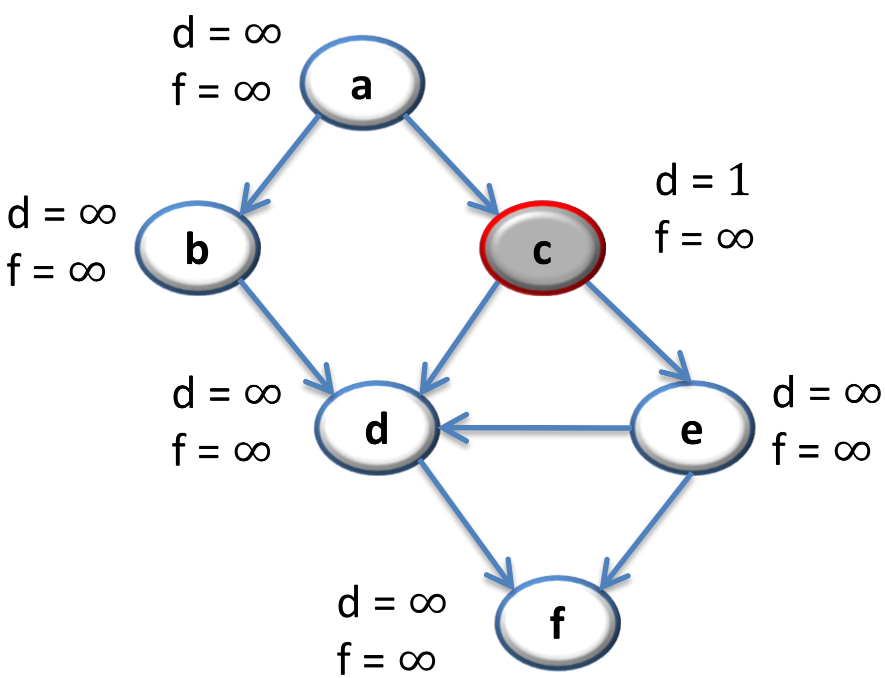
DAGs and back edges

- Can there be a **back** edge in a DFS on a DAG?
- NO! Back edges close a cycle!
- A graph **G** is a DAG <=> there is no back edge classified by DFS(**G**)

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 1

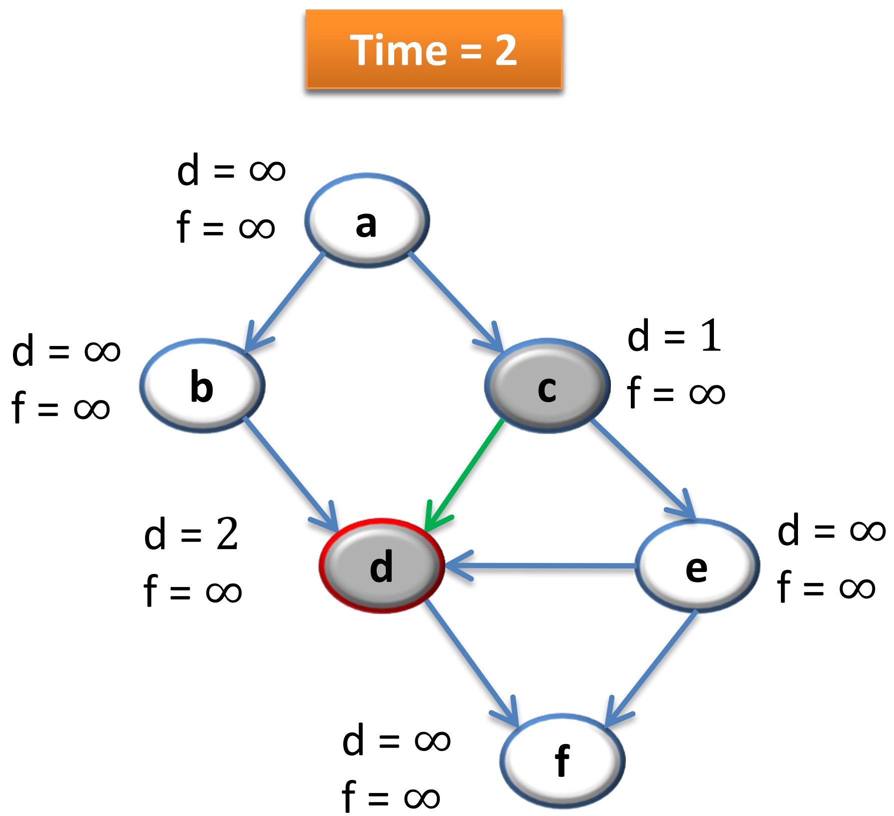


Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

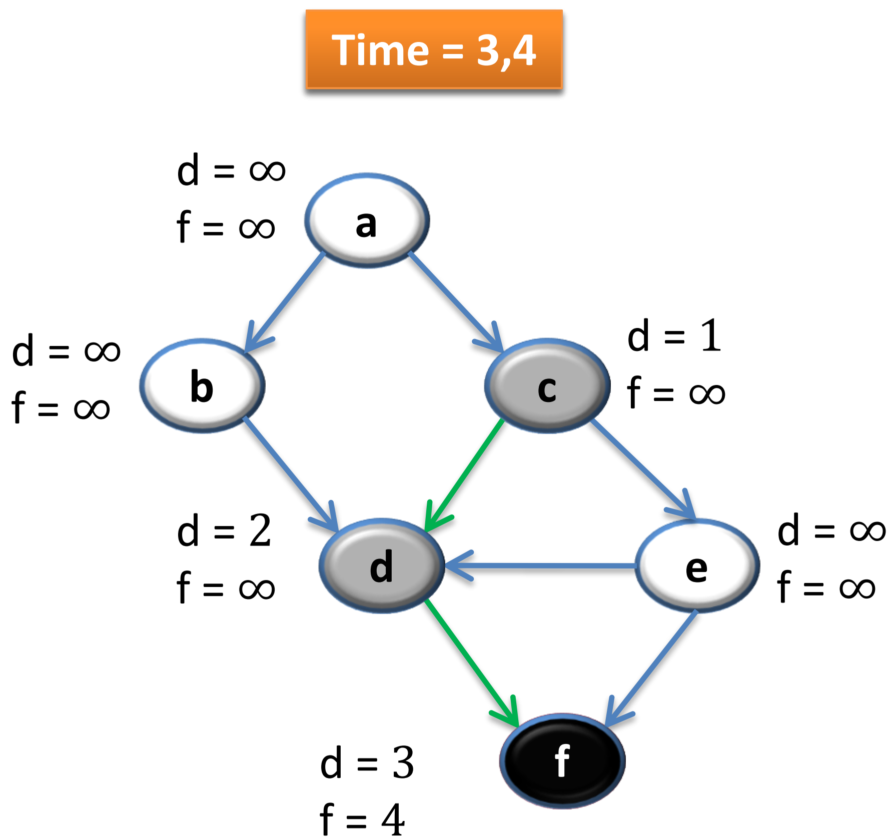
1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let’s say we start the DFS from the vertex **c**

Next we discover the vertex **d**

Topological sort

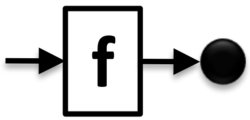


1) Call DFS(**G**) to compute the finishing times **f**[**v**]

2) as each vertex is finished, insert it onto the **front** of a linked list

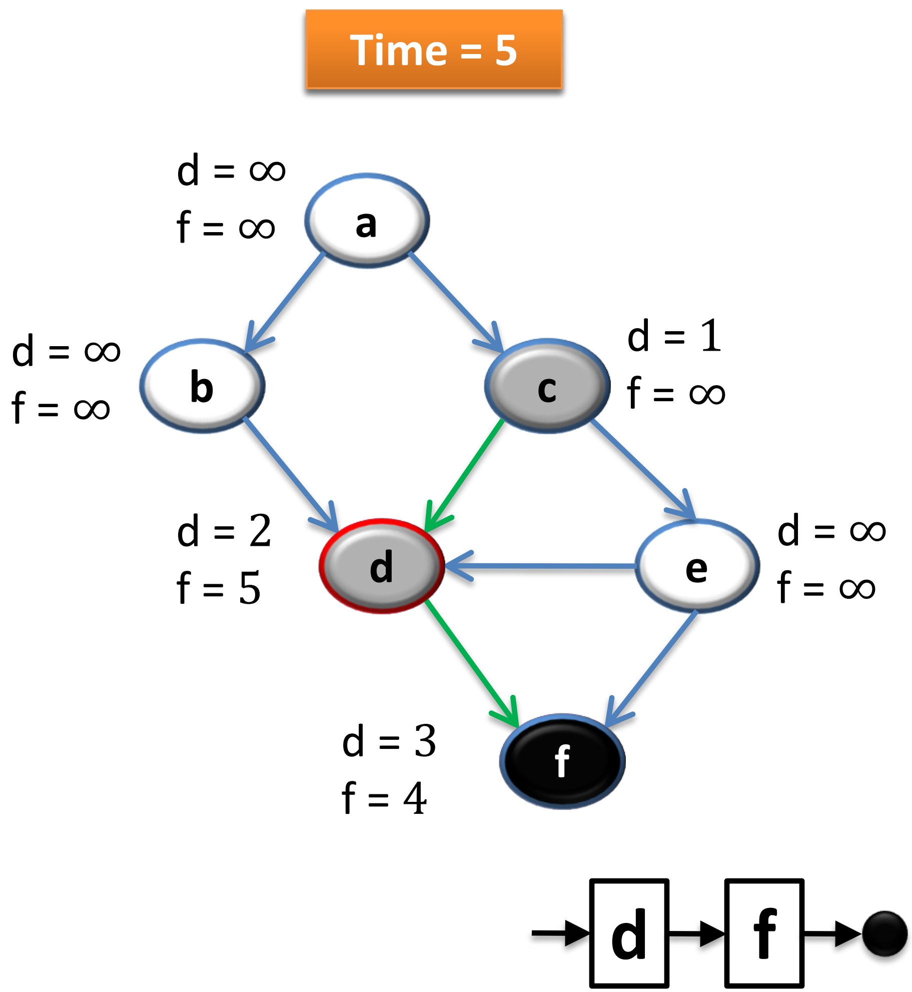
Next we discover the vertex **f**

f is done, move back to **d**



Topological sort

1) Call DFS(**G**) to compute the finishing times **f[v]**



Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

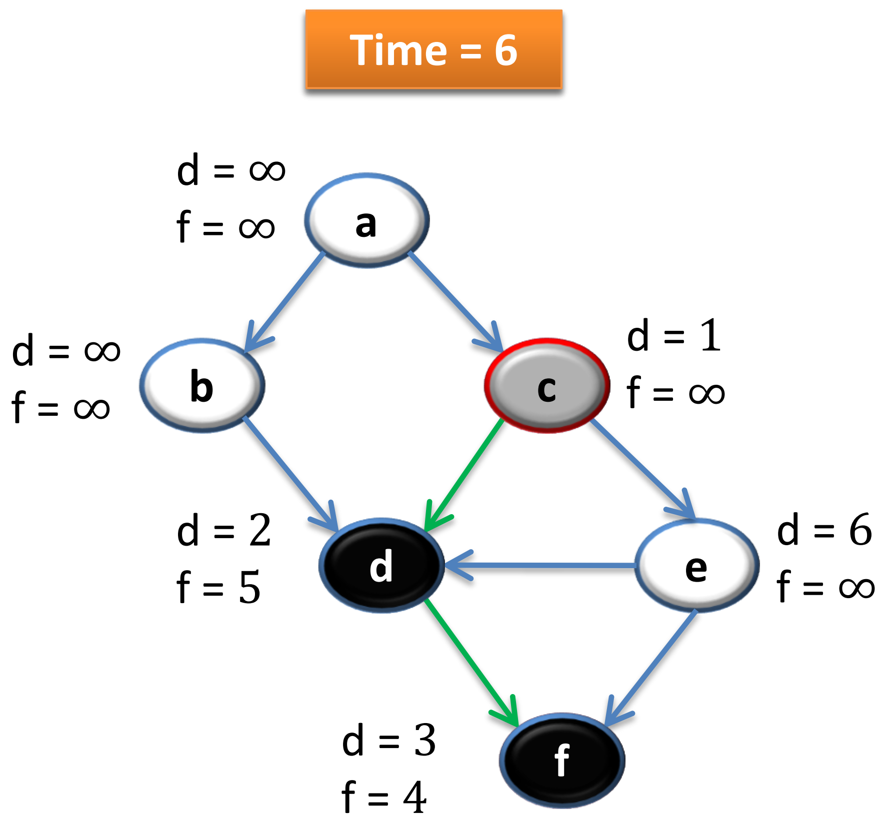
Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's say we start the DFS from the vertex **c**

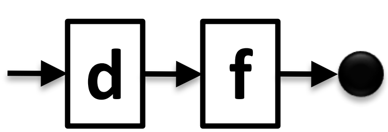
Next we discover the vertex **d**

Next we discover the vertex **f**

f is done, move back to **d**

d is done, move back to **c**

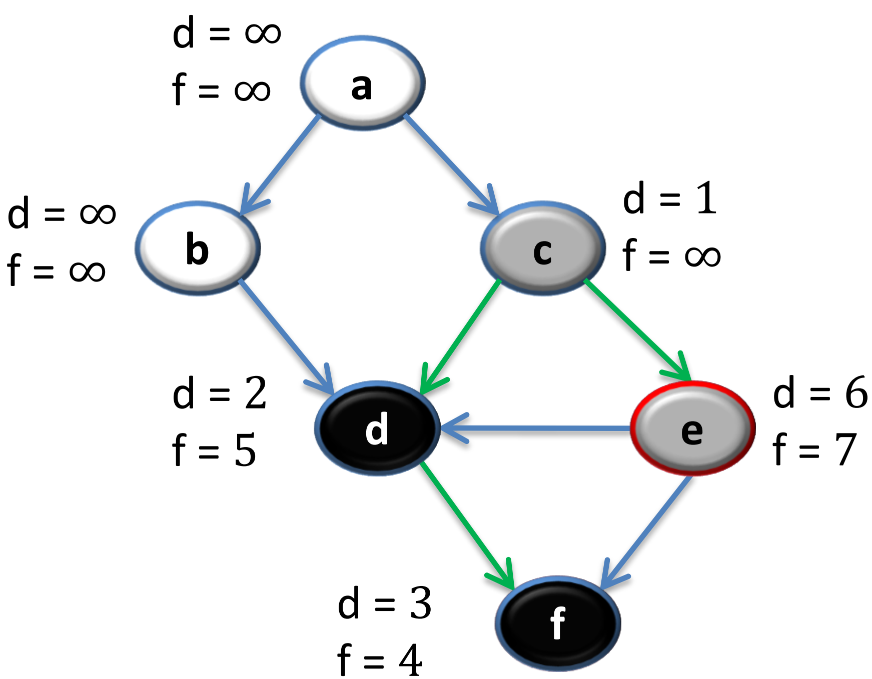
Next we discover the vertex **e**

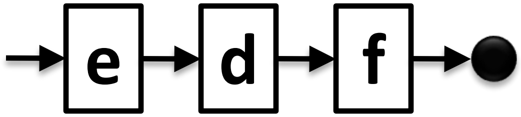


Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 7





Let's say we start the DFS from the vertex **c**

Next we discover the vertex **d**

d is done, move back to **c**

Next we discover the vertex **e**

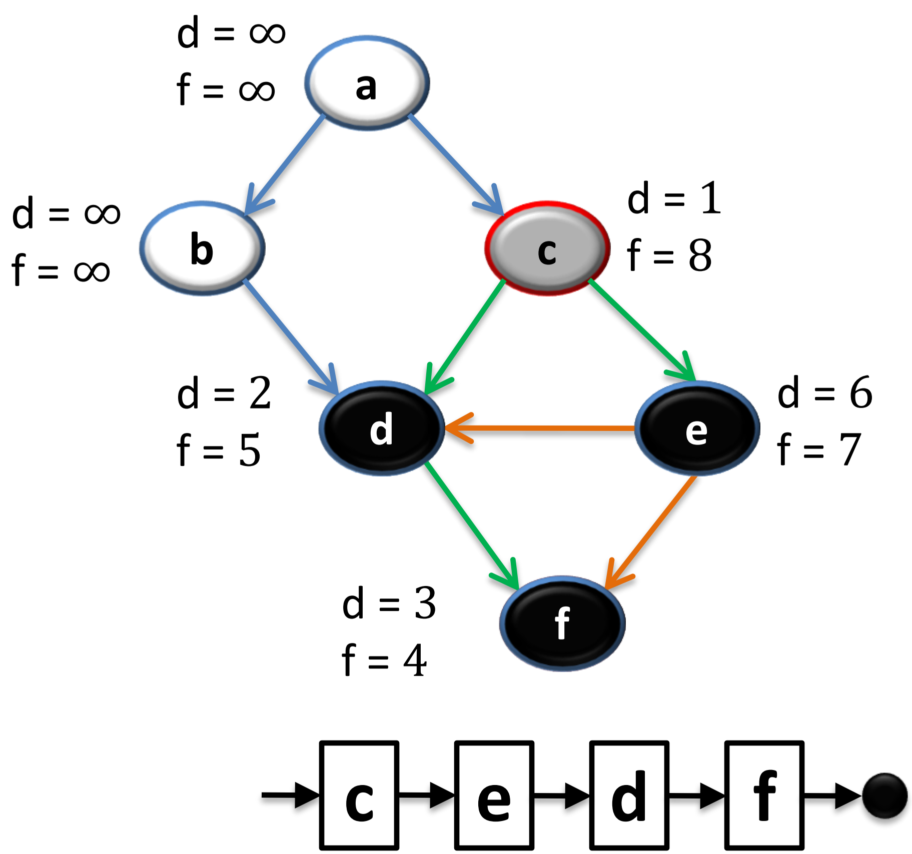
e is done, move back to **c**

Both edges from **e** are **cross edges**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

Time = 8



Let's say we start the DFS from the vertex **c**

<u>Just a note:</u> If there was (**c**,**f**) edge in the graph, it would be classified as a **forward edge** (in this particular DFS run)

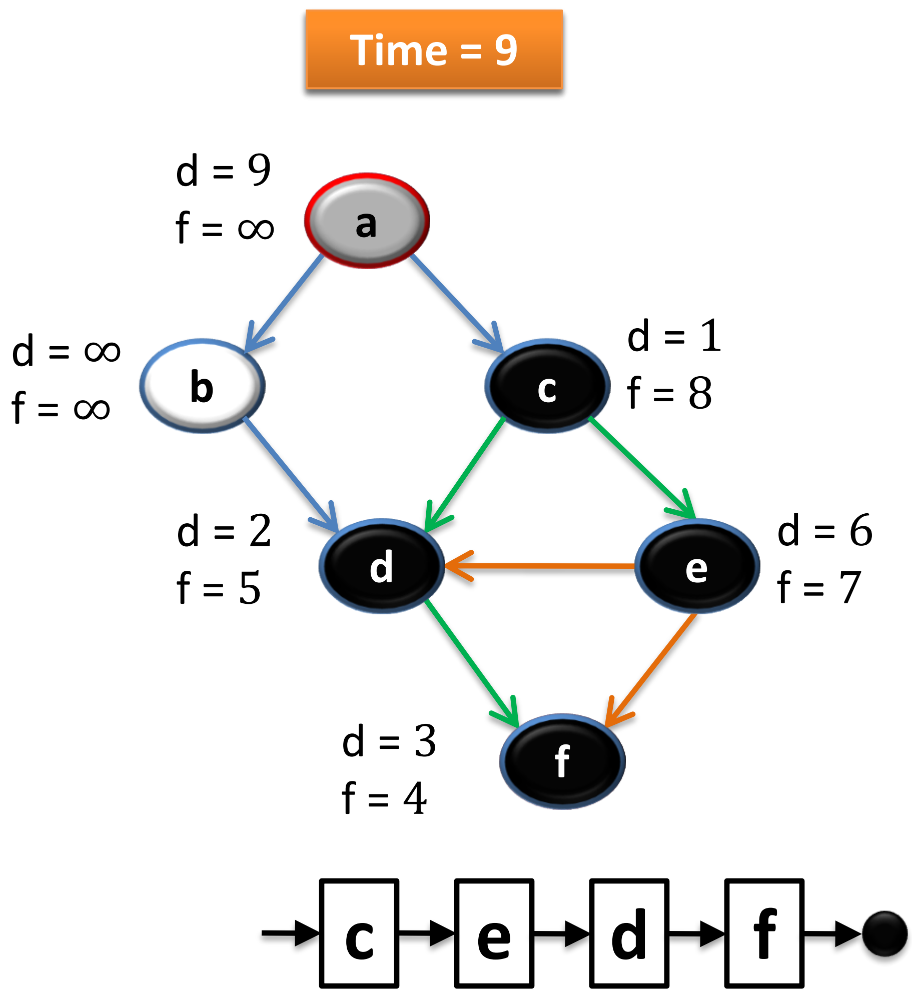
d is done, move back to **c**

Next we discover the vertex **e**

e is done, move back to **c**

c is done as well

Topological sort



1) Call DFS(**G**) to compute the finishing times **f**[**v**]

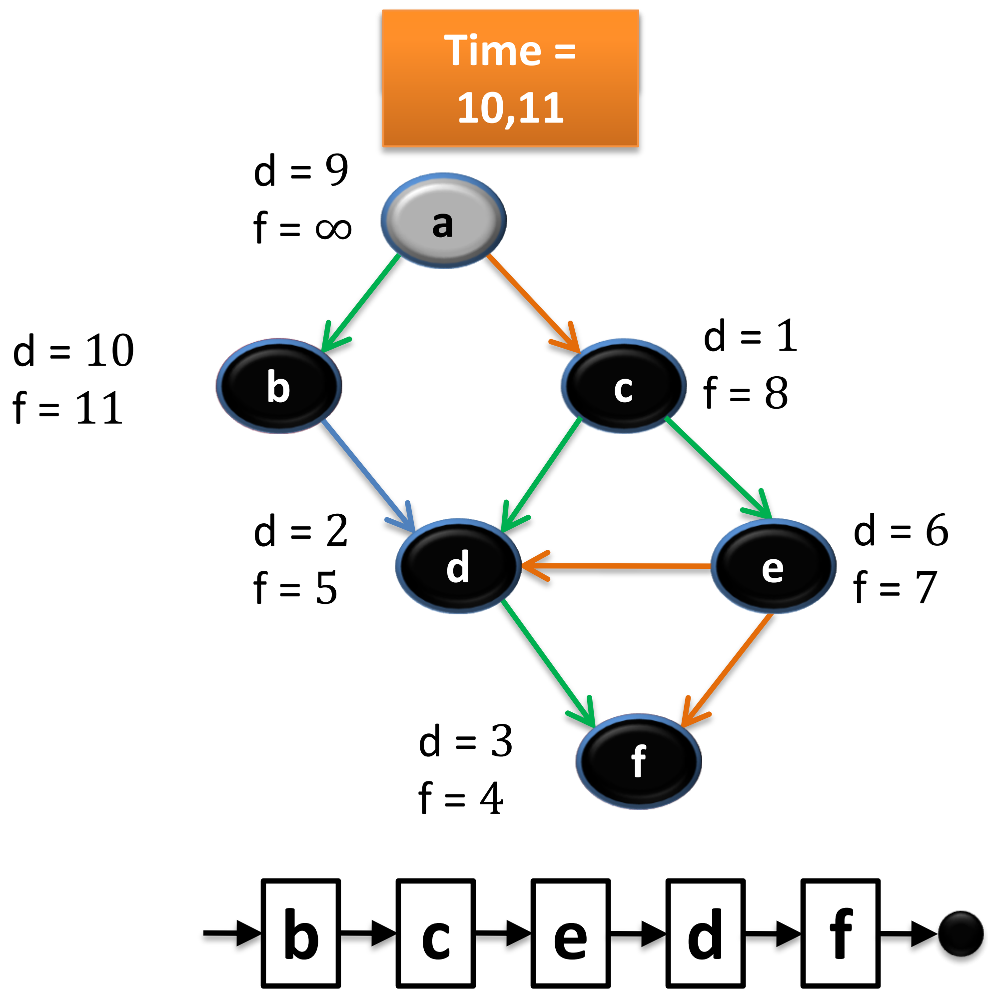
Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's now call DFS visit from the vertex **a**

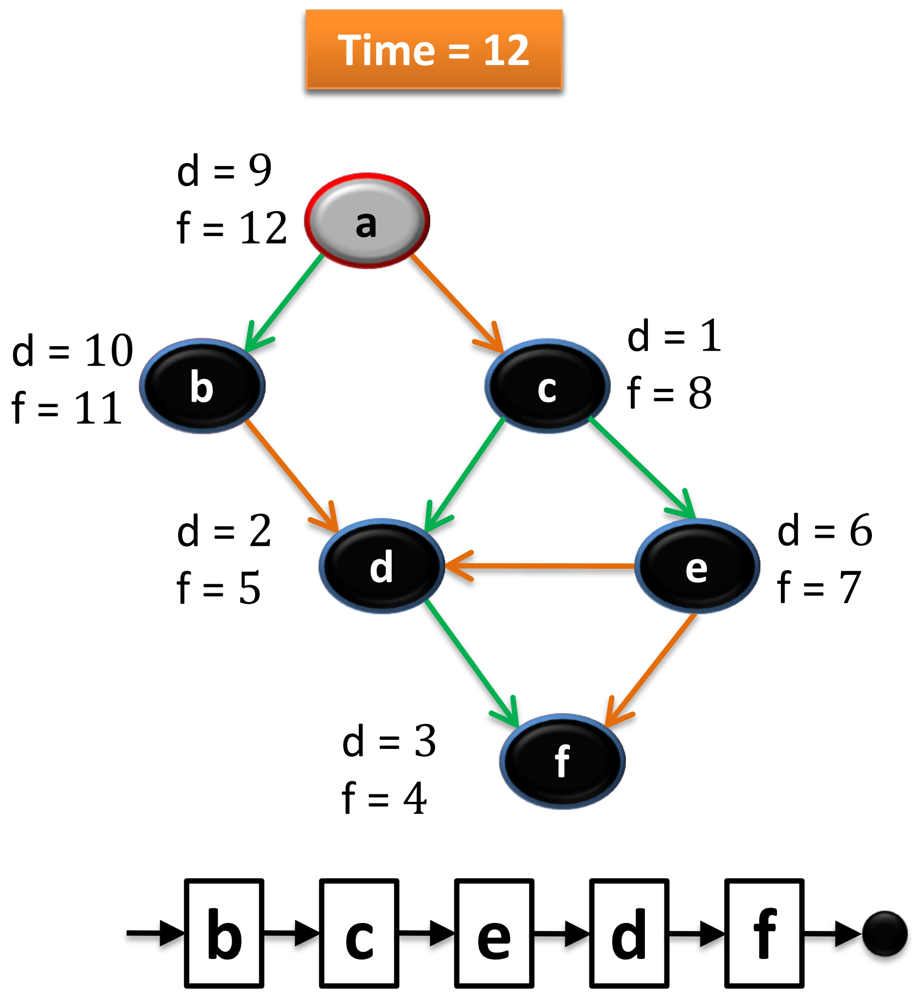
Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]



Let's now call DFS visit from the vertex **a**

Next we discover the vertex **c**, but **c** was already processed => (**a**,**c**) is a cross edge

Next we discover the vertex **b**

b is done as (**b**,**d**) is a cross edge => now move back to **c**

a is done as well

Topological sort

1) Call DFS(**G**) to compute the finishing times **f**[**v**]

WE HAVE THE RESULT!

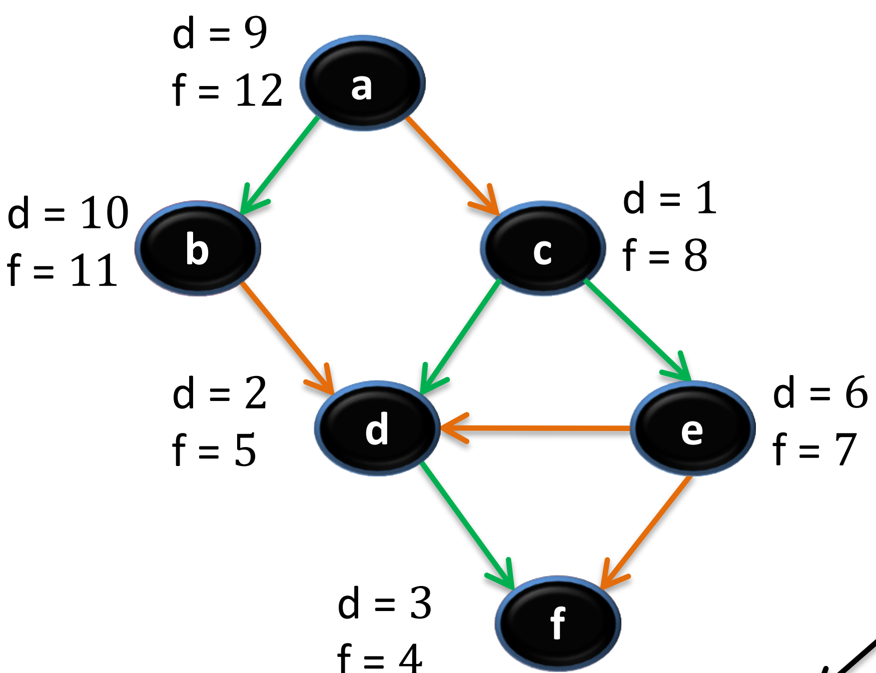
3) return the linked list of vertices

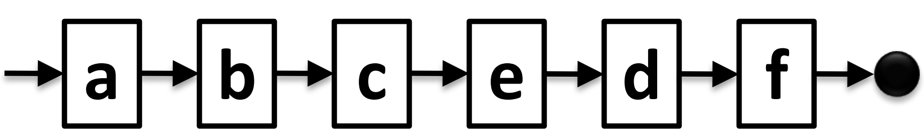
=> (**a**,**c**) is a cross edge

Next we discover the vertex **b**

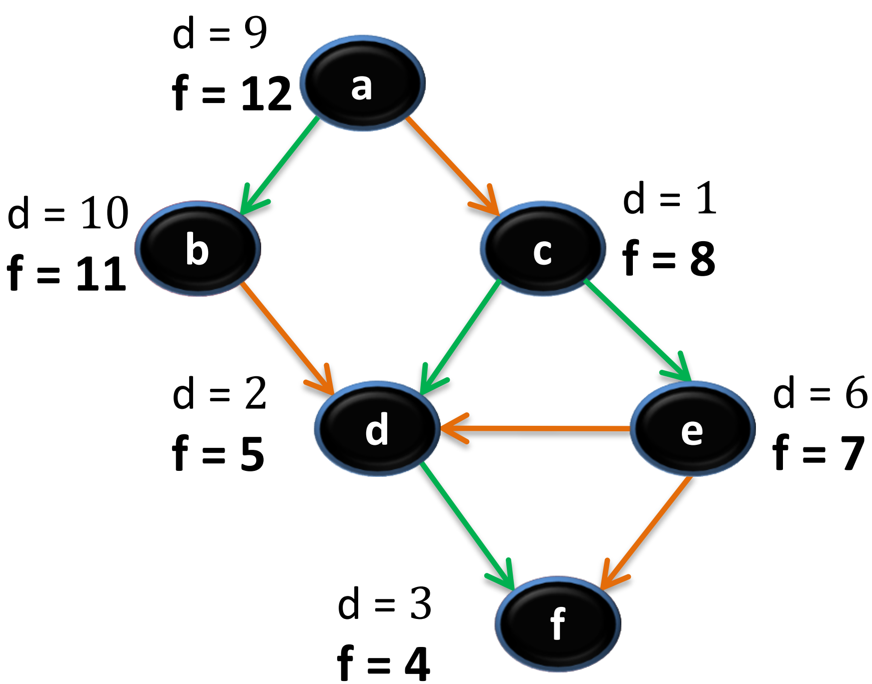
b is done as (**b**,**d**) is a cross edge => now move back to **c**

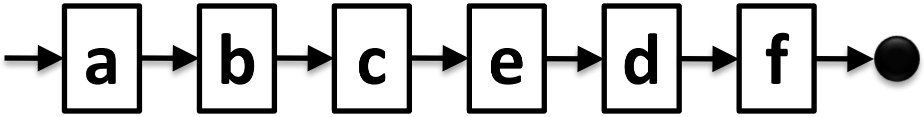
a is done as well





Topological sort





The linked list is sorted in **decreasing** order of finishing times **f**[]

Try yourself with different vertex order for DFS visit

Time complexity of TS(G)

- Running time of topological sort:

  $$\boldsymbol{\Theta(V + E)}$$

  Why? Depth first search takes $\boldsymbol{\Theta(V + E)}$ time in the worst case, and inserting into the front of a linked list takes $\boldsymbol{\Theta(1)}$ time

Proof of correctness

- **Theorem**: TOPOLOGICAL-SORT(**G**) produces a topological sort of a DAG **G**

- The TOPOLOGICAL-SORT(**G**) algorithm does a DFS on the DAG **G**, and it lists the nodes of **G** in order of decreasing finish times **f**[]
- We must show that this list satisfies the topological sort property, namely, that for every edge (**u**,**v**) of **G**, **u** appears before **v** in the list
- **Claim**: For every edge (**u**,**v**) of **G**: **f**[**v**] < **f**[**u**] in DFS

Proof of correctness

“For every edge (**u**,**v**) of **G**, **f**[**v**] < **f**[**u**] in this DFS”

- The DFS classifies (**u**,**v**) as a **tree edge**, a **forward edge** or a **cross-edge** (it cannot be a back-edge since **G** has no cycles):
  1. If (**u**,**v**) is a **tree** or a **forward edge** $\Rightarrow$ **v** is a descendant of **u** $\Rightarrow$ **f**[**v**] < **f**[**u**]
  2. If (**u**,**v**) is a **cross-edge =>** v will be discovered and finished before u hence f[v] < f[u]

Proof of correctness

- TOPOLOGICAL-SORT(G) lists the nodes of G from highest to lowest finishing times

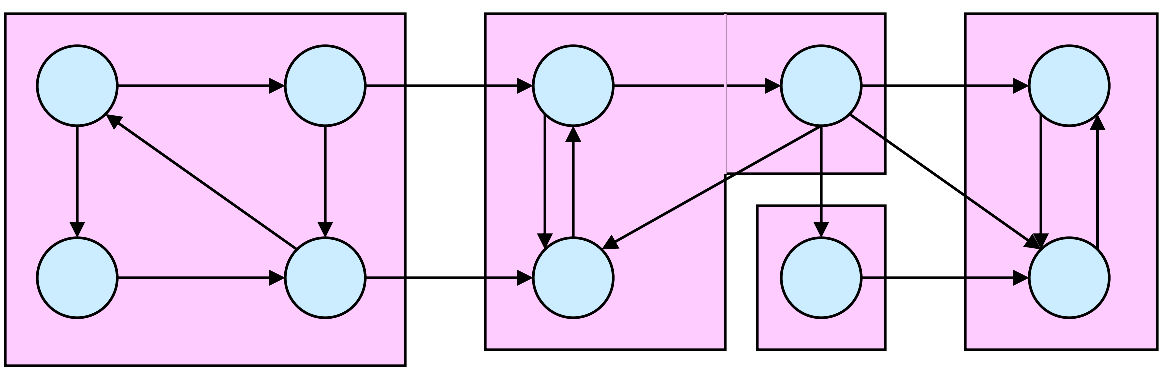
- By the **Claim**, for every edge (**u**,**v**) of **G**:

$$\mathbf{f}[\mathbf{v}] < \mathbf{f}[\mathbf{u}]$$

$\Rightarrow$ **u** will appear before **v** in the sorted list

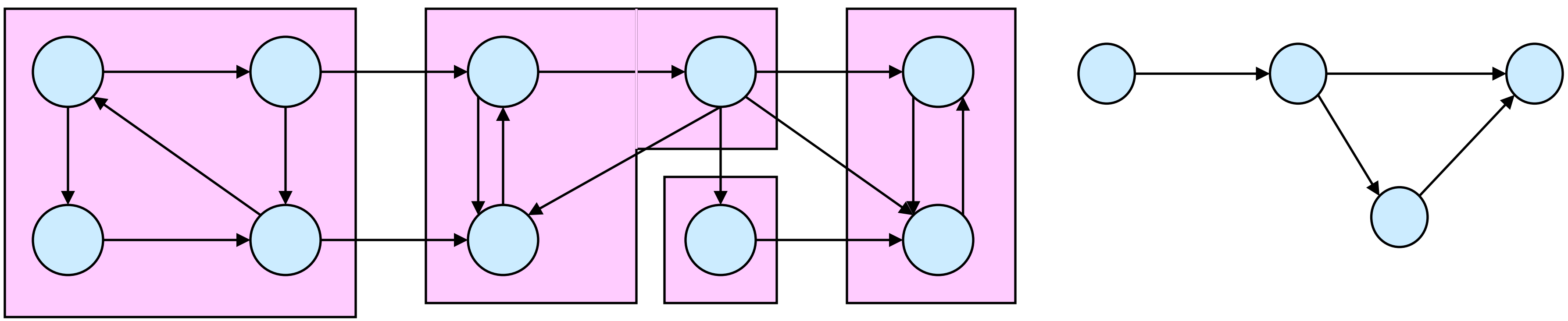
Strongly Connected Components

- G is strongly connected if every pair (u, v) of vertices in G is reachable from one another.
- 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 $u \leadsto v$ and $v \leadsto u$ exist.



Component Graph

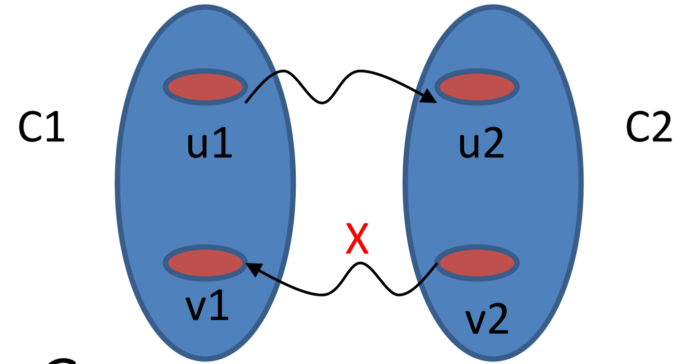
- $G^{SCC} = (V^{SCC}, E^{SCC})$.
- V^{SCC} has one vertex for each SCC in G.
- E^{SCC} has an edge if there's an edge between the corresponding SCC's in G.
- G^{SCC} for the example considered:



G^{SCC} is a DAG

Lemma

Let C and C' be distinct SCC's in G, let $u1$, $v1 \in C1$ and $u2$, $v2 \in C2$, and suppose there is a path $u1 \rightsquigarrow u2$ in G. Then there cannot also be a path $v2 \rightsquigarrow v1$ in G.



Proof:

- Suppose there is a path $v2 \rightsquigarrow v1$ in G.
- Then there are paths $v1 \rightsquigarrow u1 \rightsquigarrow u2 \rightsquigarrow v2$ in G.
- Therefore, $v1$ and $v2$ are reachable from each other, so they are not in separate SCC's.

Transpose of a Directed Graph

- G^{T} = **transpose** of directed G.
  - $G^{\mathrm{T}} = (V, E^{\mathrm{T}}), E^{\mathrm{T}} = \{(u, v) : (v, u) \in E\}$.
  - G^{T} is G with all edges reversed.
- Can create G^{T} in $\Theta(V + E)$ time if using adjacency lists.
- 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}.)

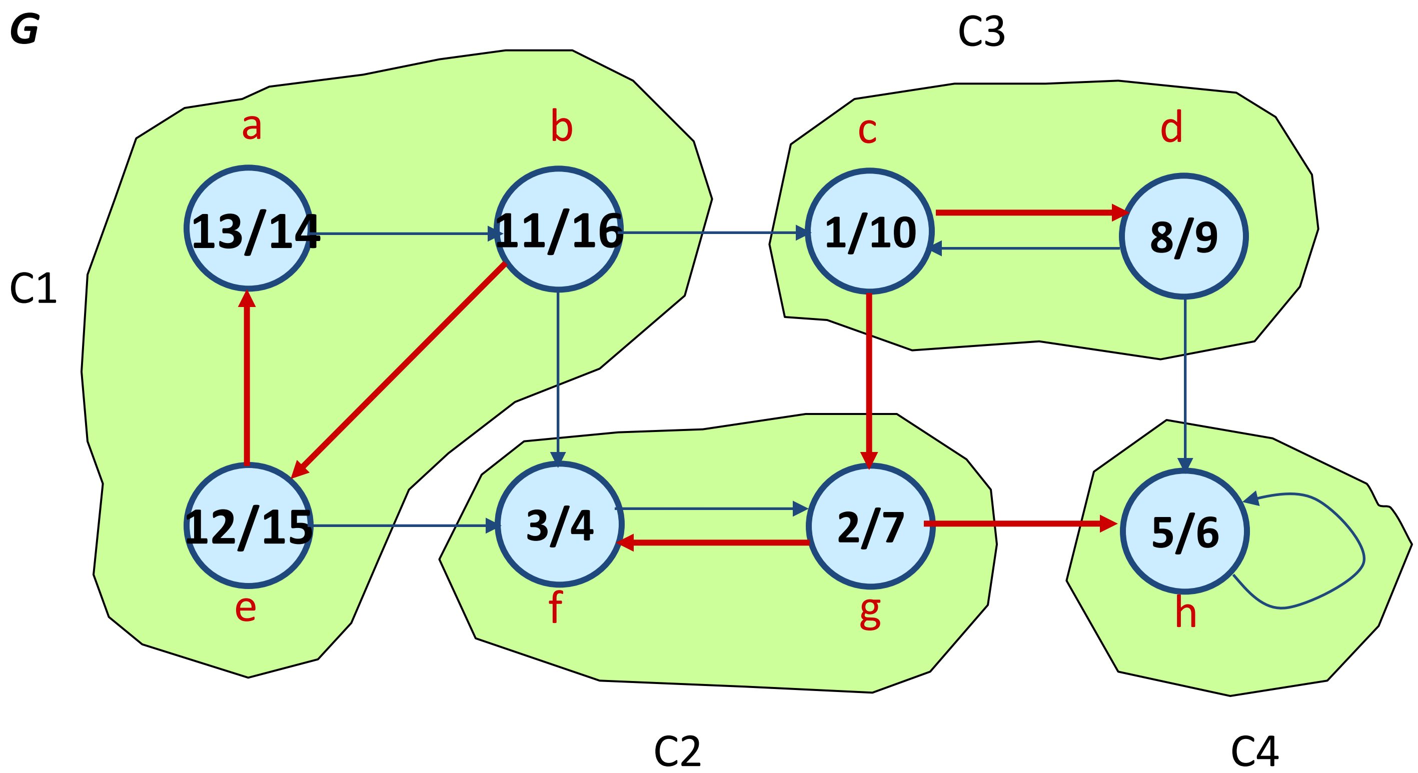
Algorithm to determine SCCs

<u>SCC(G)</u>

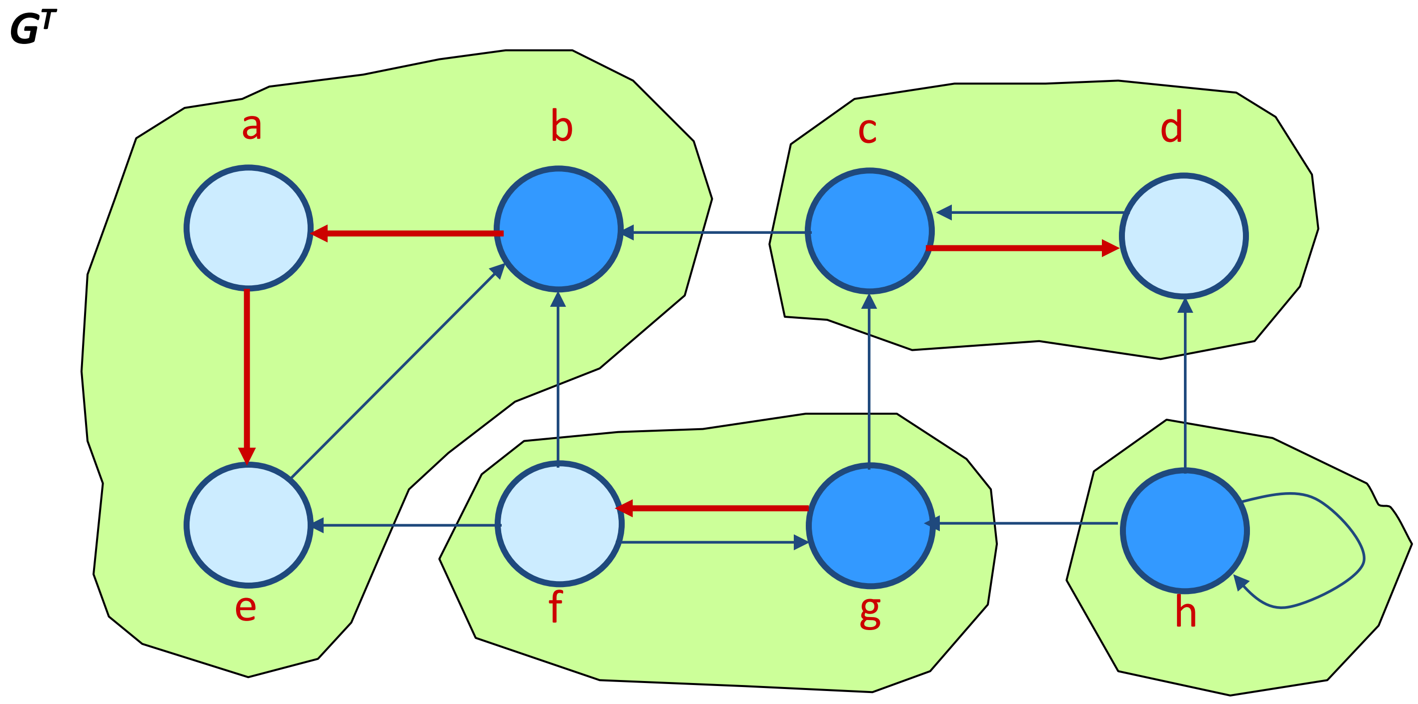
1. call DFS(G) to compute finishing times $f[u]$ for all u
2. compute G^{T}
3. call DFS(G^{T}), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

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

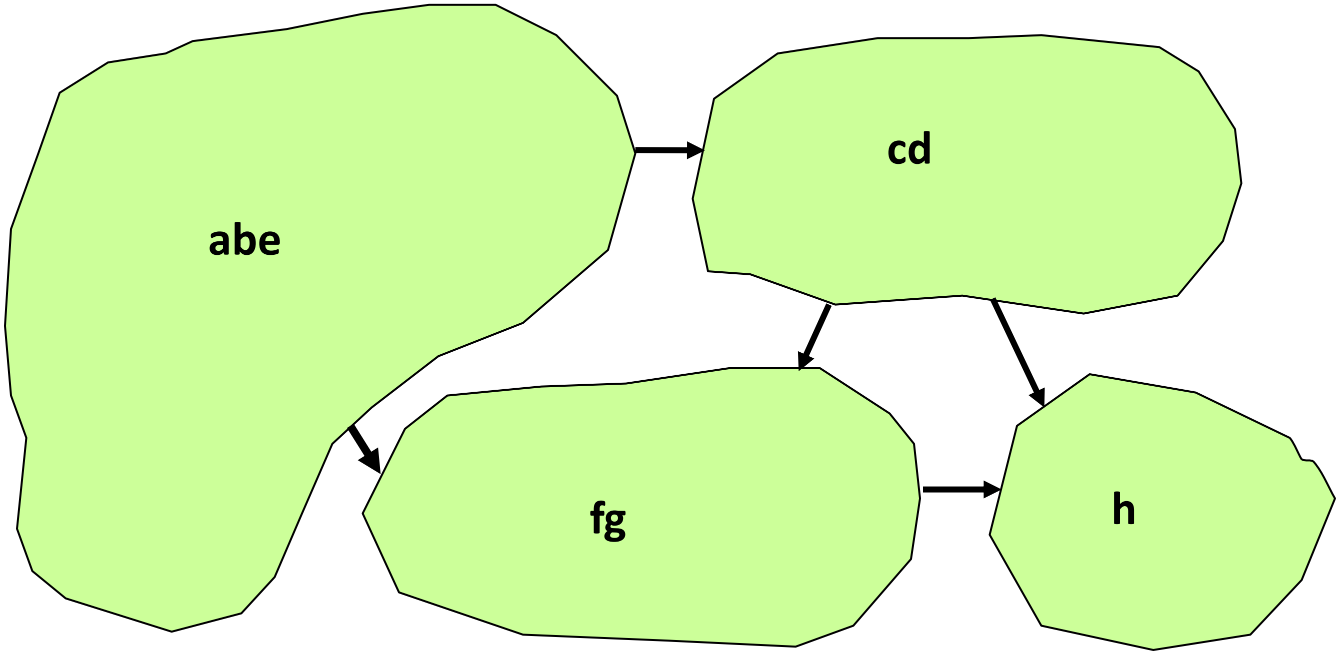
Example



Example



Example



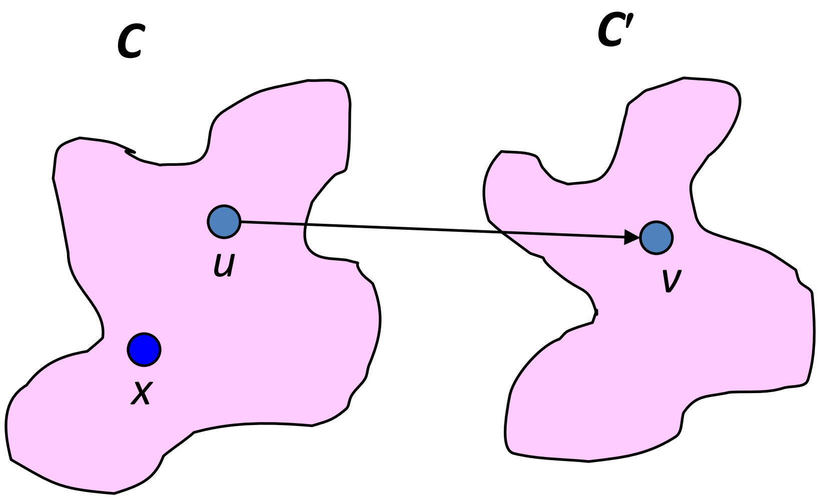
SCCs and DFS finishing times

Lemma 22.14

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:

- Case 1: $d(C) < d(C')$
  - 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, $f[x] = f(C) > f(C')$.



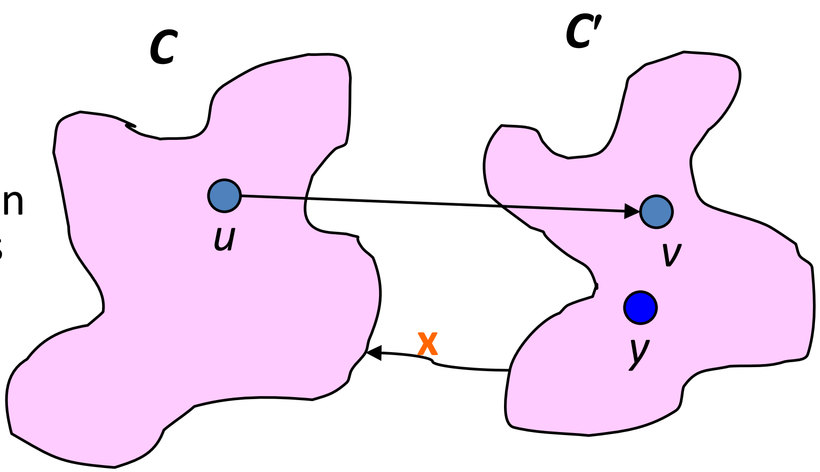
SCCs and DFS finishing times

Lemma 22.14

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:

- Case 2: $d(C) > d(C')$
  - 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 also 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 $f(C) > f(C')$.



Correctness of SCC

- 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 22.15 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.
  - Which means that the depth-first tree rooted at x contains *exactly* the vertices of C.

Correctness of SCC

- 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*.
  - Therefore, the only tree edges will be to vertices in C'.
- We can continue the process.
- 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.