Elementary Graph Algorithms

Dr. Navjot Singh

Design and Analysis of Algorithms

Graphs

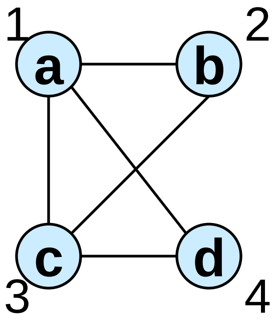
- *Graph* $G = (V, E)$
  - V = set of vertices
  - E = set of edges $\subseteq (V \times V)$
- Types of graphs
  - Undirected: edge $(u, v) = (v, u)$; for all v, $(v, v) \notin E$ (No self loops.)
  - Directed: (u, v) is edge from u to v, denoted as $u \to v$. Self loops are allowed.
  - Weighted: each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

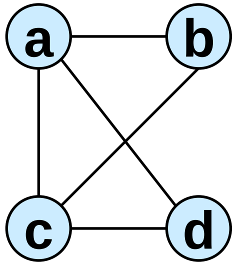
Representation of Graphs

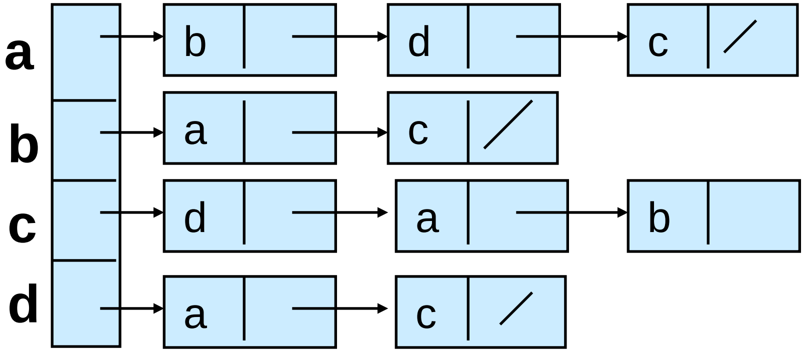
- Two standard ways.
  - Adjacency Matrix



	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 List

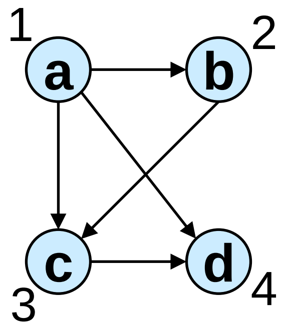




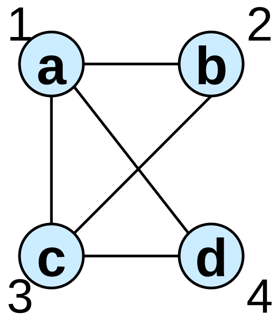
Adjacency Matrix

- $|V| \times |V|$ 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

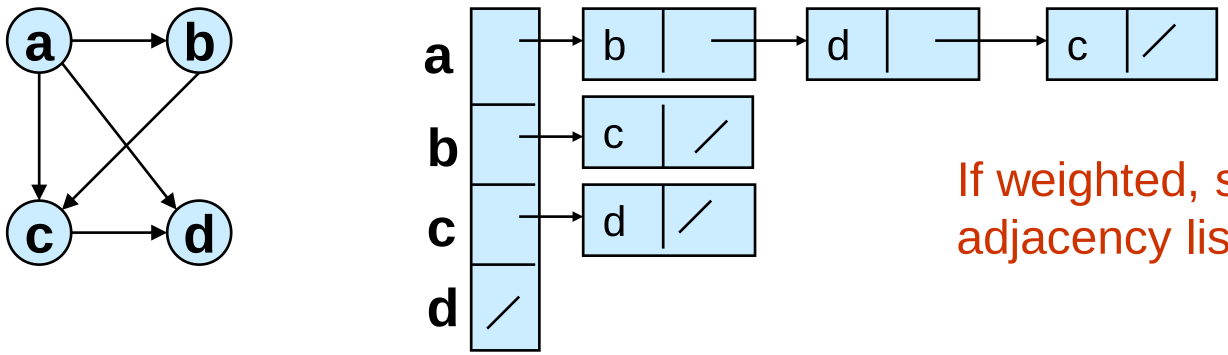
$A = A^{\mathrm{T}}$ for undirected graphs.

Space and Time

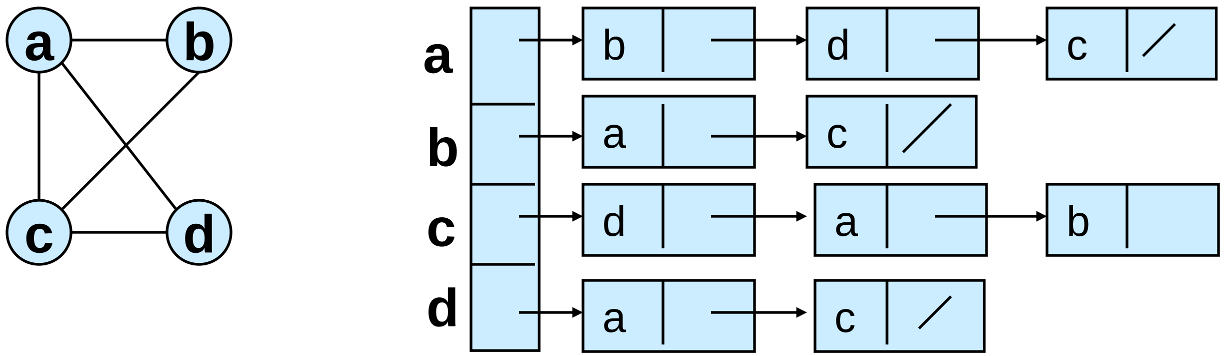
- **Space:** $\Theta(V^2)$.
  - Not memory efficient for large graphs.
- **Time:** to list all vertices adjacent to u: $\Theta(V)$.
- **Time:** to determine if $(u, v) \in E$: $\Theta(1)$.
- Can store weights instead of bits for weighted graph.

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. 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.
  - Can be modified to support many graph variants.
- 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.
    - $\Theta(V)$ in the worst case.

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.

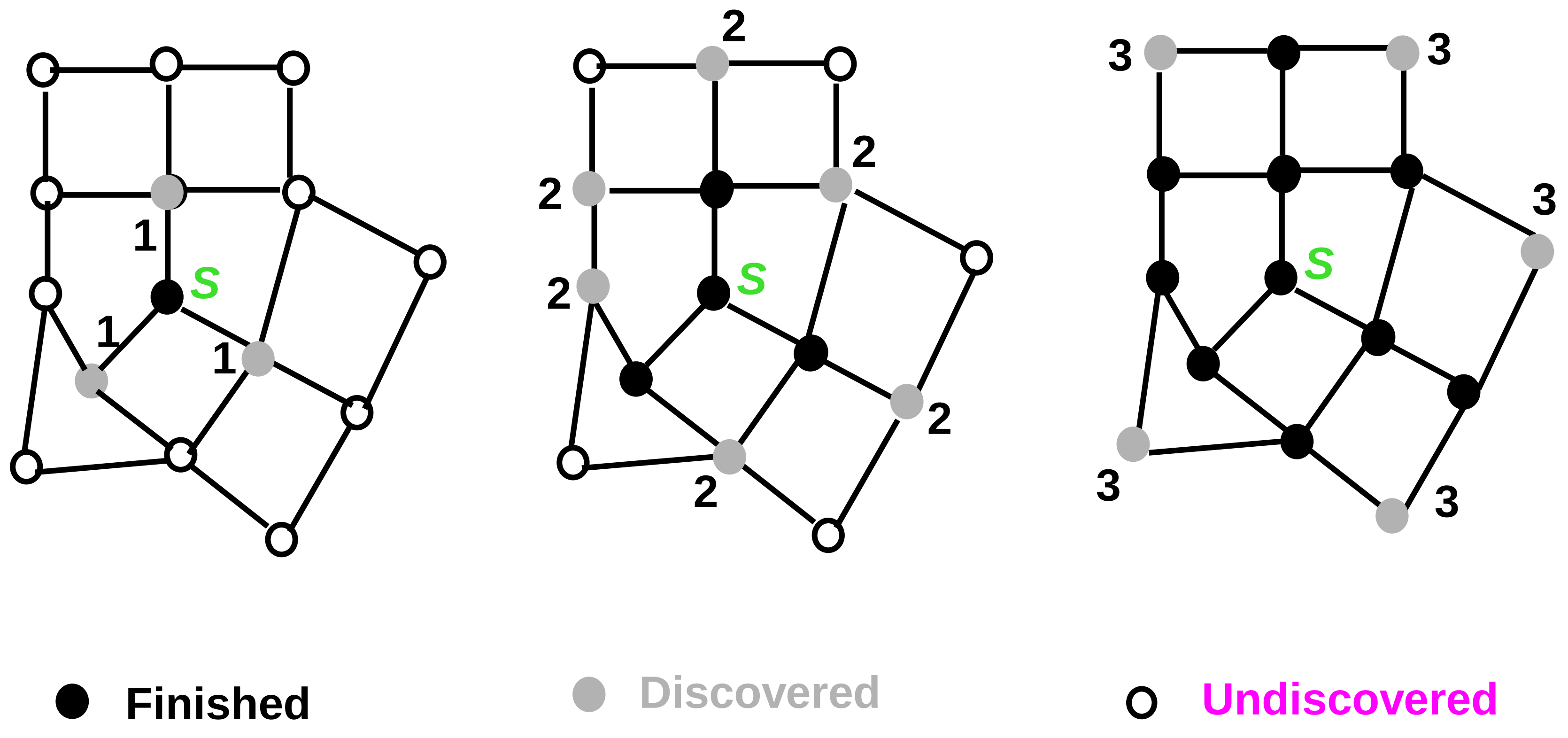
Definitions:
Path between vertices u and v: Sequence of vertices $(v_1, v_2, \ldots, v_k)$ such that $u=v_1$ and $v=v_k$, and $(v_i,v_{i+1}) \in E$, for all $1 \le i \le k$-1.
Length of the path: Number of edges in the path.
Path is simple if no vertex is repeated.

Error!

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 for Shortest Paths



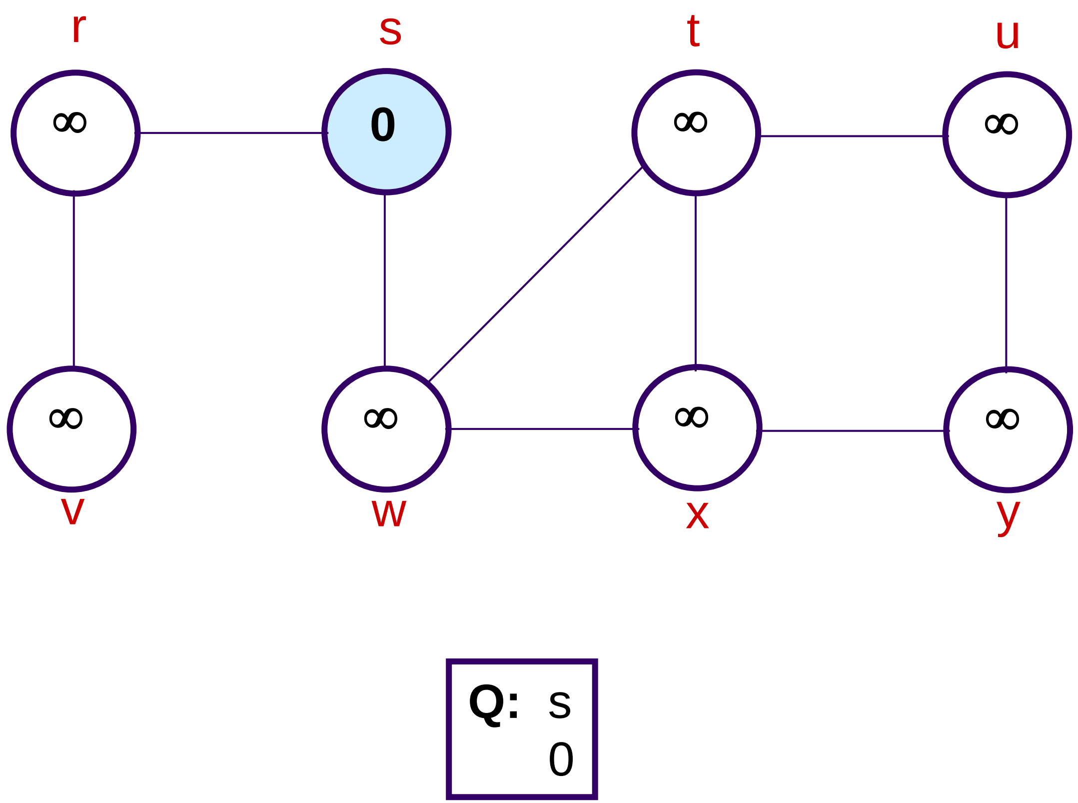
BFS(G,s)

```
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
```

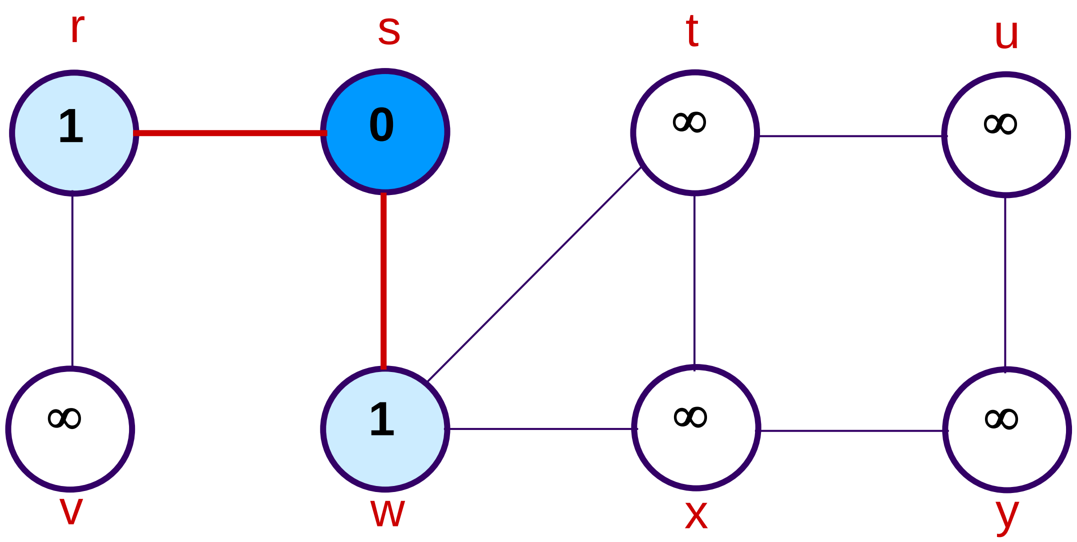
white: undiscovered
gray: discovered
black: finished

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

Example (BFS)

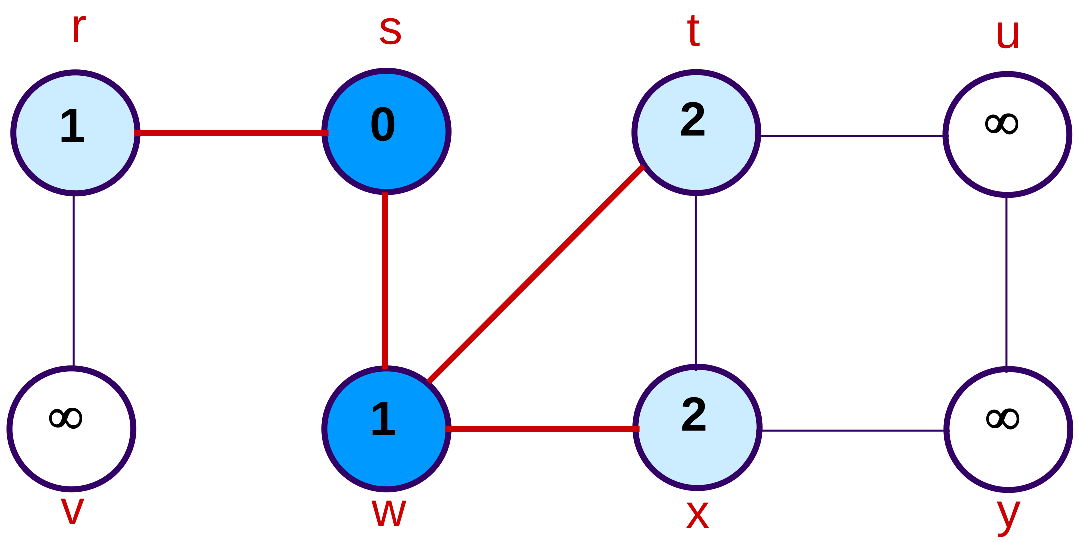


Example (BFS)



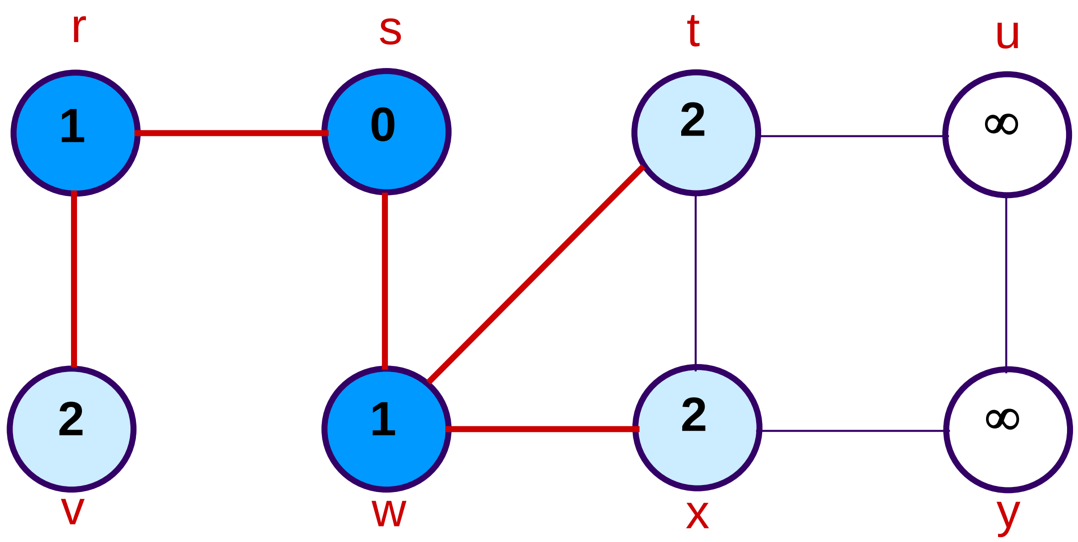
Q:	w	r
	1	1

Example (BFS)



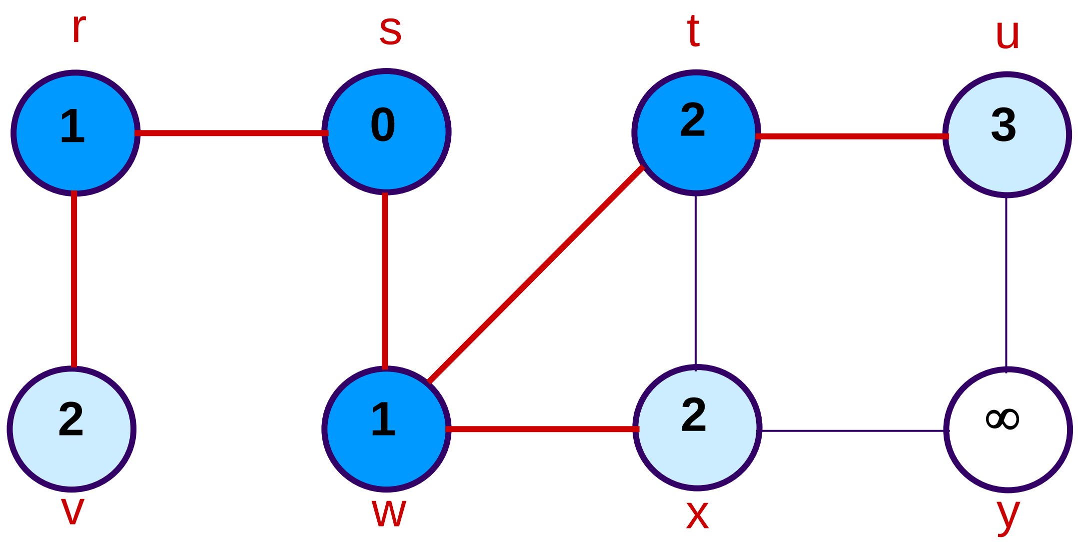
Q:	r	t	x
	1	2	2

Example (BFS)



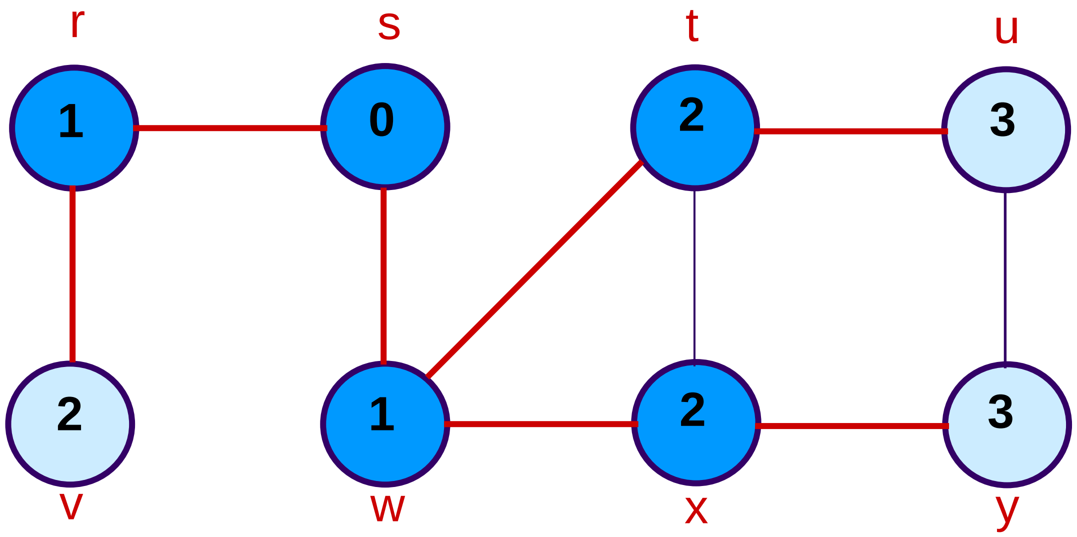
Q:	t	x	v
	2	2	2

Example (BFS)



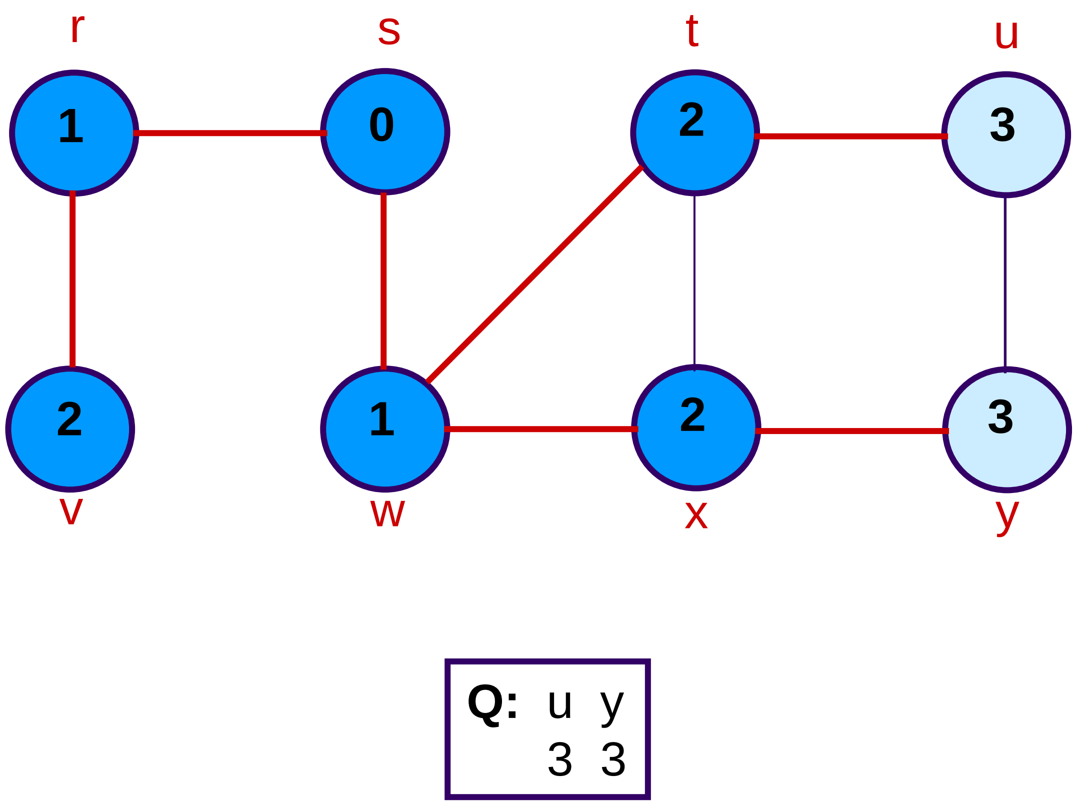
Q: x v u
2 2 3

Example (BFS)

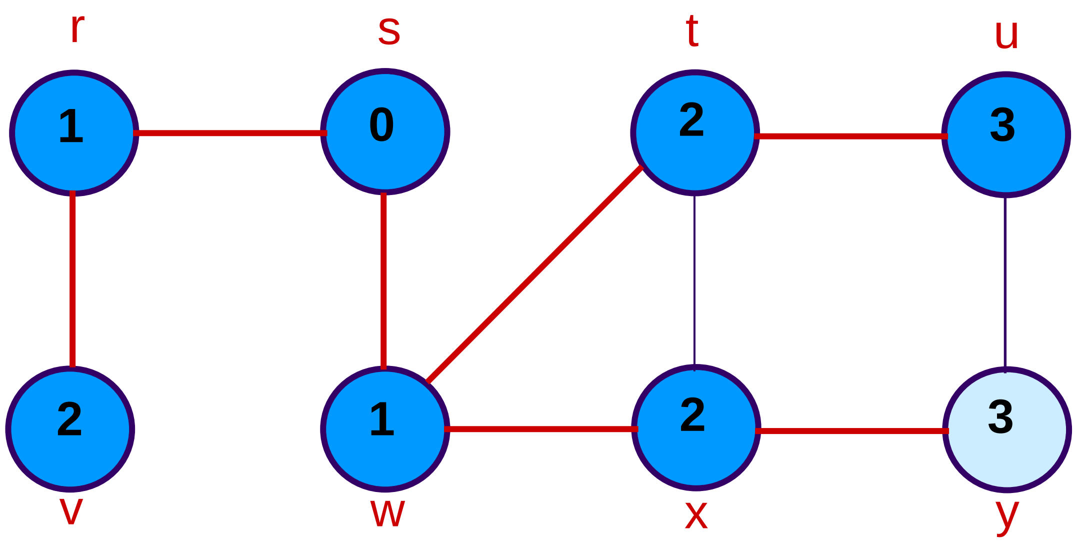


Q:	v	u	y
	2	3	3

Example (BFS)

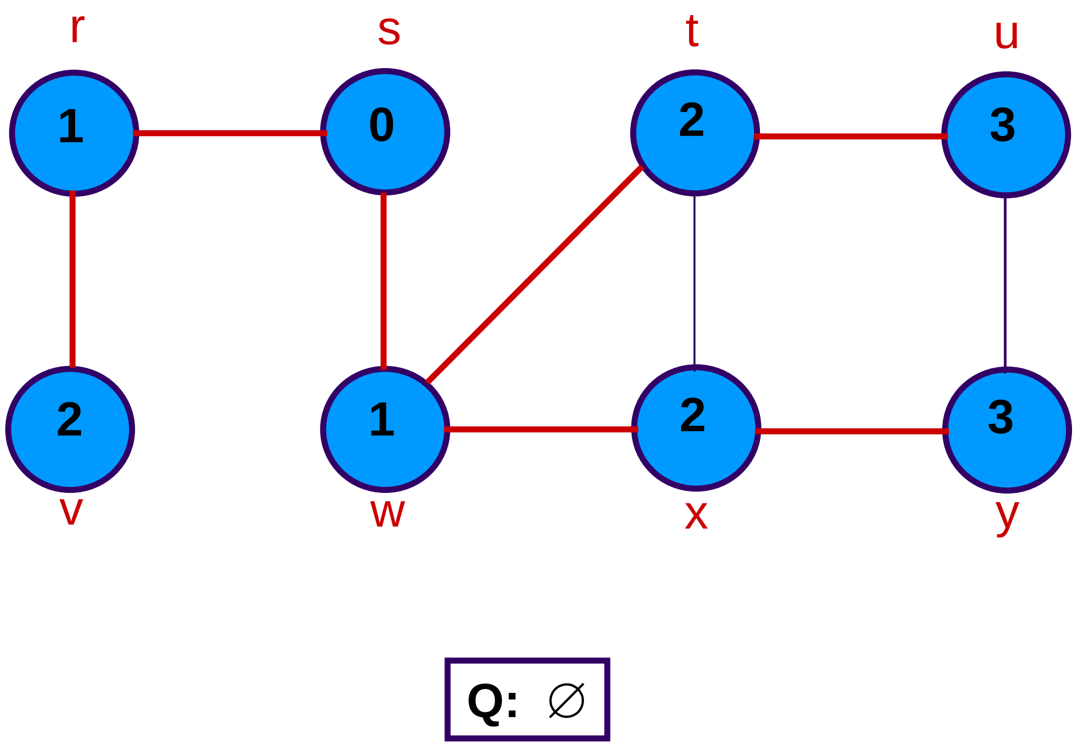


Example (BFS)

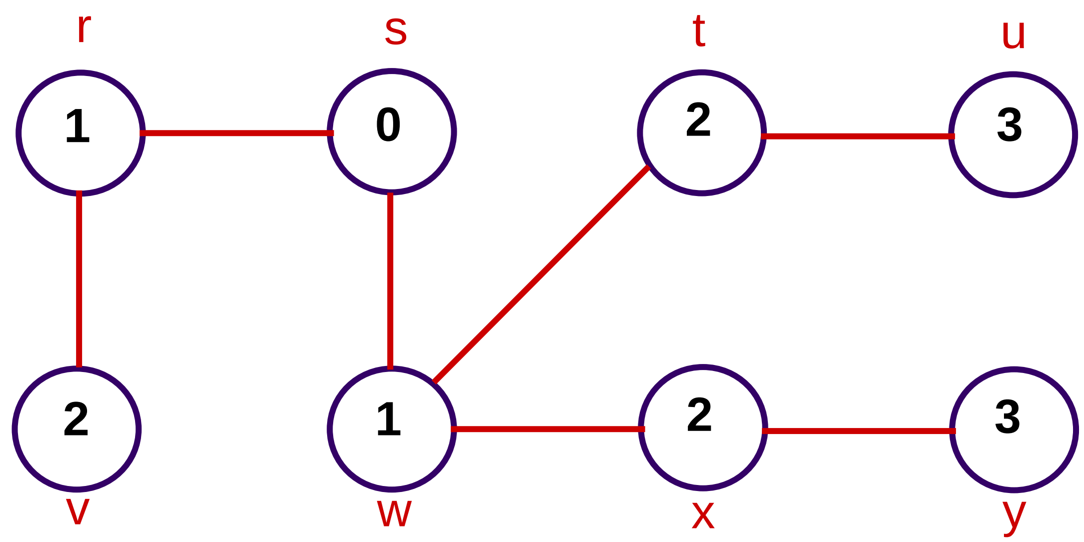


Q: y
3

Example (BFS)



Example (BFS)



Breadth-First 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.

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** if:
  - 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.

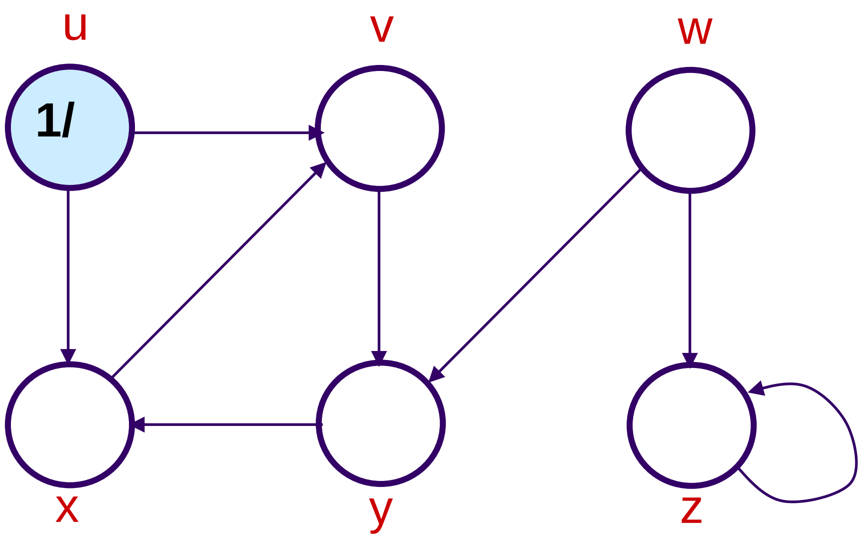
Pseudo-code

```
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)
```

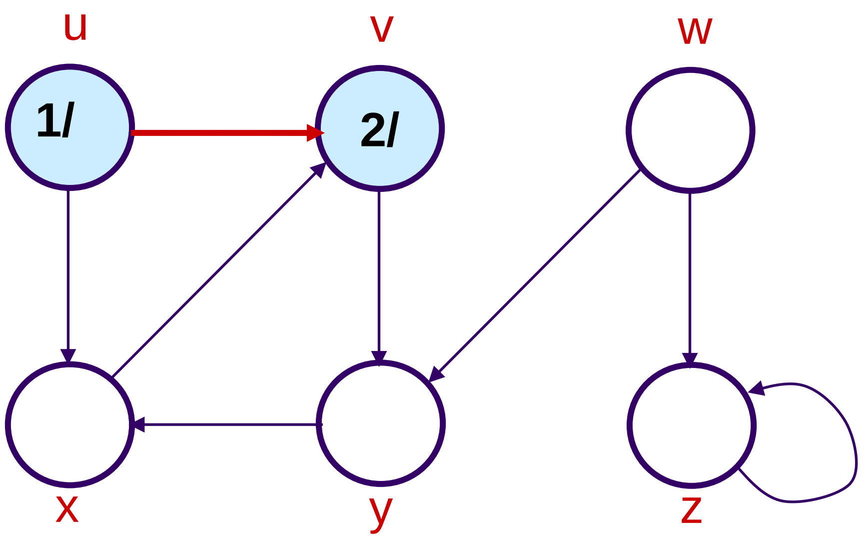
Uses a global timestamp ***time***.

```
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
```

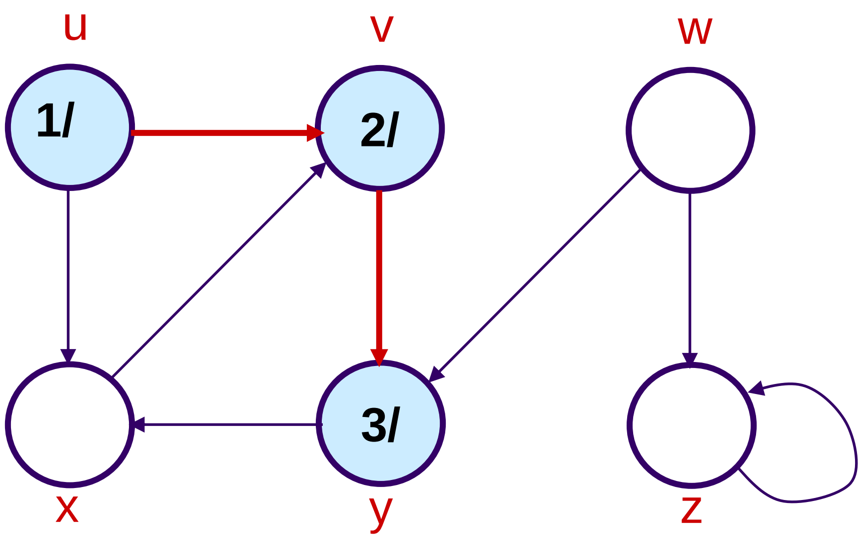
Example (DFS)



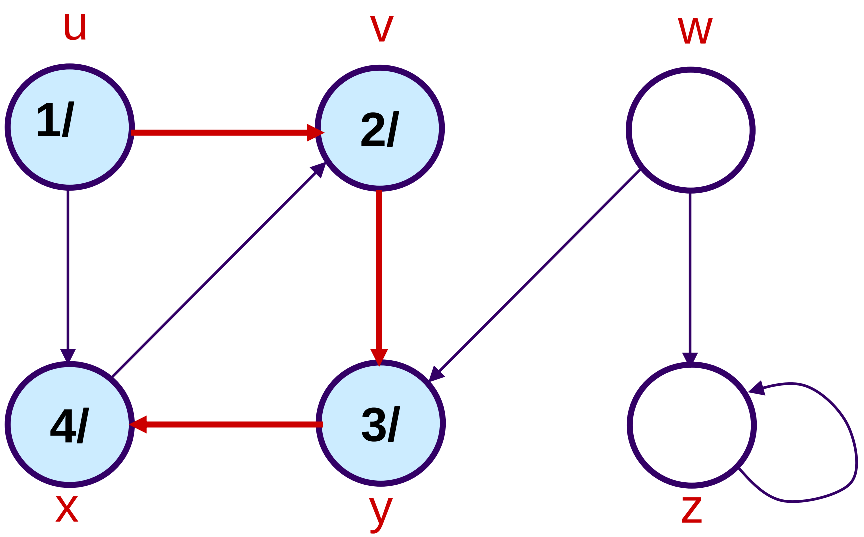
Example (DFS)



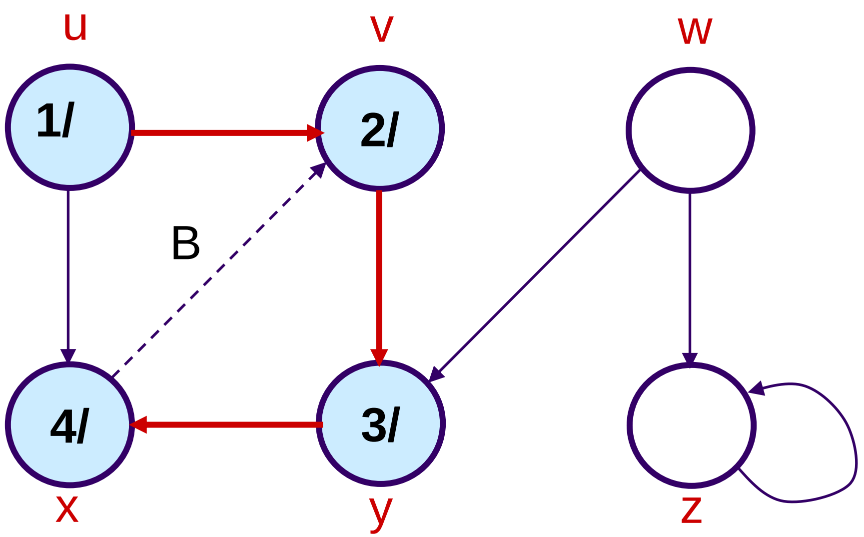
Example (DFS)



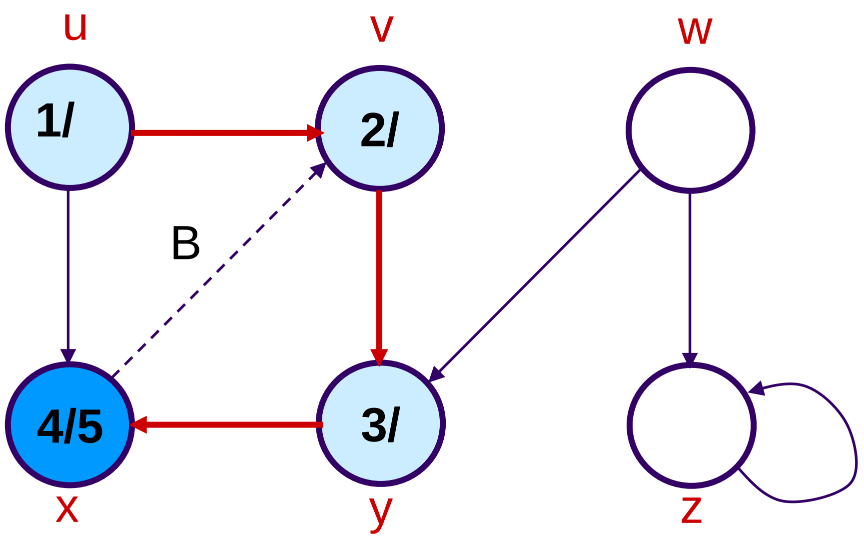
Example (DFS)



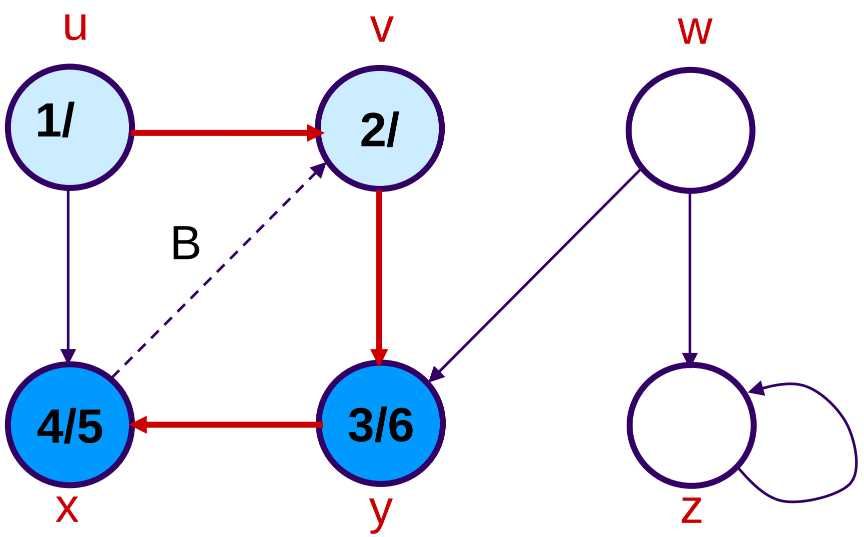
Example (DFS)



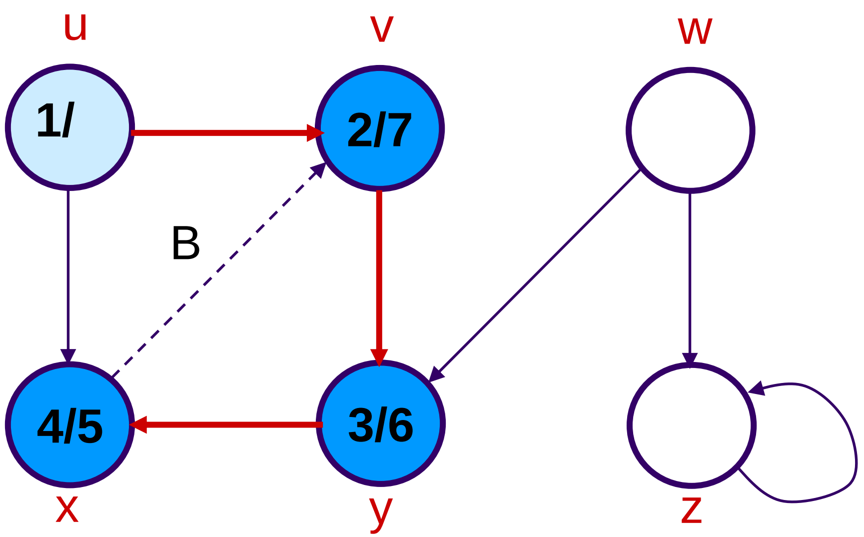
Example (DFS)



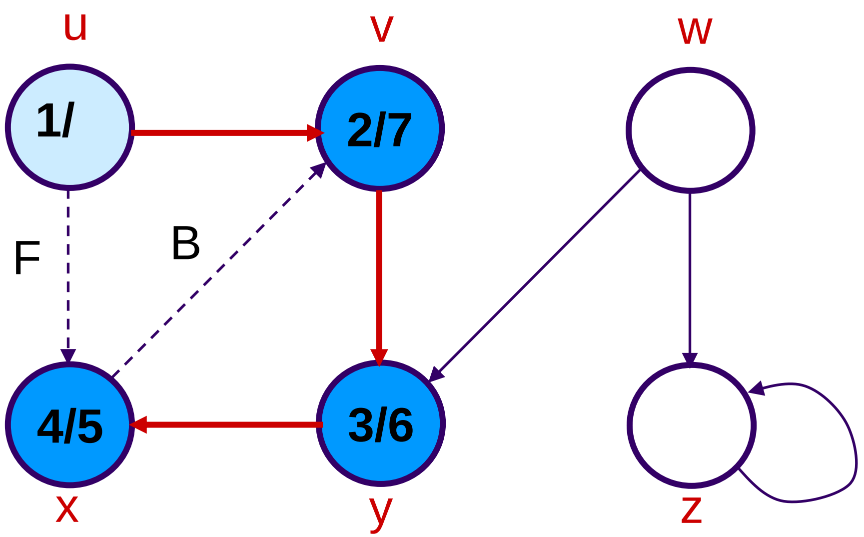
Example (DFS)



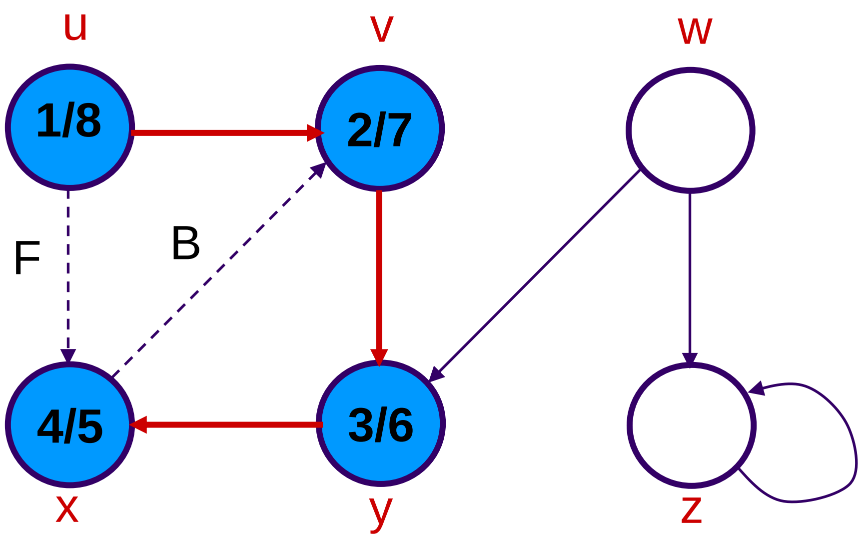
Example (DFS)



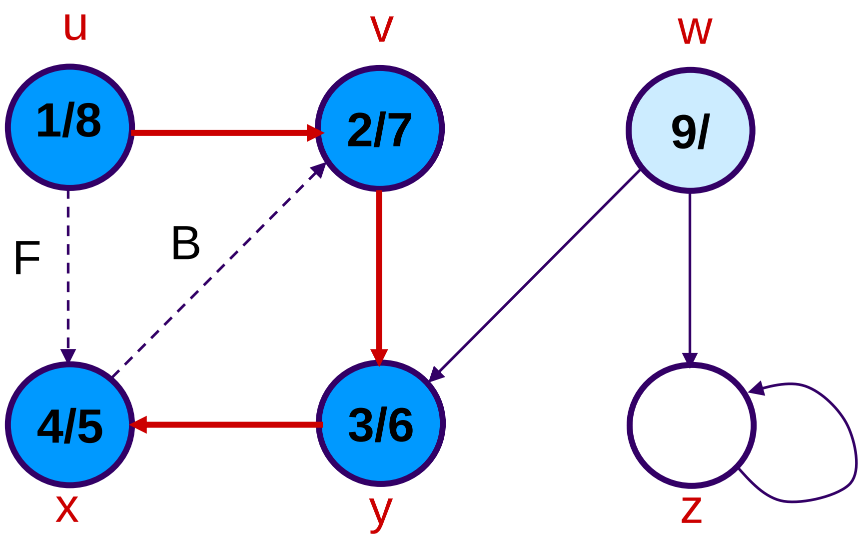
Example (DFS)



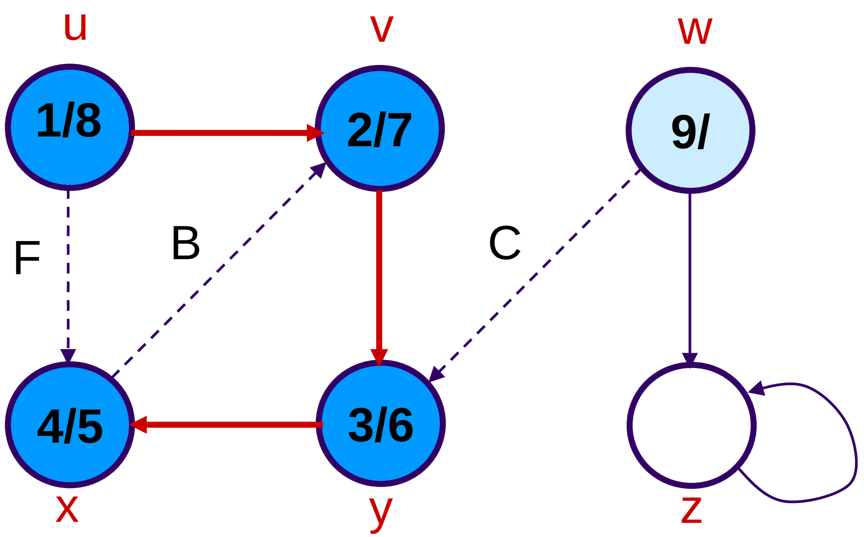
Example (DFS)



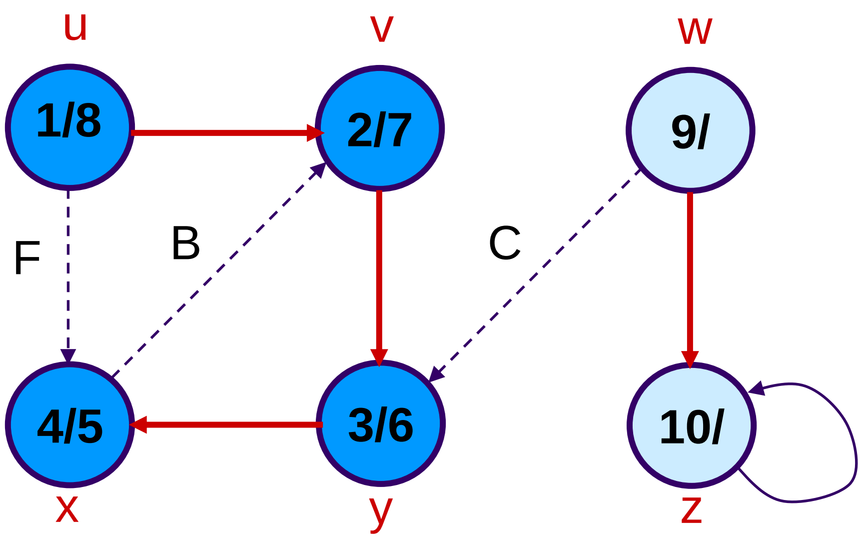
Example (DFS)



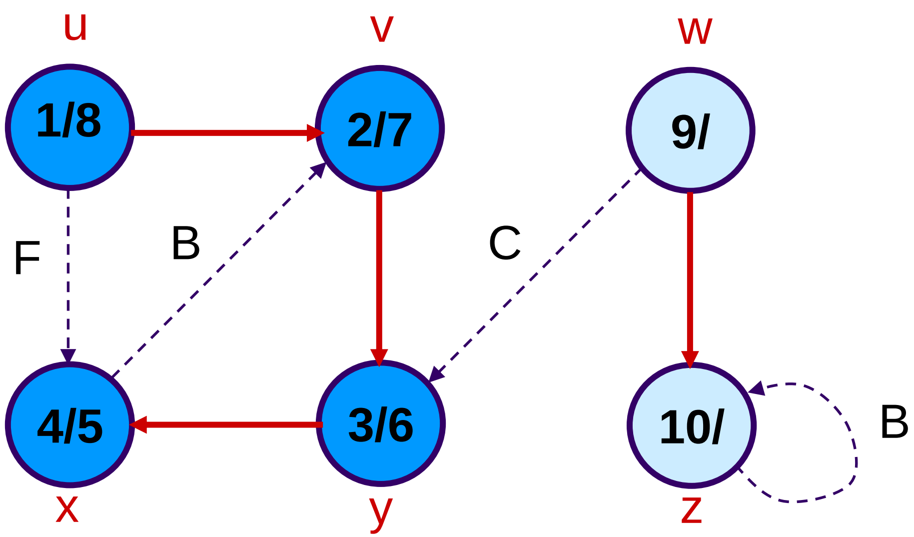
Example (DFS)



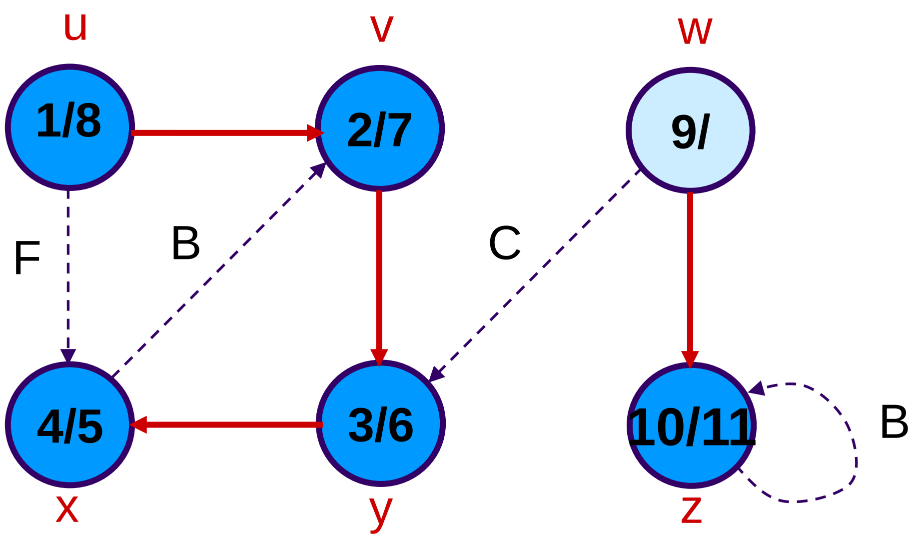
Example (DFS)



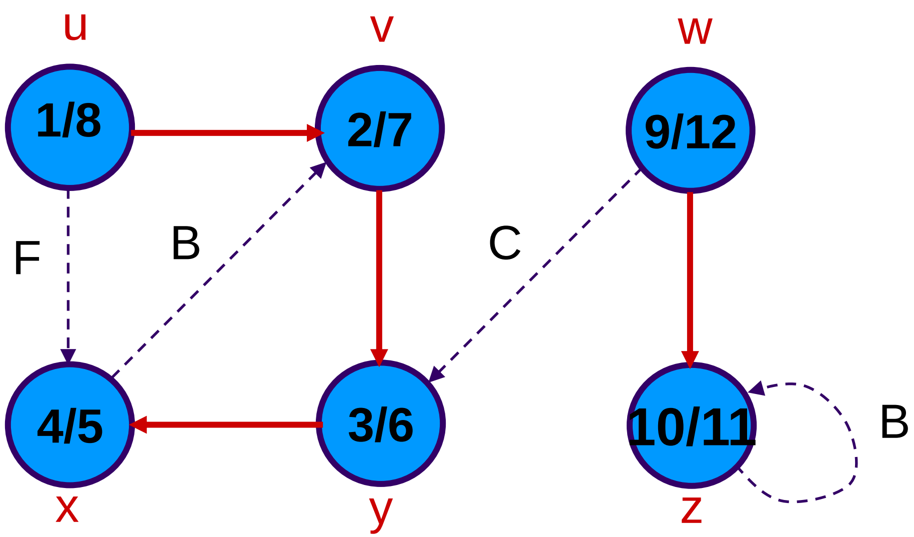
Example (DFS)



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 3-6 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 $\Theta(V+E)$.

Parenthesis Theorem

Theorem

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

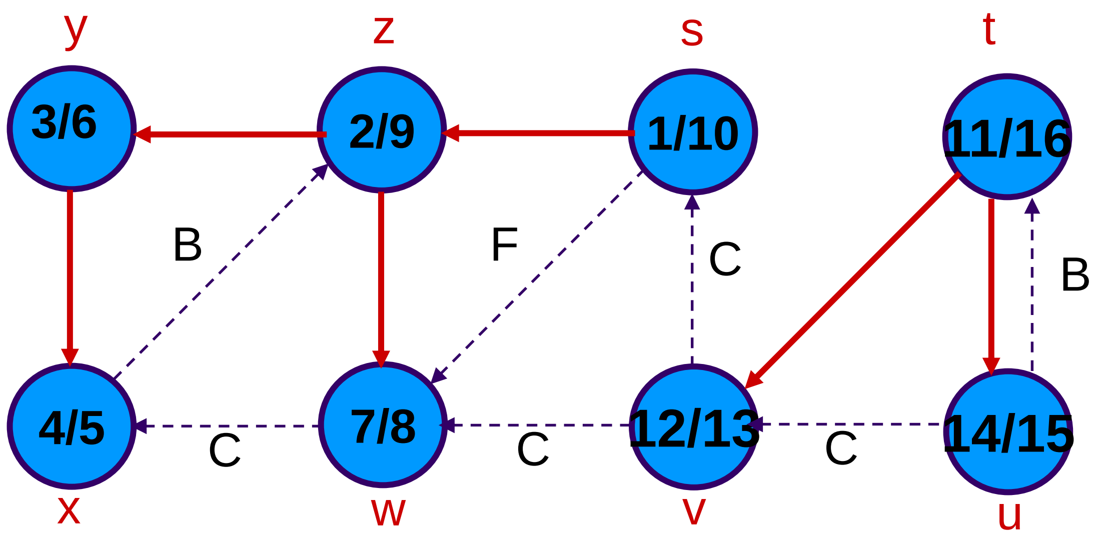
1. $d[u] < f[u] < d[v] < f[v]$ or $d[v] < f[v] < d[u] < f[u]$ and neither u nor v is a descendant of the other.
2. $d[u] < d[v] < f[v] < f[u]$ and v is a descendant of u.
3. $d[v] < d[u] < f[u] < f[v]$ and u is a descendant of v.

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

Example (Parenthesis Theorem)



(s (z (y (x x) y) (w w) z) s) (t (v v) (u u) t)

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

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.

Identification of Edges

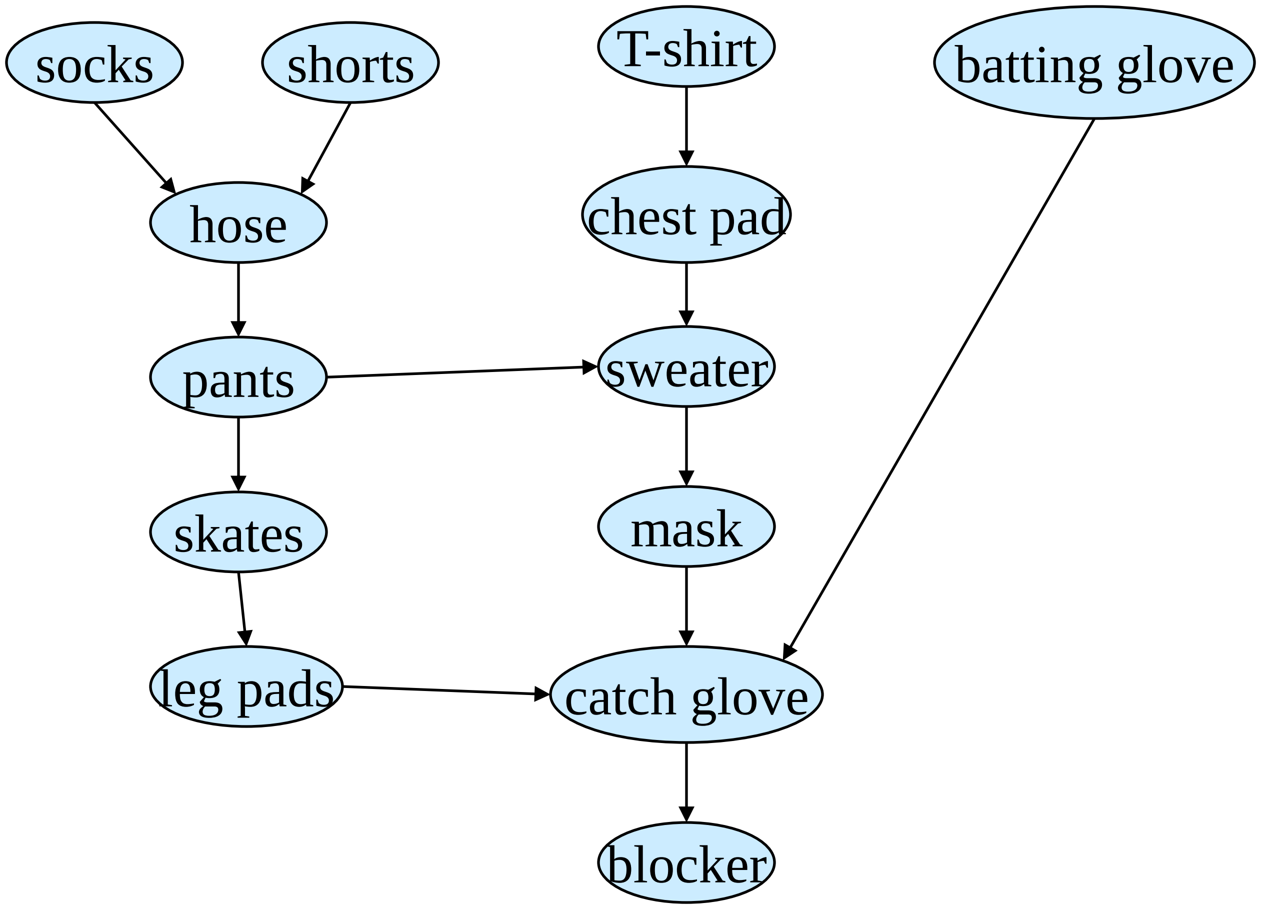
- Edge type for edge (u, v) can be identified when it is first explored by DFS.
- Identification is based on the **color of *v***.
  - White – tree edge.
  - Gray – back edge.
  - Black – forward or cross edge.

Directed Acyclic Graph

- DAG – 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.

Example

DAG of dependencies for putting on goalie equipment.

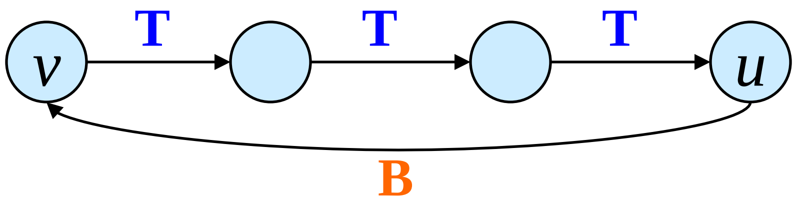


Characterizing a DAG

Lemma
A directed graph G is acyclic iff 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 $v \rightsquigarrow u$, so $v \rightsquigarrow u \rightsquigarrow v$ is a cycle.

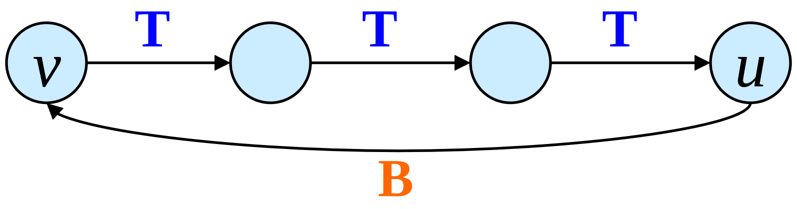


Characterizing a DAG

Lemma
A directed graph G is acyclic iff a DFS of G yields no back edges.

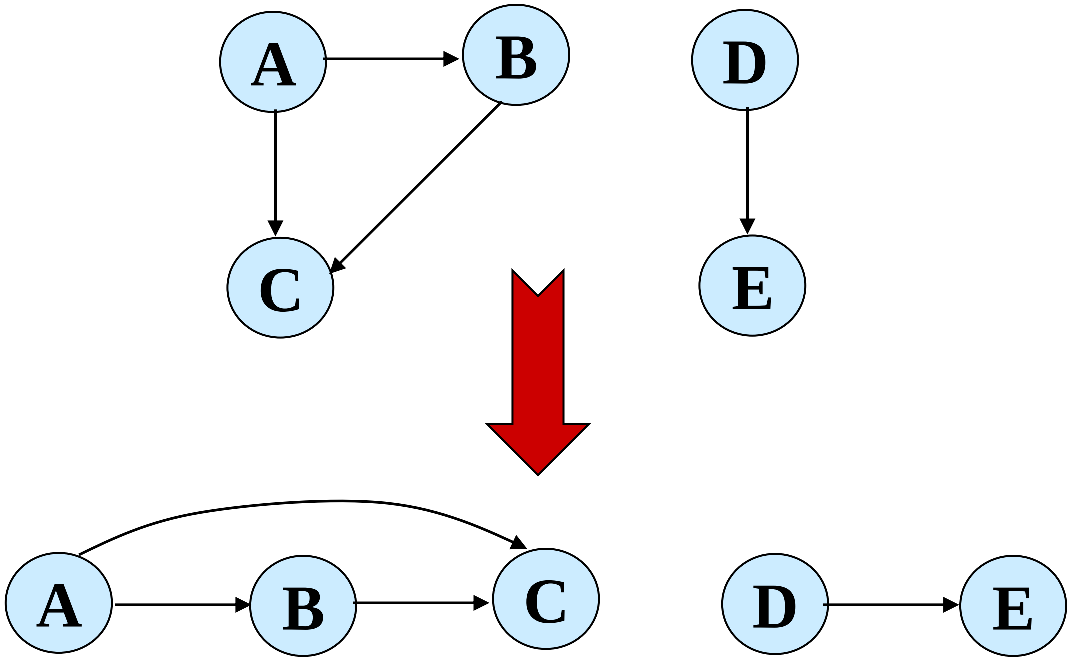
Proof (Contd.):

- $\Leftarrow$: Show that a cycle implies a back edge.
  - c : cycle in G, v : first vertex discovered in c, (u, v) : preceding edge in c.
  - At time $d[v]$, vertices of c form a white path $v \rightsquigarrow u$. Why?
  - By white-path theorem, u is a descendent of v in depth-first forest.
  - Therefore, (u, v) is a back edge.



Topological Sort

Want to "sort" a directed acyclic graph (DAG).



Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

Topological Sort

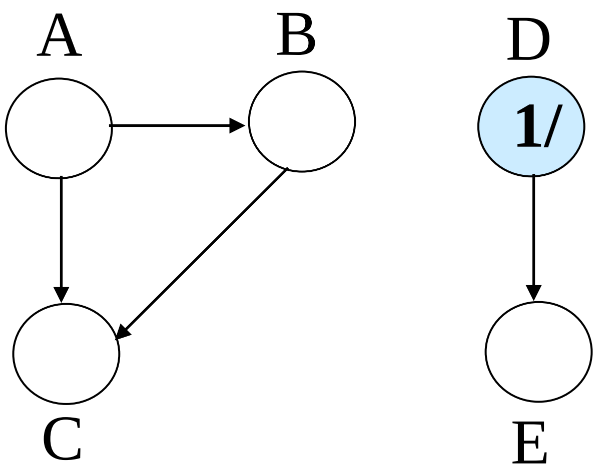
- Performed on a DAG.
- Linear ordering of the vertices of G such that if $(u, v) \in E$, then u appears somewhere before v.

Topological-Sort (G)

1. call DFS(G) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. **return** the linked list of vertices

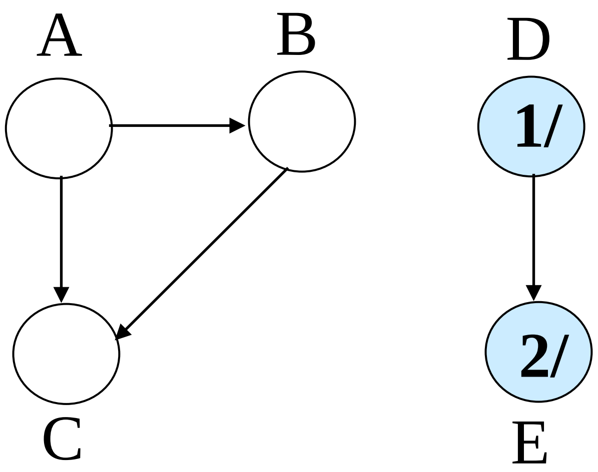
Time: $\Theta(V + E)$.

Example



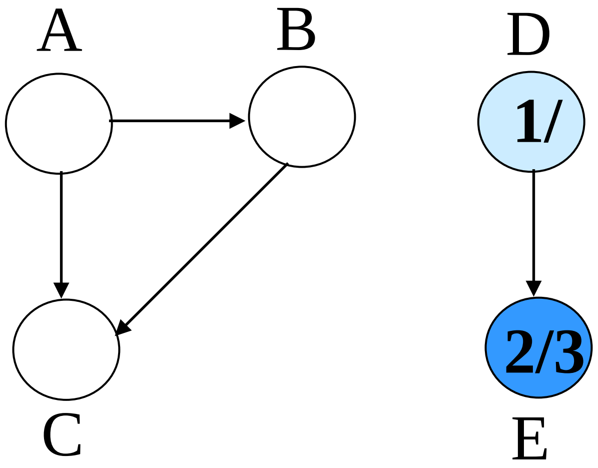
Linked List:

Example

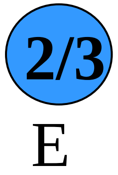


Linked List:

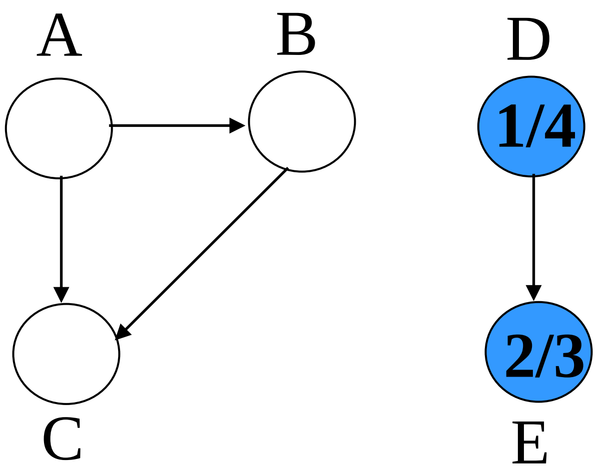
Example



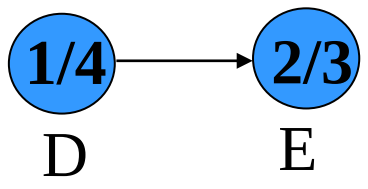
Linked List:



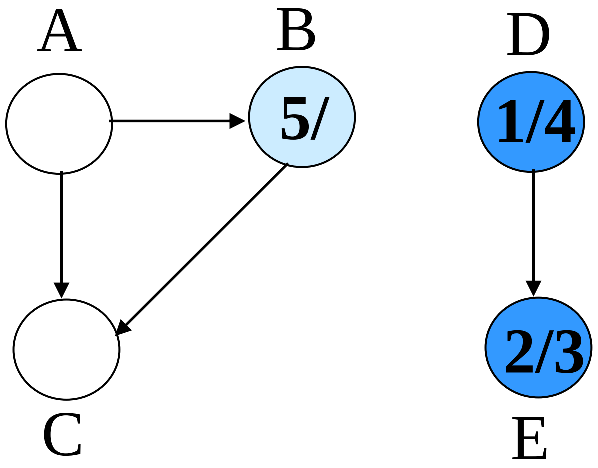
Example



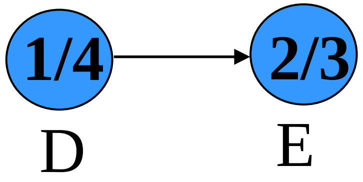
Linked List:



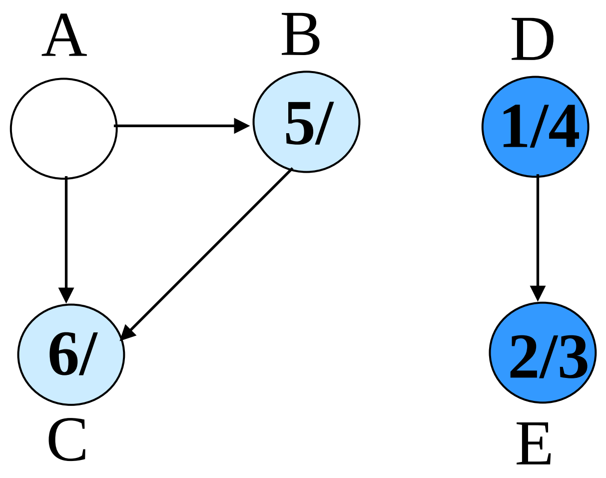
Example



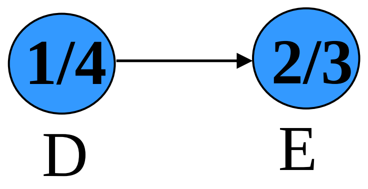
Linked List:



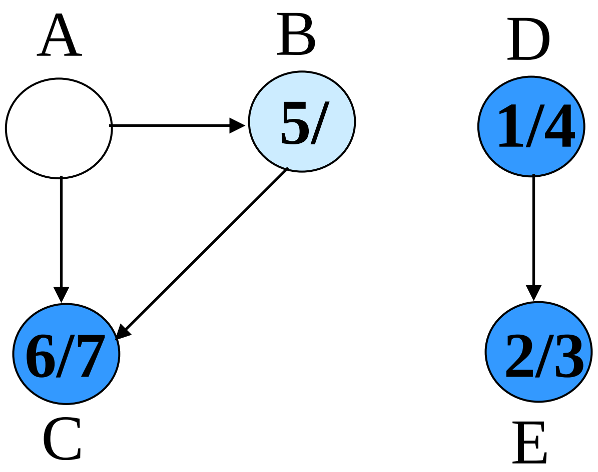
Example



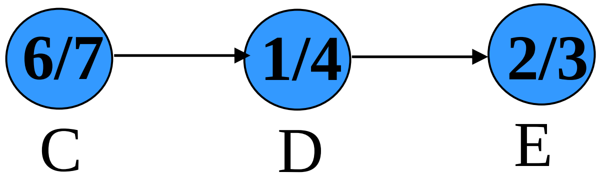
Linked List:



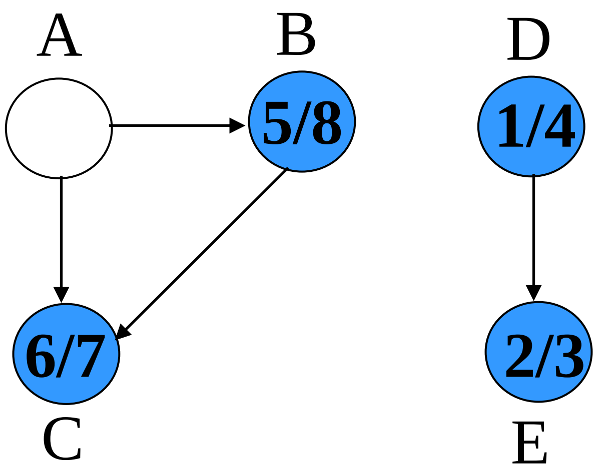
Example

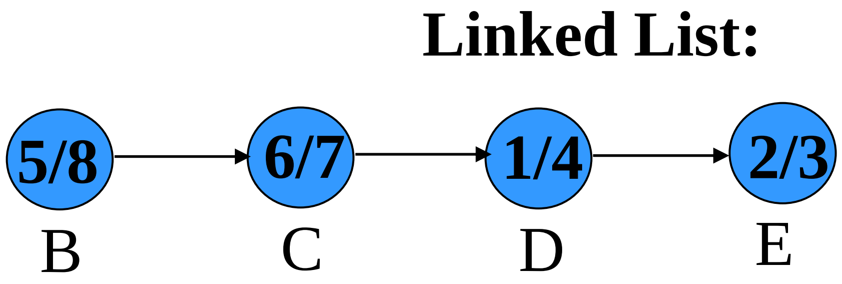


Linked List:

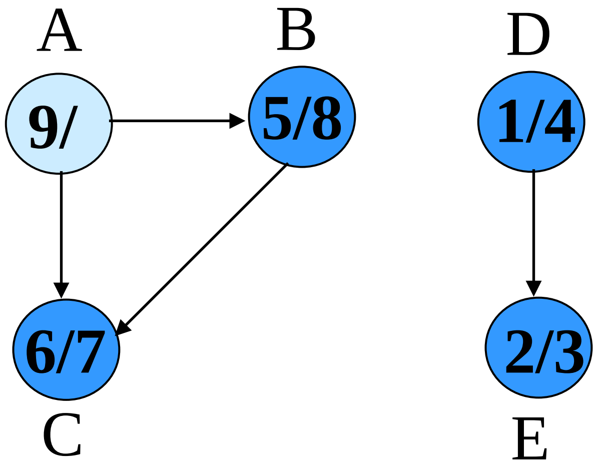


Example

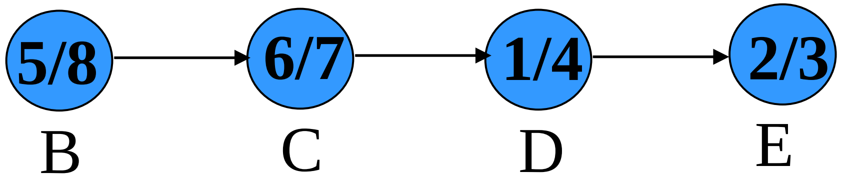




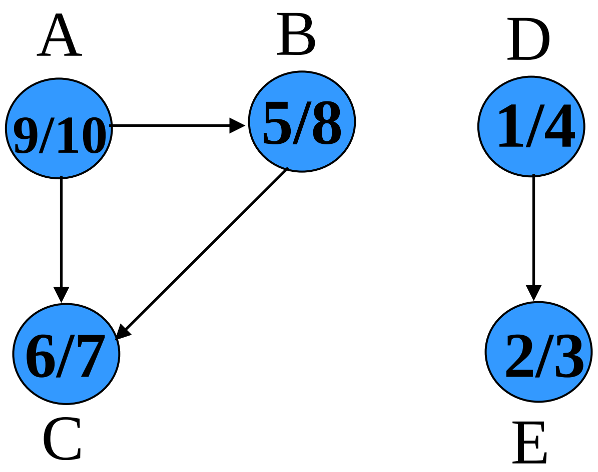
Example



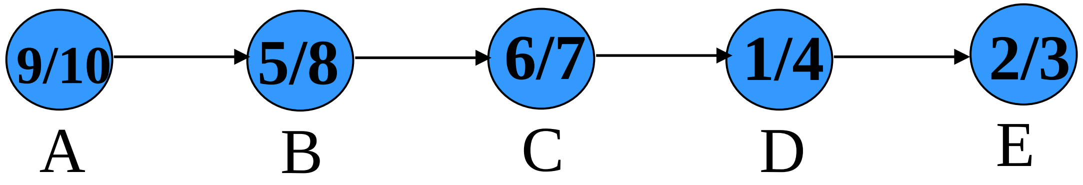
Linked List:



Example



Linked List:

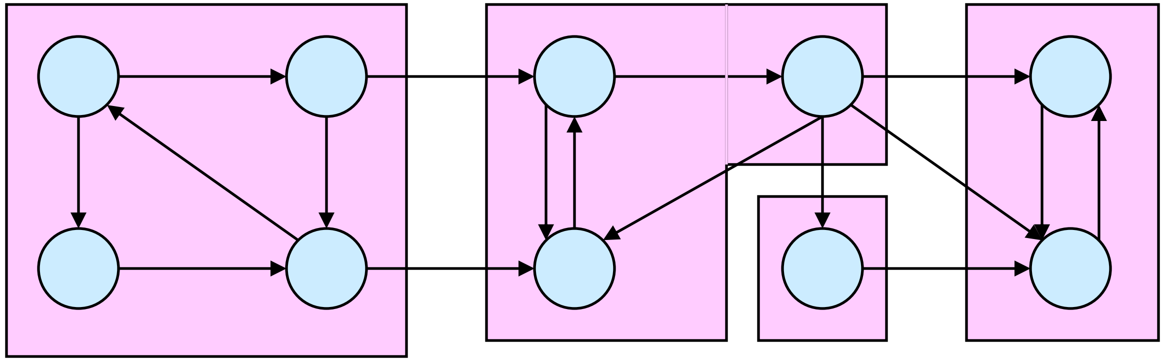


Correctness Proof

- Just need to 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 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, $f[v] < f[u]$.

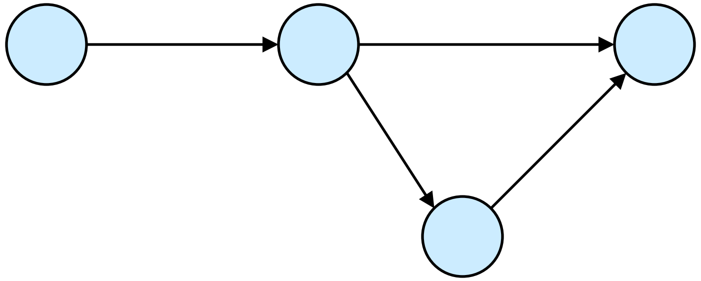
Strongly Connected Components

- 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 22.13

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 $u \leadsto u'$ in G. Then there cannot also be a path $v' \leadsto v$ in G.

Proof:

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

Transpose of a Directed Graph

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

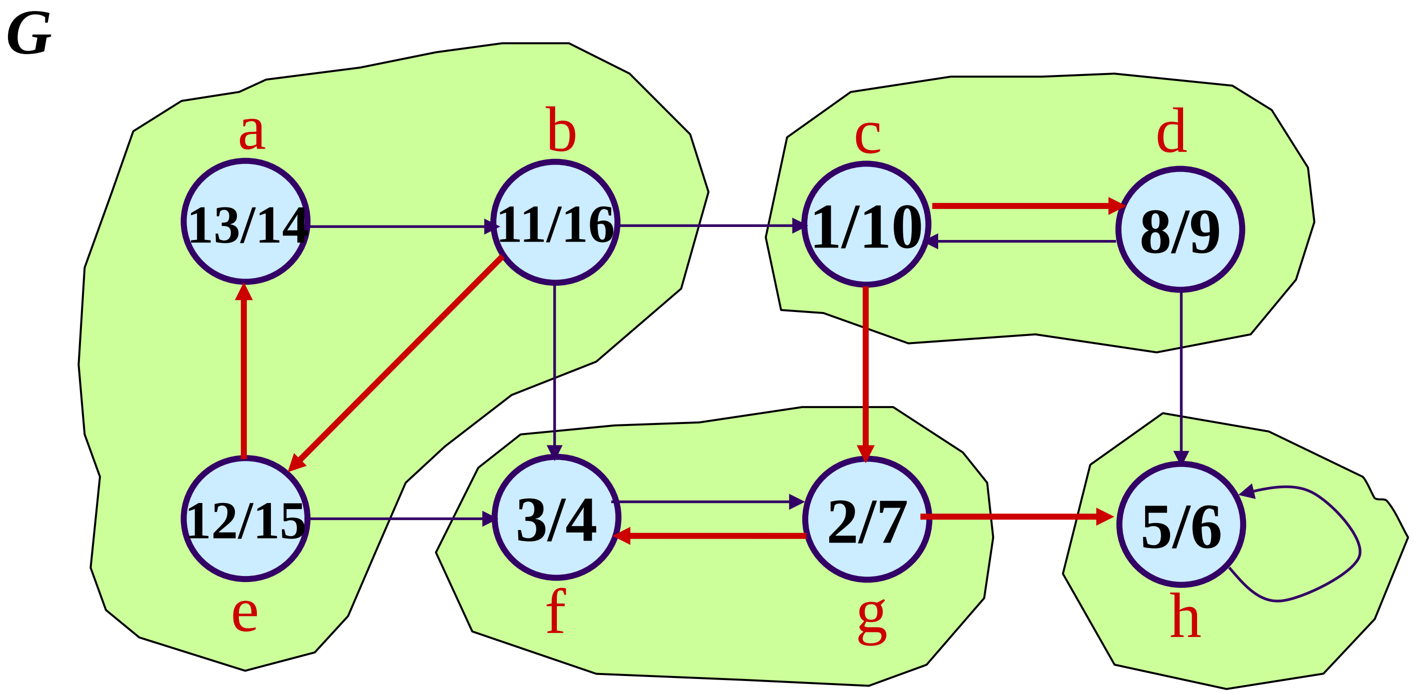
Algorithm to determine SCCs

SCC(G)

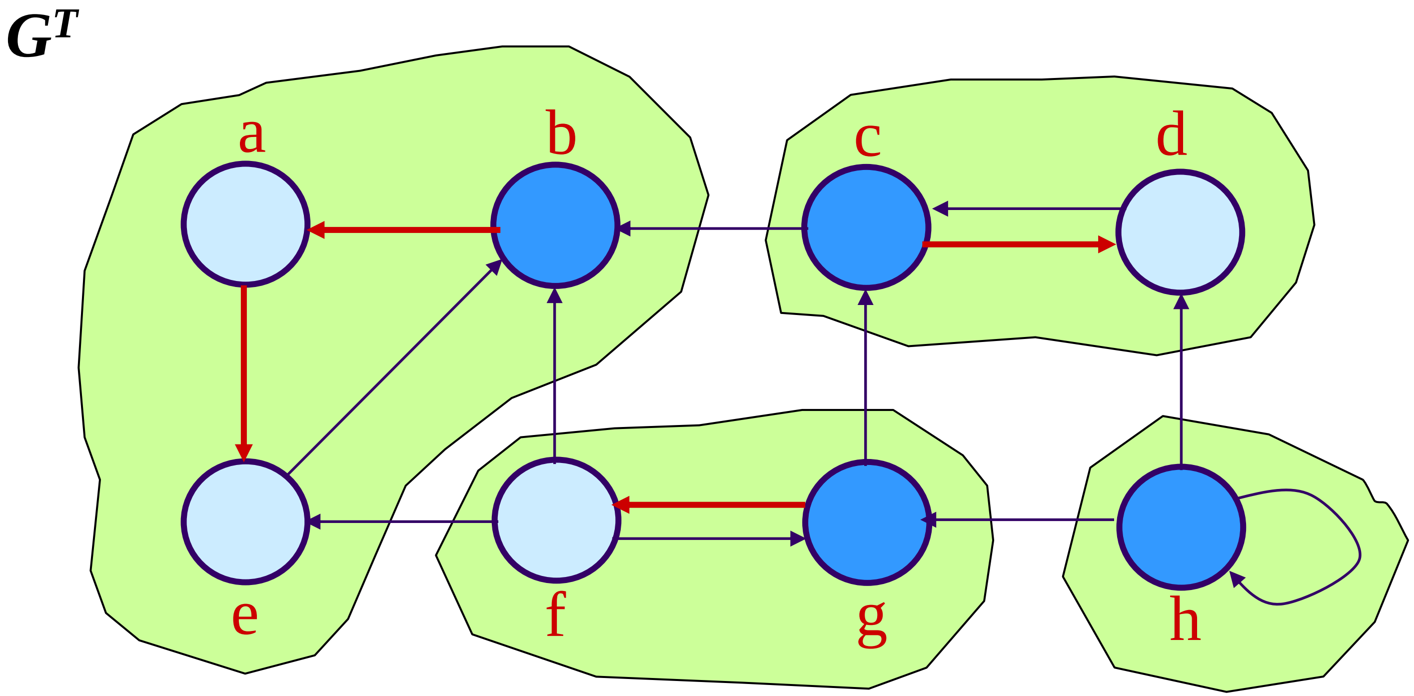
1. call DFS(G) to compute finishing times $f[u]$ for all u
2. compute G^T
3. call DFS(G^T), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

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

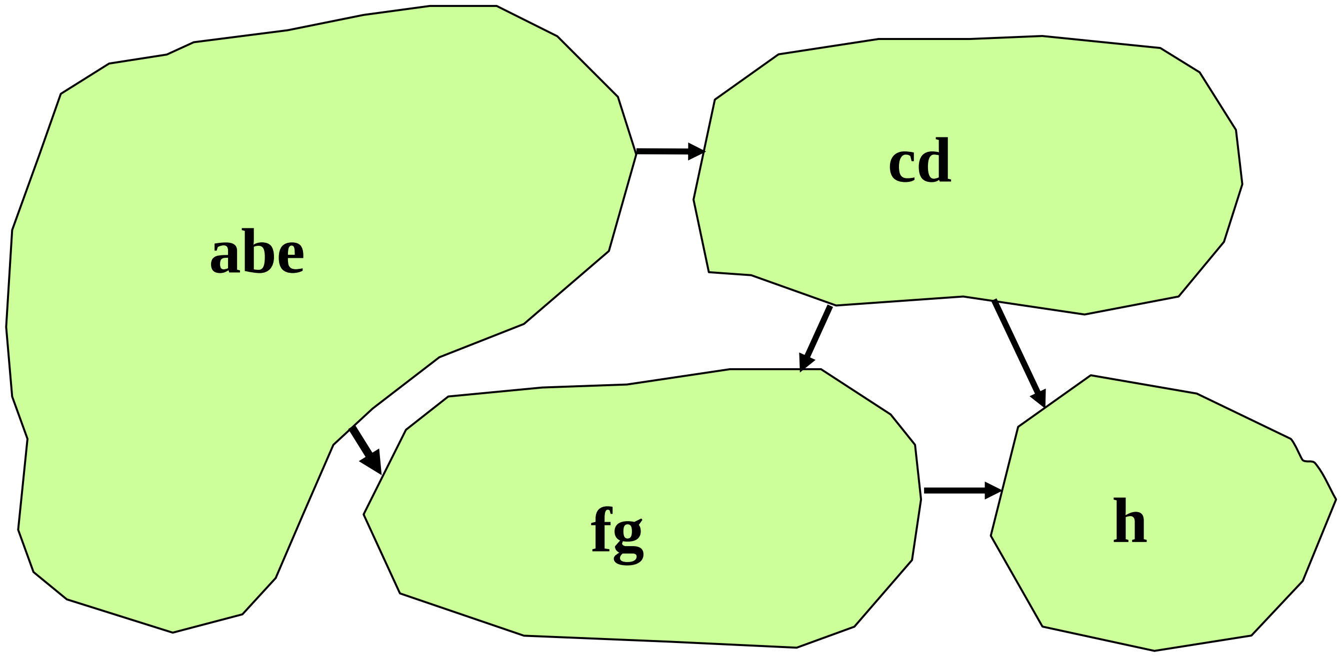
Example



Example



Example



How does it 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 topologically sorted order.
  - Because we are running DFS on G^T, we will not be visiting any v from a u, where v and u are in different components.
- **Notation:**
  - $d[u]$ and $f[u]$ always refer to *first* DFS.
  - Extend notation for d and f to sets of vertices $U \subseteq V$:
  - $d(U) = \min_{u \in U}\{d[u]\}$ (earliest discovery time)
  - $f(U) = \max_{u \in U}\{f[u]\}$ (latest finishing time)

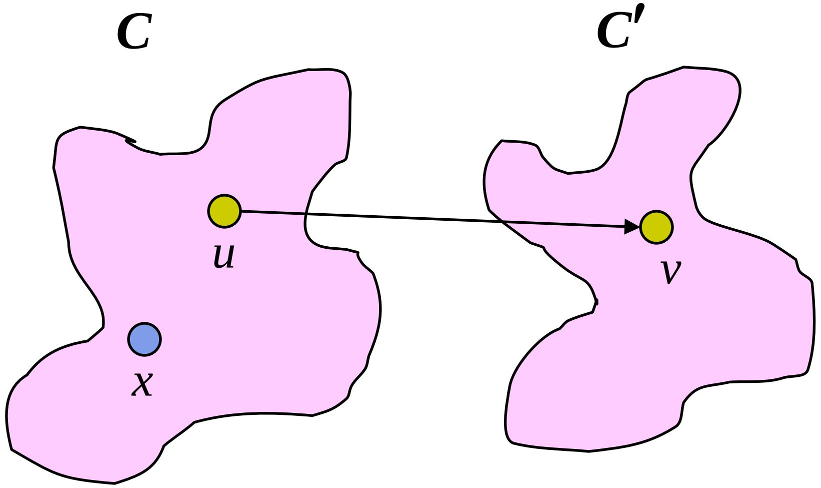
SCCs and DFS finishing times

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:

- 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, $f[x] = f(C) > f(C')$.



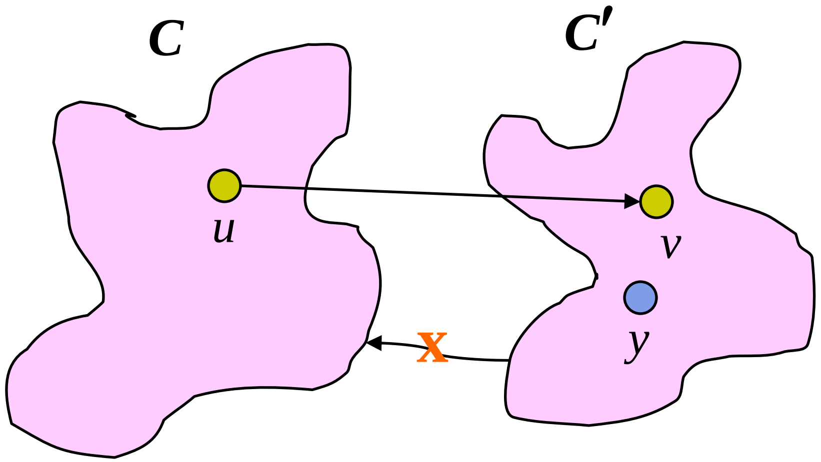
SCCs and DFS finishing times

Lemma

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



SCCs and DFS finishing times

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, $f(C') > f(C)$, by Lemma.

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.

Acknowledgements

- Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., Introduction to algorithms. MIT press, 2009
- Dr. David Kauchak, Pomona College
- Prof. David Plaisted, The University of North Carolina at Chapel Hill