Homework 9 Due Tuesday Dec 6

- CLRS 19.2-4 (correctness of heap union)
- CLRS 22.3-4 (depth-first search)
- CLRS 22-1 (breadth-first search)

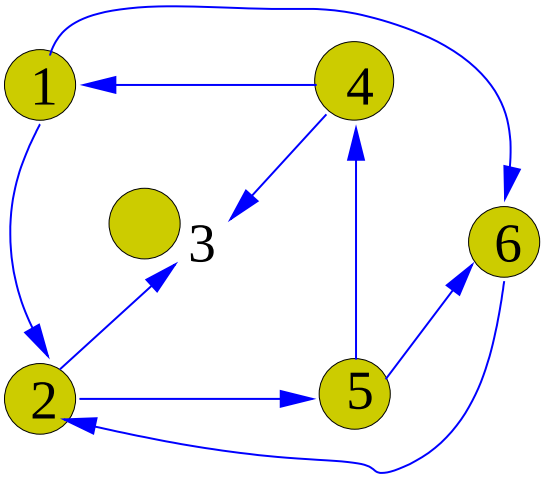
Chapter 22: Elementary Graph Algorithms

- Graph representation
- Search strategies
- Shortest path
- Topological sort
- Strongly connected components

Representations

1. **Adjacency-List Representation** A list of adjacent nodes per node. Encoding size $= \Theta(E + V)$. Suitable for **sparse graphs**.
2. **Adjacency-Matrix Representation** The $|V| \times |V|$ matrix that represents connection between nodes. Encoding size $= \Theta(V^2)$. Suitable for **dense graphs**.

Adjacency-List Representation



$1: \ [2,6]$

$2: \ [3,5]$

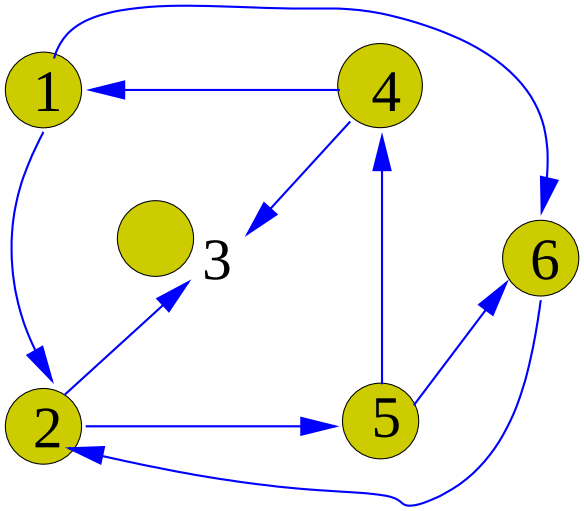
$3: \ []$

$4: \ [1,3]$

$5: \ [4,6]$

$6: \ [2]$

Adjacency-Matrix Representation



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

Traversal of Nodes

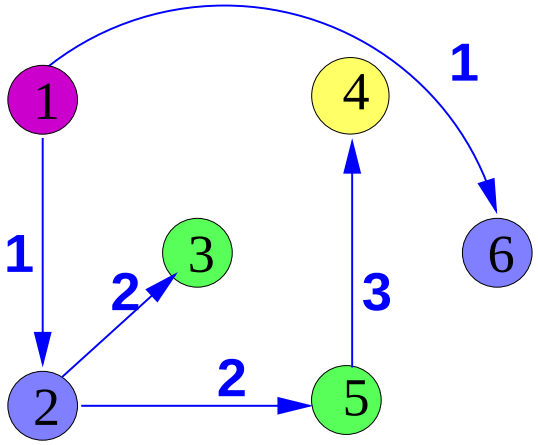
The problem of visiting all the nodes of a given graph G starting from a specific node s.

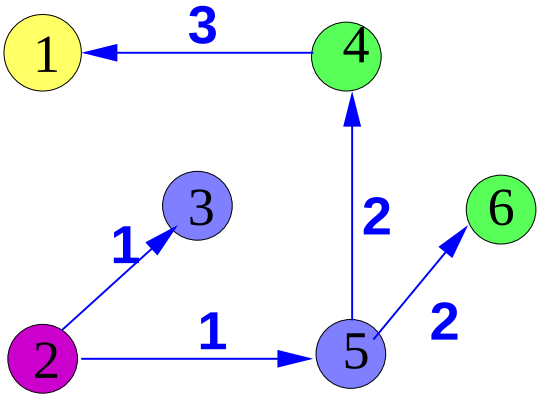
1. **Breadth-First Search** Mark all the unmarked adjacent nodes. Then recursively visit each of the adjacent nodes.
2. **Depth-First Search** If there are unmarked adjacent nodes visit one of them.

Connectivity in Undirected Graphs Nodes u and v are **connected** if there is a path between them. A graph G is **connected** if every pair of nodes is connected.

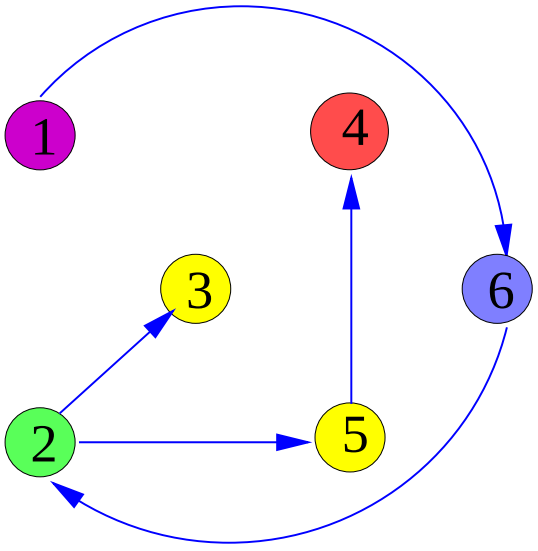
So, when search is finished check whether any node is yet to be visited. If so, start the search from any such one.

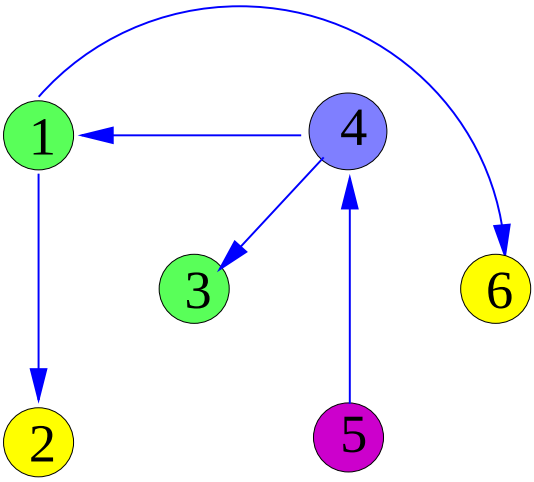
Breadth-First Search





Depth-First Search





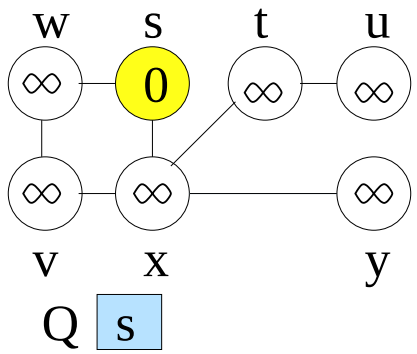
Computing the Minimum Distance from s with BFS

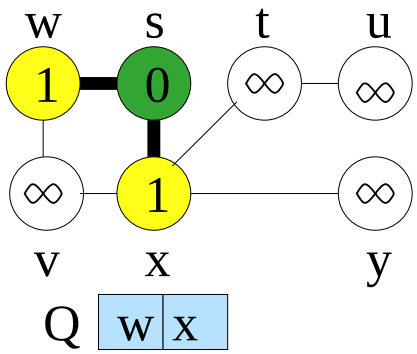
$\delta(v) \stackrel{\text{def}}{=}$ the minimum distance of v from s

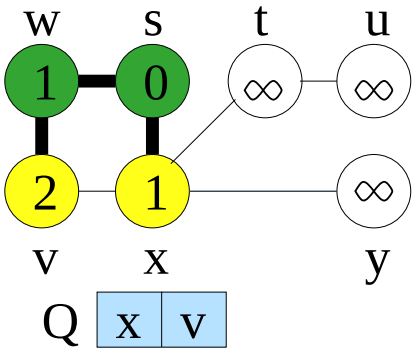
- $\delta(v) = 0$ if and only if $v = s$.
- For all $i \geq 1$, $\delta(v) = i$ if and only if $\delta(v) \notin \{0, 1, \ldots, i-1\}$ and there is a node u such that $\delta(u) = i - 1$ and $(u, v) \in E$.

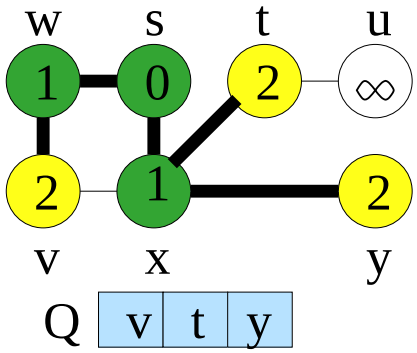
Use a queue Q. Initially, we set $Q = \{s\}$, $d(s) = 0$, and for all $v \neq s$, set $d[v] = +\infty$. Then while $Q \neq \emptyset$, do the following:

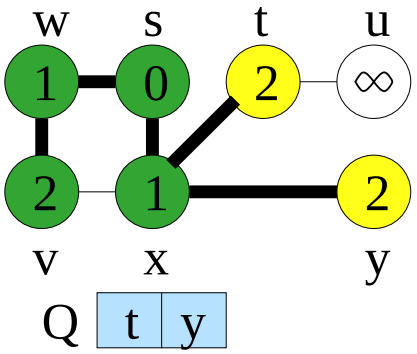
- Pop the top element u from Q.
- For each v such that $(u, v) \in E$, if $d(v) \neq +\infty$ do nothing; otherwise, set $d[v] = d[u] + 1$ and push v into Q.

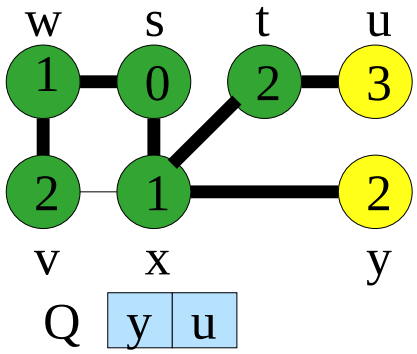

w s t u
∞ 0 ∞ ∞
∞ ∞ ∞
v x y
Q s

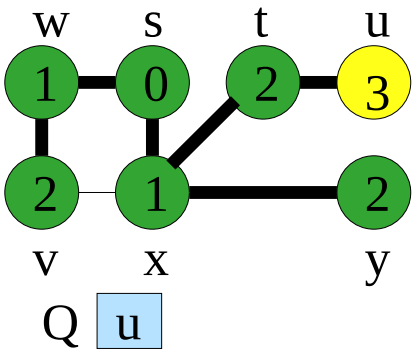

w s t u
1 0 ∞ ∞
∞ 1 ∞
v x y
Q w x

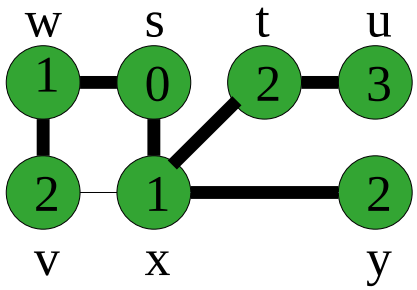

w s t u
1 0 ∞ ∞
2 1 ∞
v x y
Q x v


w s t u
1 0 2 ∞
2 1 2
v x y
Q v t y


w s t u
1 0 2 ∞
2 1 2
v x y
Q t y


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


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


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

Correctness Proof

Theorem A For each vertex v, $d[v] = \delta(v)$ at the end.

Proof Suppose that G is connected. Then every node is put in the queue at least once. Also,

- At any point of the algorithm if $Q = [v_1, \ldots, v_m]$ then $d[v_1] \leq \cdots \leq d[v_m] \leq d[v_1] + 1$.
- For all v, once $d[v]$ is set to a finite value $d[v]$ is unchanged to another finite value unless $d[v]$ becomes $+\infty$ again.

These imply that the value assigned to $d[v]$ after initialization never exceeds $n - 1$, which implies that a node is never put in the queue twice. So, every node is put in the queue exactly once.

Now we use induction on the value of $d[v]$ to show the correctness: for all $t \geq 0$ and for all v, $d[v] = t$ if and only if $\delta(v) = t$.

The base case is when $t = 0$. The proof is trivial for this case.

Why?

There is only one node whose d-value is 0.

The unique node is s.

The value of $d[s]$ is set to 0.

For the induction step, let $t > 0$ and suppose that the claim holds for all values of t less than the current one. Let v be such that $d[v] = t$. By our induction hypothesis $\delta(v) \geq t$. There is a node u such that $d[u] = t - 1$ and the algorithm sets $d[v]$ to t by identifying (u, v). By our induction hypothesis $\delta(u) = d[u]$. So, $\delta(v) \leq t$. Thus, $\delta(v) = t$. ∎

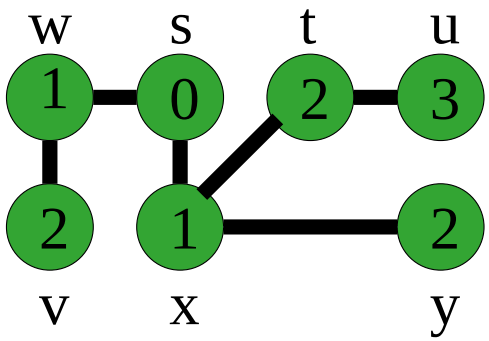
Constructing a Tree from BFS

Suppose that for all nodes v we record its "predecessor," i.e. the node from which v is touched, as $\pi[v]$. Then the edge set $\{(\pi[v], v) \mid v \in V - \{s\}\}$ defines a tree. We call it **the BFS tree** of G.

The complexity of BFS

- A node is placed in a queue just once
- An edge is examined twice

node	π
s	—
w, x	s
v	w
t, y	x
u	t



DFS

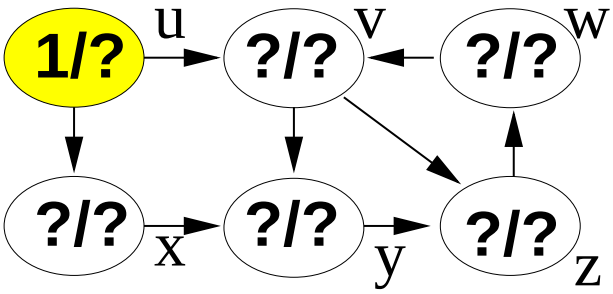
Use recursive calls to a subroutine **Visit**. Use a global clock, initially set to 0. The clock is incremented by one when **Visit** is called and when a call to **Visit** is finished.

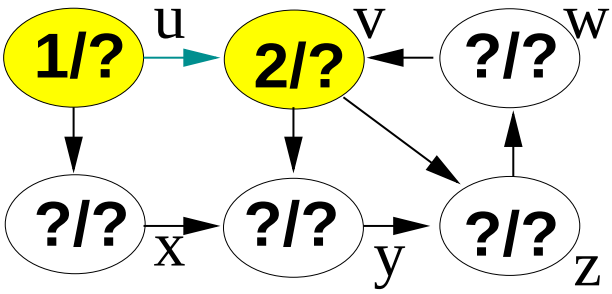
The main-loop:

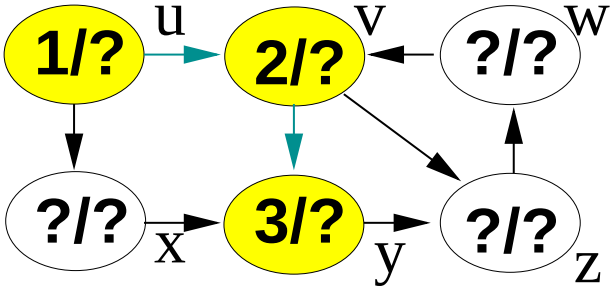
- For all u, set $d[u] = \infty$, $\pi[u] = \mathbf{nil}$, and $clock = 0$.
- For each u, if $d[u] = \infty$ then call $\mathbf{Visit}(u)$.

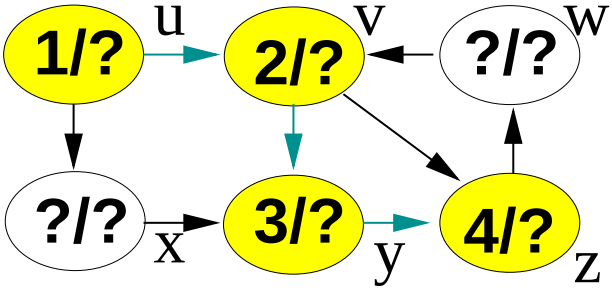
$\mathbf{Visit}(u)$:

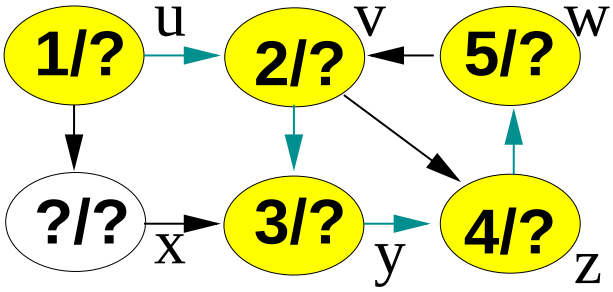
1. Add 1 to $clock$ and set $d[u] = clock$.
2. For each $v \in Adj[u]$, if $d[v] = \infty$ then set $\pi[v] = u$ and call $\mathbf{Visit}(v)$.
3. Add 1 to $clock$ and set $f[u] = clock$.

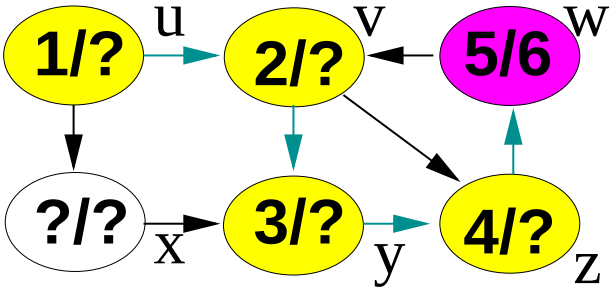

1/?
u
?/?
v
?/?
w
?/?
x
?/?
y
?/?
z

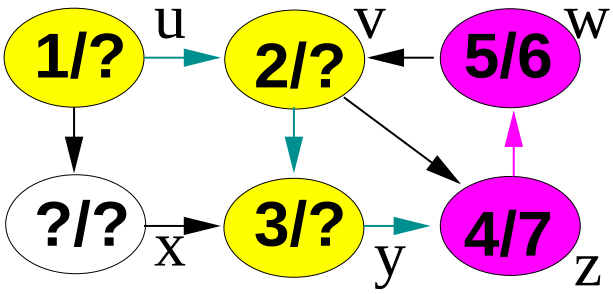

1/?
u
2/?
v
?/?
w
?/?
x
?/?
y
?/?
z

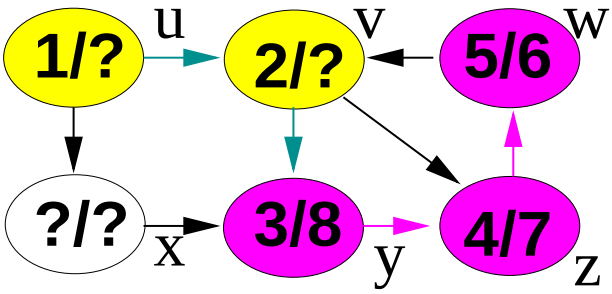

1/?
u
2/?
v
?/?
w
?/?
x
3/?
y
?/?
z

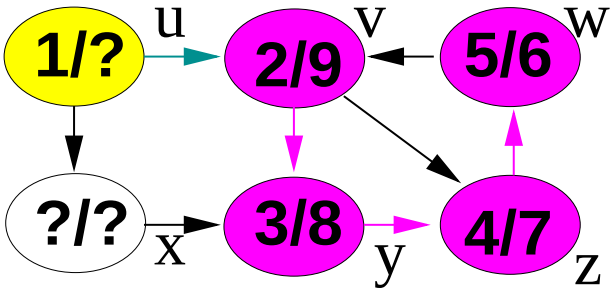

1/?
u
2/?
v
?/?
w
?/?
x
3/?
y
4/?
z

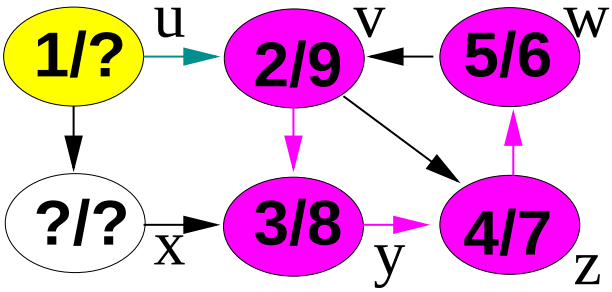

1/?
u
2/?
v
5/?
w
?/?
x
3/?
y
4/?
z

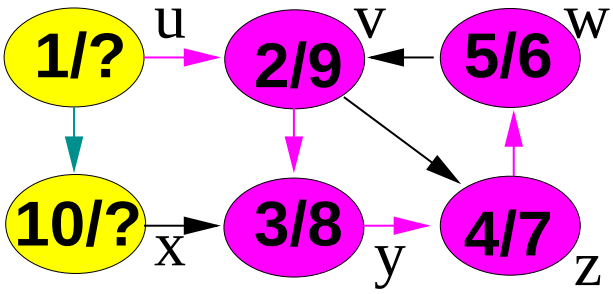

1/?
u
2/?
v
5/6
w
?/?
x
3/?
y
4/?
z

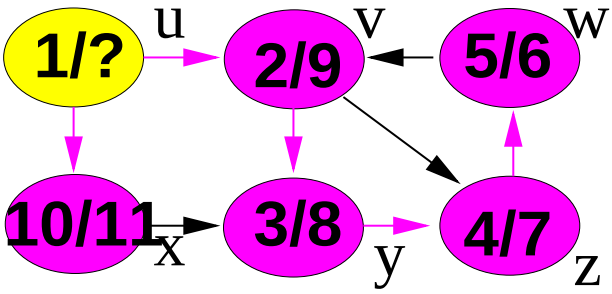

1/?
u
2/?
v
5/6
w
?/?
x
3/?
y
4/7
z


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


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


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


1/?
u
2/9
v
5/6
w
10/?
x
3/8
y
4/7
z


1/?
u
2/9
v
5/6
w
10/11
x
3/8
y
4/7
z

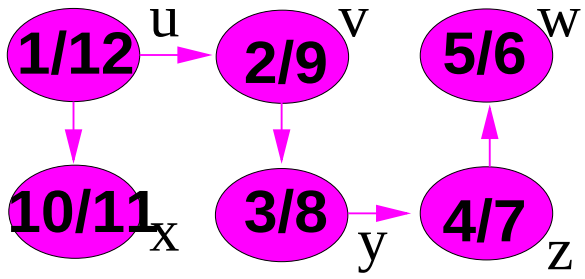
Running Time Analysis

- A call of **Visit** with respect to a node is exactly once.
- Each edge is examined exactly twice.

So, what's the running time?

Use the π field to constuct a tree, called the **DFS tree**.

node	π
u	—
v, x	u
w	z
y	v
z	y

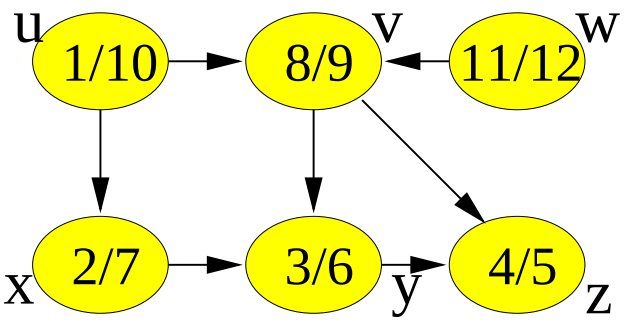


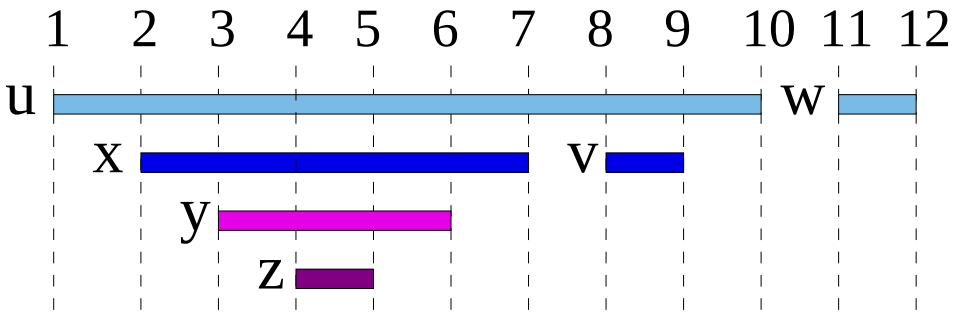
The Parenthesis Structure of DFS

For each u, let $I[u] = (d[u], f[u])$. Then, for all u and v, exactly one of the following three holds for $I[u]$ and $I[v]$,

- $I[u] \cap I[v] = \emptyset$. This is the case when u and v are not on the same path from s.
- $I[u] \subseteq I[v]$. This is the case when u is a descendant of v on a path from s.
- $I[v] \subseteq I[u]$. This is the case when v is a descendant of u on a path from s.

This is called the parenthesis structure of DFS.


u
1/10
v
8/9
w
11/12
x
2/7
3/6
y
4/5
z


1 2 3 4 5 6 7 8 9 10 11 12
u
w
x
v
y
z

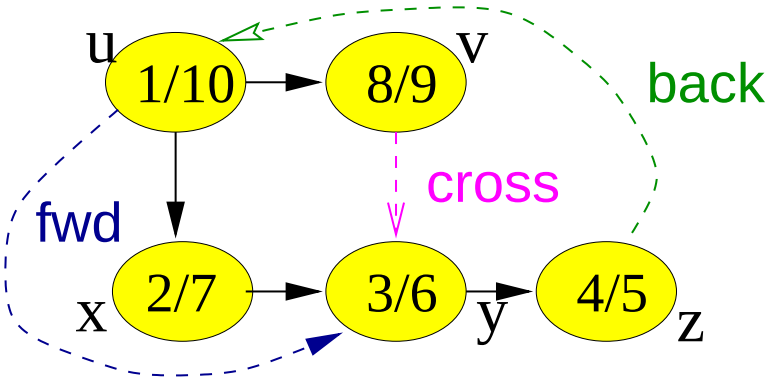
Classification of edges

1. **The Tree Edges**: The edges on the tree.
2. **The Back Edges**: The non-tree edges connecting descendants to ancestors (including self-loops).
3. **The Forward Edges**: The non-tree edges connecting ancestors to descendants.
4. **The Cross Edges**: The rest.

In $\mathbf{DFS}$, when $e = (u, v)$ is first explored:

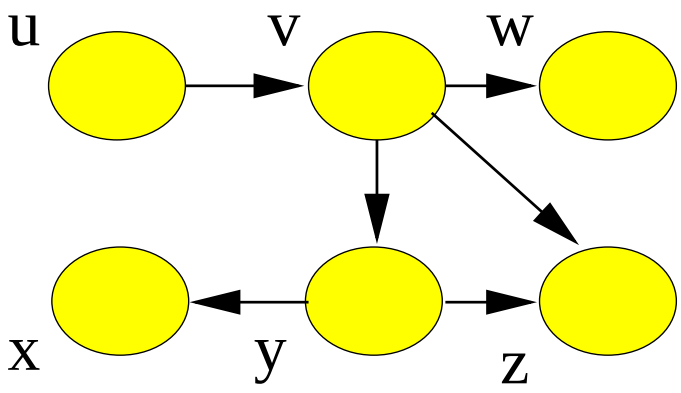
- $d[v] = \infty \Rightarrow e$ is a tree edge,
- $d[v] < f[v] = \infty \Rightarrow e$ is a back edge, and
- $f[v] < \infty \Rightarrow e$ is a forward or cross edge.

Theorem B Every edge is either a tree edge or a back edge for an undirected graph. ∎


u
1/10
8/9
v
back
cross
fwd
x
2/7
3/6
y
4/5
z

Topological sort

Let G be a DAG (directed acyclic graph). **Topological sorting** of the nodes of G is a linear ordering of the nodes such that for all u and v if there is an arc from u to v (i.e., $(u, v) \in E$) then u precedes v in the ordering.

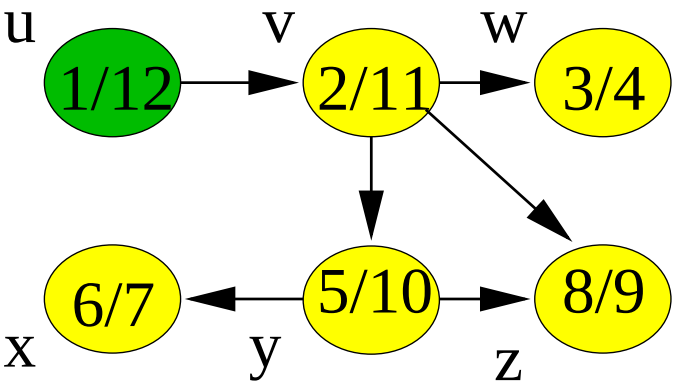


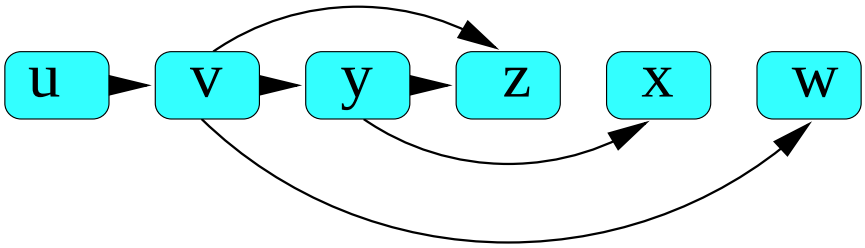
What is a topological sort of these nodes?

An Algorithm for Topological Sort

Call $\mathbf{DFS}(G)$ to compute f-values. While doing this, each time a node, say v, is done, insert v as the top element of the list.

The running time is $O(E + V)$.

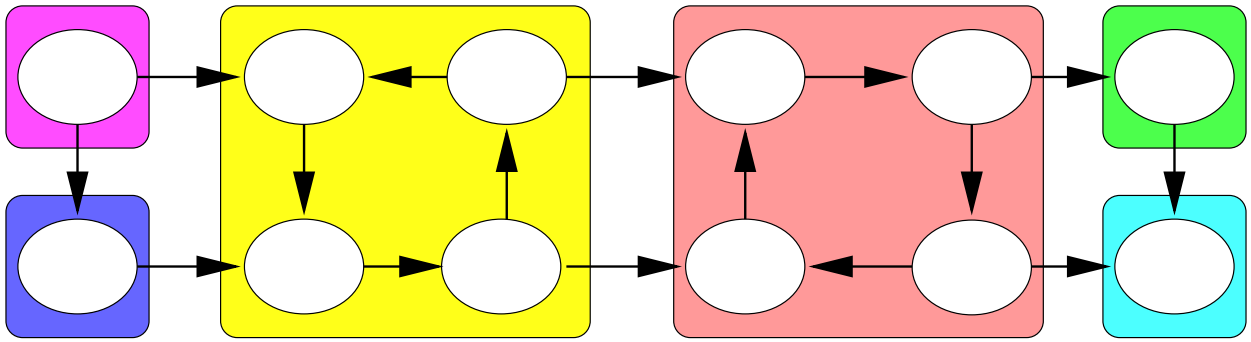

u
v
w
1/12
2/11
3/4
6/7
5/10
8/9
x
y
z


u
v
y
z
x
w

Strongly Connected Components

Let G be a directed graph. For all nodes u and v, write $u \rightsquigarrow v$ if there is a directed path from u to v in G.

Two vertices u and v of a directed graph G are **strongly connected** if $u \rightsquigarrow v$ and $v \rightsquigarrow u$. A strongly connected component of G is a maximal set S of vertices in G in which every two nodes are strongly connected.



Algorithms for Computing Strongly Connected Components

A trivial algorithm would be to compute for each u the set, W_u, defined by $\{v \mid u \rightsquigarrow v\}$, and then to check for all u and v whether it holds that $u \in W_v$ and $v \in W_u$.

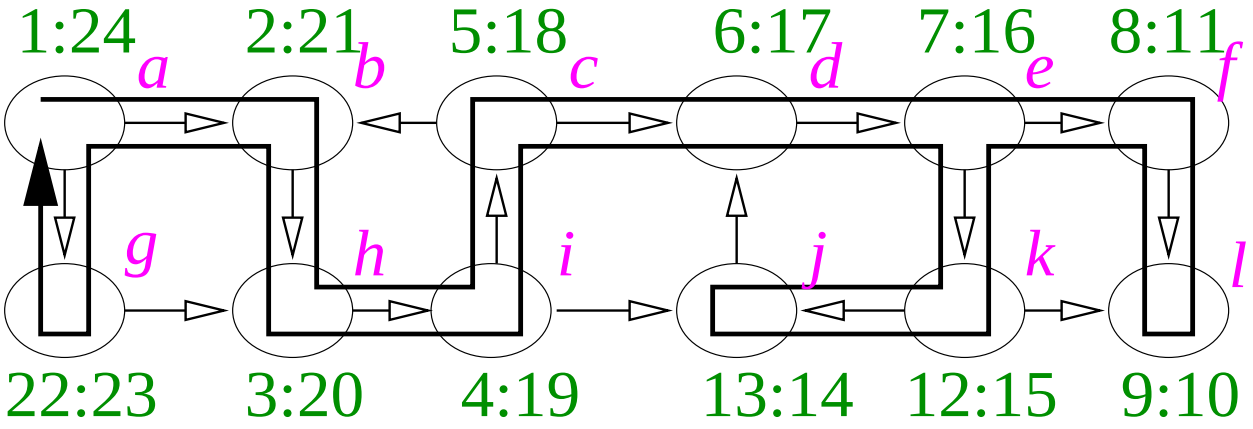
How efficiently can this algorithm be implemented?

An $O(E+V)$-Step Method

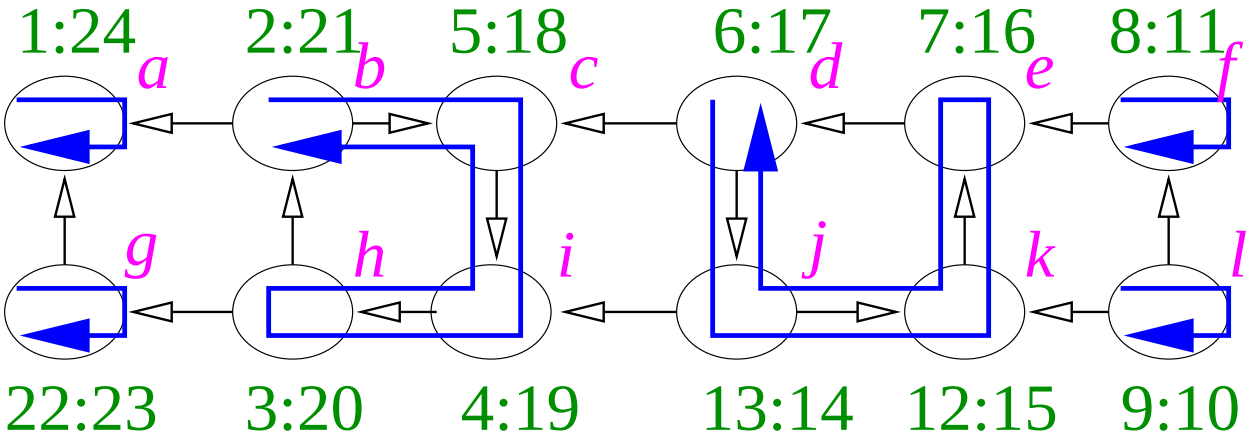
Define $G^{\mathbf{T}}$ to be the graph G in which the direction of each edge is reversed. We do the following:

1. Call $\mathbf{DFS}(G)$ to compute $f[u]$ for all u.
2. Compute $H = G^{\mathbf{T}}$ where the nodes are enumerated in order of decreasing f.
3. Call $\mathbf{DFS}(H)$, in which whenever the paths have been exhausted, find the next node that is not visited yet in the above ordering.
4. Output the vertices of each DFS-tree of H as a separate strongly connected component.

The f-values:



The DFS-trees of H:



Correctness of Strongly Connected Components

Let C, C' be SCCs in $G = (V, E)$. If there is an edge $(u, v) \in E$, where $u \in C$ and $v \in C'$, $f(C) > f(C')$

Induction on number of components.

Hypothesis: first k tree produced in second DFS are SCCs.

When visiting next vertex u,
$f[u] = F(C) > F(C')$ for any SCC C' not yet visited. Any edge leaving C in $G^{\mathbf{T}}$ is to a SCC already visited. All vertices in C will be descendents of u in DFS.