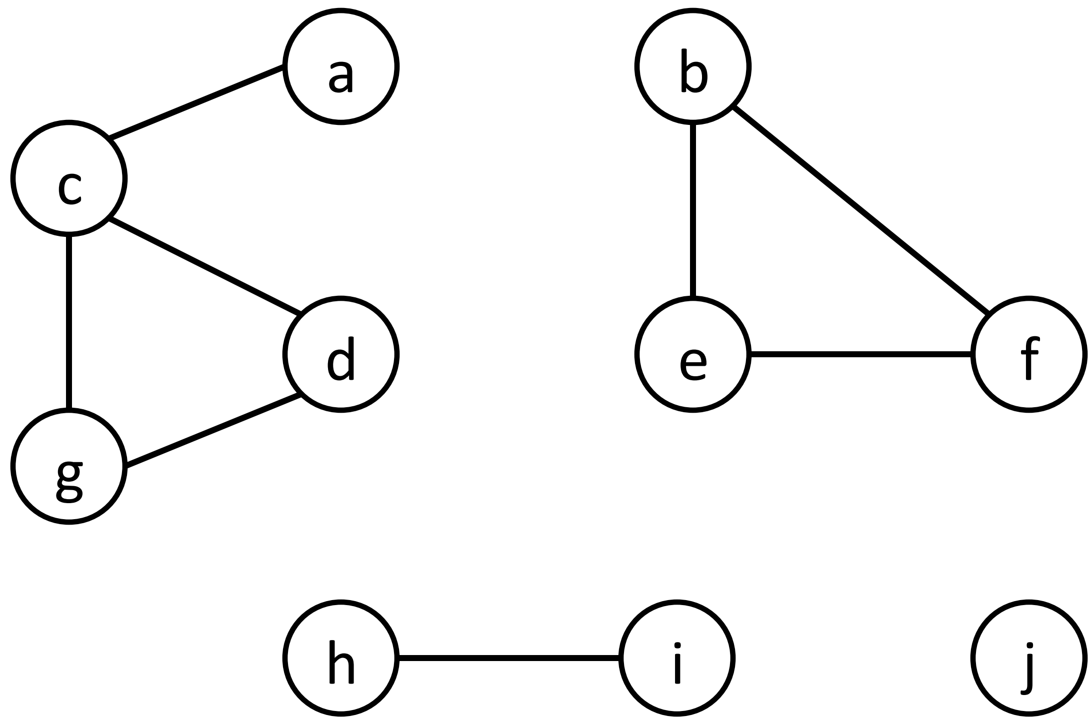
COS 528
Depth-First Search

An undirected graph

4 connected components



Vertex j is isolated: no incident edges

Undirected graph search

$G = (V, E)$ V = vertex set, E = edge set

$n = |V|, m = |E|$

Each edge $(v, w) \in E$ connects two vertices v, w; can be traversed in either direction: from v to w, or from w to v.

Graph search: From a given start vertex v, visit all vertices and edges reachable from v, once each.

Graph exploration: while some vertex is unvisited, choose a start vertex v, search from v.

Connected components: subgraphs induced by maximal sets of mutually reachable vertices: *x* and *y* are in the same component iff there is a path from *x* to *y* (and back).

To find components, do an *exploration*: each search visits the vertices and edges of one component.

Edge-guided search

Maintain a set S of traversable edges (one end visited), generate a set T of *tree arcs*

explore(V, E):

{**for** $v \in V$ **do** mark v unvisited;

$S \leftarrow \{\ \}$; $T \leftarrow \{\ \}$;

for $v \in V$ **do if** v unvisited *then search*(v)}

search(*v*):

    {*visit*(*v*);

    while $\exists(x, y) \in S$ **do**

        {$S \leftarrow S - (x, y)$;

        if (x, y) untraversed **then** *traverse*(*x*, *y*);

        if *y* unvisited **then**

            {$T \leftarrow T \cup \{(x, y)\}$; *visit*(*y*)}}

visit(*v*):{ mark *v* visited; $S \leftarrow \{(v, w) \in E\}$}

traverse(*v*, *w*): mark (*v*, *w*) traversed

Exploration traverses each edge once in each direction, generates a set of *tree arcs* that form rooted trees, one spanning each connected component; roots are start vertices. These trees form a *spanning forest*.

Graph representation:

For each vertex v, set of edges (v, w), stored in a list or in an array

Each edge is in two incidence sets

Exploration time: $O(n + m)$

Types of search

Can find connected components using *any* search order. For harder problems, specific search orders give efficient algorithms

Breadth-first (*BFS*): *S* is a queue

Depth-first (*DFS*): *S* is a stack

Recursive implementation of DFS

explore(*V*, *E*):

{**for** $v \in V$ **do** mark v unvisited;

for $v \in V$ **do** **if** v unvisited *then search*(v)}

```
search(v):
    {previsit(v) [visit(v)];
     for (v, w) ∈ E do
        if (v, w) untraversed then
           {advance(v, w) [traverse(v, w)];
            if w unvisited then
               {T ← T ∪ {(v, w)}; search(w)};
            retreat(v, w)};
     postvisit(v)}
```

previsit(v): mark v visited

traverse(v, w): mark (v, w) traversed

DFS is *local*: each advance or retreat moves to an adjacent vertex

Origins: maze traversal

preorder numbering pre(*v*): number vertices from 1 to *n* as they are previsited

postorder numbering post(*v*): number vertices from 1 to *n* as they are postvisited

Nesting lemma: *v* is an ancestor of *w* in the DFS forest iff *pre*(*v*) ≤ *pre*(*w*) and *post*(*v*) ≥ *post*(*w*)

Proof: For any vertex *v*, the preorder numbers of the descendants of *v* are consecutive, with *v* numbered smallest; the postorder numbers of the descendants of *v* are also consecutive, with *v* numbered largest.

Can implement DFS non-recursively using a stack of *current arcs*: the current arc into a vertex is its entering tree arc. The current arcs define the *current path* from the start vertex of the search to the *current vertex* of the search. The vertices on the current path are exactly those that have been previsited but not postvisited.

Graph structure imposed by search

Convert each edge into an arc by directing it in the direction it is first traversed.

In addition to generating spanning trees of the connected components, exploration imposes a structure on the non-tree arcs, depending on the type of search.

DFS

If (*v*, *w*) is a non-tree arc, *w* is an ancestor of *v* in the DFS forest. Any edge connects two related vertices

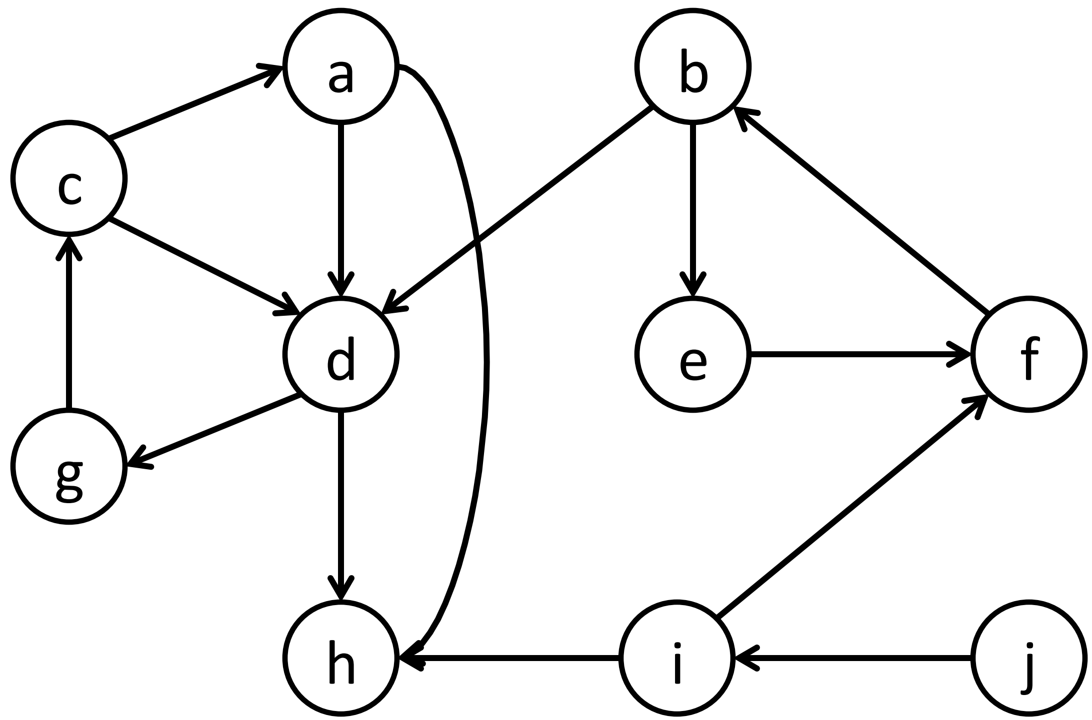
Proof: Let (*v*, *w*) be a non-tree arc. Then (*v*, *w*) is first traversed from *v*, between *previsit*(*v*) and *postvisit*(*v*). Since (*v*, *w*) is not a tree arc, *previsit*(*w*) precedes *traverse*(*v*, *w*). Since (*v*, *w*) was not traversed from *w*, *postvisit*(*w*) follows *traverse*(*v*, *w*). Thus *w* must be on the current path, and hence an ancestor of *v*.

Path lemma: Any path between v and w contains a common ancestor of v and w in the DFS forest.

Proof: Let u be a vertex of smallest depth on the path. Claim: u is an ancestor of every vertex on the path. Let x be a vertex that violates the claim. Consider the part of the path between u and x. It must contain an edge (y, z) with y but not z a descendant of u. Vertex z must be an ancestor of y, and hence must be a proper ancestor of u. But $d(z) < d(u)$, a contradiction.

Among the vertices on the path, u is smallest in preorder and largest in postorder.

A directed graph



Directed graph search

Each arc $(v, w) \in E$ can be traversed in only one direction, from v to w

Directed graph search (forward) is just like undirected graph search, except that each arc is already directed, and is in only one incident arc set: (v, w) is in the set of arcs out of v

Backward search: for each vertex, store the set of incoming arcs; to search, (conceptually) reverse the arc directions

Exploration of a digraph generates a set of tree arcs that form trees spanning the sets of vertices reached from the start vertices of the searches. Arcs can lead between trees (but only from later to earlier visited vertices). The exploration imposes a structure on the non-tree arcs, depending on the type of search. The imposed structure is weaker than in undirected graph search, but the nesting lemma holds.

DFS (digraph)

Arc Lemma: Each arc (v, w) is of one of four types:

tree arc: $pre(v) < pre(w)$, $post(v) > post(w)$, w unvisited when (v, w) is traversed

forward arc: $pre(v) < pre(w)$, $post(v) > post(w)$, w visited when (v, w) is traversed

back arc: $pre(v) > pre(w)$, $post(v) < post(w)$

cross arc: $pre(v) > pre(w)$, $post(v) > post(w)$

Proof: We show that the excluded case, $pre(v) < pre(w)$ and $post(v) < post(w)$, cannot happen. If w is unvisited when (v, w) is traversed, then (v, w) is a tree arc, and $post(v) > post(w)$. If w is visited when (v, w) is traversed, but $pre(v) < pre(w)$, w must be previsited between the previsit and the postvisit of w. This implies w is a descendant of v; hence $post(v) > post(w)$.

Preorder lemma: Let P be a path whose first vertex u has $pre(u) = \min\{pre(x) \mid x \text{ on } P\}$. Then u is an ancestor of every vertex on P.

Proof: Suppose the lemma is false. Let (y, z) be the first arc on the path with z not a descendant of u. Then $pre(z) < pre(y)$. (Otherwise, z is a descendant of y and hence of u. But $pre(z) > pre(u)$. Thus z is previsited between the previsit to u and the previsit to y, which implies z is a descendant of u.

Postorder lemma: Let P be a path whose last vertex u has $post(u) = \max\{post(x) \mid x \text{ on } P\}$. Then u is an ancestor of every vertex on P.

Proof: Suppose the lemma is false. Let (y, z) be the last arc on P such that y is not a descendant of u. Since $post(y) < post(u)$, y is not an ancestor of u; hence it is unrelated to u, and to z. Thus (y, z) is a cross arc, so $pre(y) > pre(z)$. But $pre(z) \geq pre(u)$, so $pre(y) > pre(u)$, which implies y is a descendant of u.

Path Lemma (digraph): If $pre(v) \le pre(w)$ or $post(v) \le post(w)$, any path P from v to w contains a common ancestor of v and w.

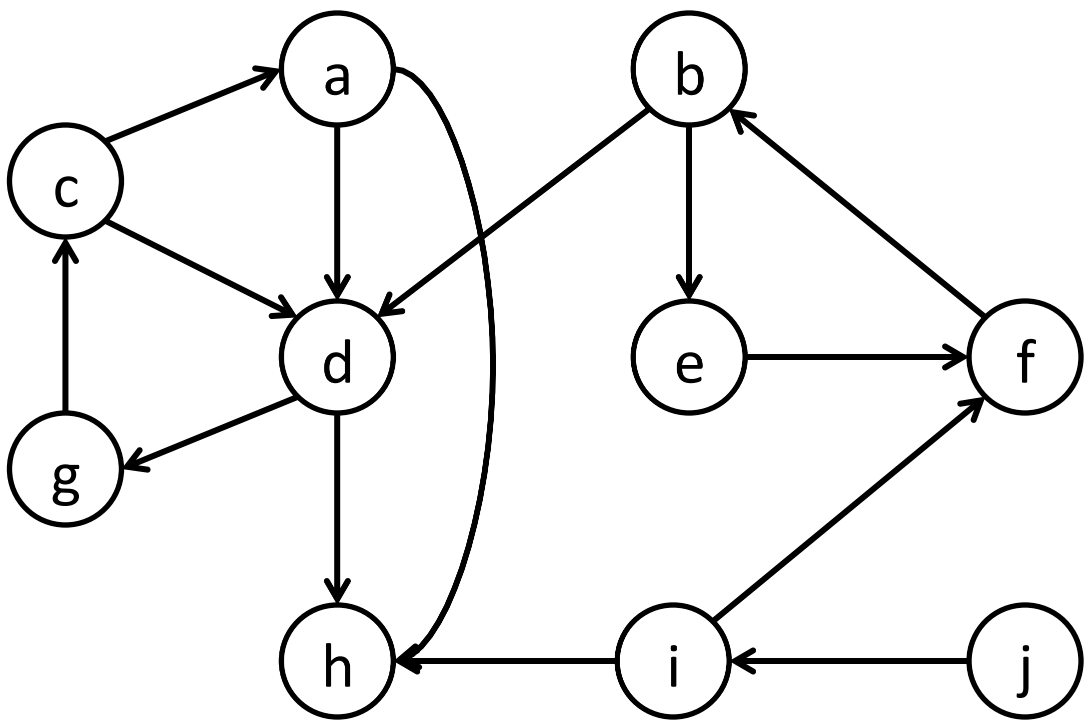
Proof: Let $x = \text{argmin}\{pre(u) | u \text{ on } P\}$ and $y = \text{argmin}\{post(u) | u \text{ on } P\}$. By the preorder lemma, x is an ancestor of w; by the postorder lemma, y is an ancestor of v. If $x = y$, the lemma holds. Suppose $x \ne y$. Since $pre(x) < pre(y)$ and $post(x) < post(y)$, x and y are unrelated. But w a descendant of x and v a descendant of y implies $pre(w) < pre(v)$ and $post(w) < post(v)$, contradicting the hypothesis of the lemma.

Finding a topological order or a cycle

Number the vertices from n to 1 in postorder. This is *reverse postorder*, *rpost*(v). If no arc (v, w) has *rpost*(v) ≥ *rpost*(w), then reverse postorder is a *topological order*: every arc leads from a smaller to a larger vertex. If some arc (v, w) has *rpost*(v) > *rpost*(w), then w is an ancestor of v, and there is a cycle consisting of (v, w) and the path from w to v in the DFS forest

→DFS gives an $O(n + m)$-time algorithm to find either a topological order or a cycle

Depth-first exploration: search from a visits a, d, h, g, c; search from b visits b, e, f; search from i visits i; search from j visits j

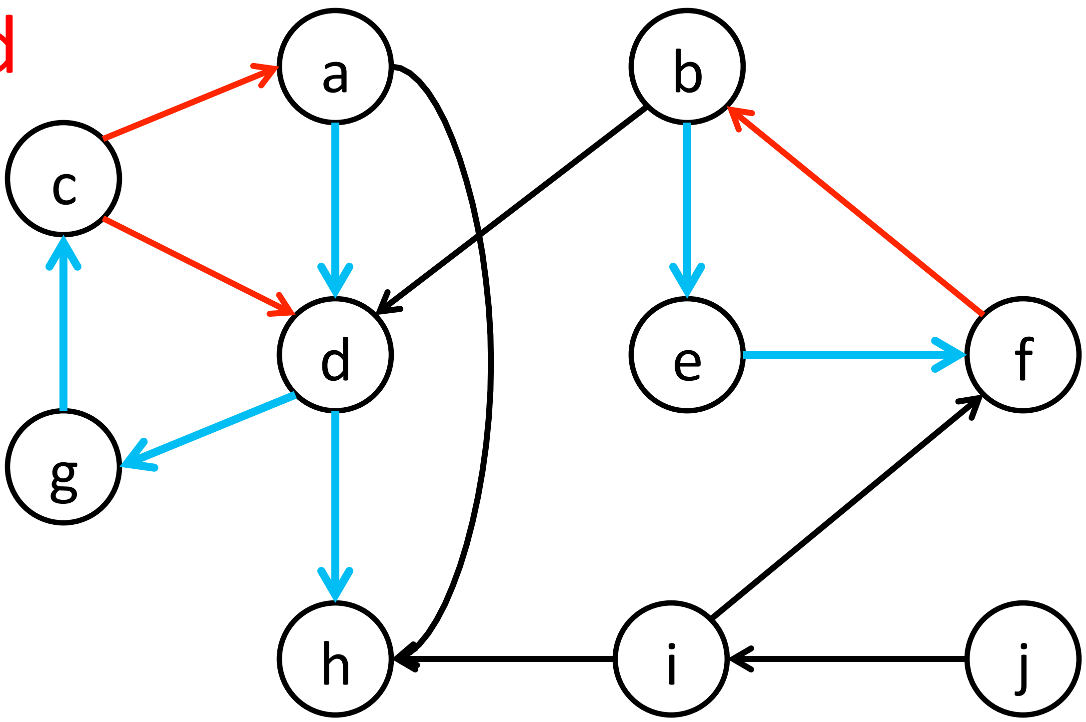


preorder a, d, h, g, c; b, e, f; i; j

postorder h, c, g, d, a; f, e, b; i; j

tree arcs in blue

cycle arcs in red



Alternative topological order algorithm

while there is a vertex *v* with no incoming arcs

do {give *v* the next number;

delete *v* and its outgoing arcs}

If this algorithm successfully numbers all the vertices, the numbering is topological. If not, every remaining vertex has at least one incoming arc, can find a cycle by doing a backward DFS from any vertex.

Efficient implementation

For each vertex x, compute $in(x)$, the number of arcs into x:

{initialize $in(x) \leftarrow 0$ for all x;
for arc $(v, w) \in E$ **do** add 1 to $in(w)$}

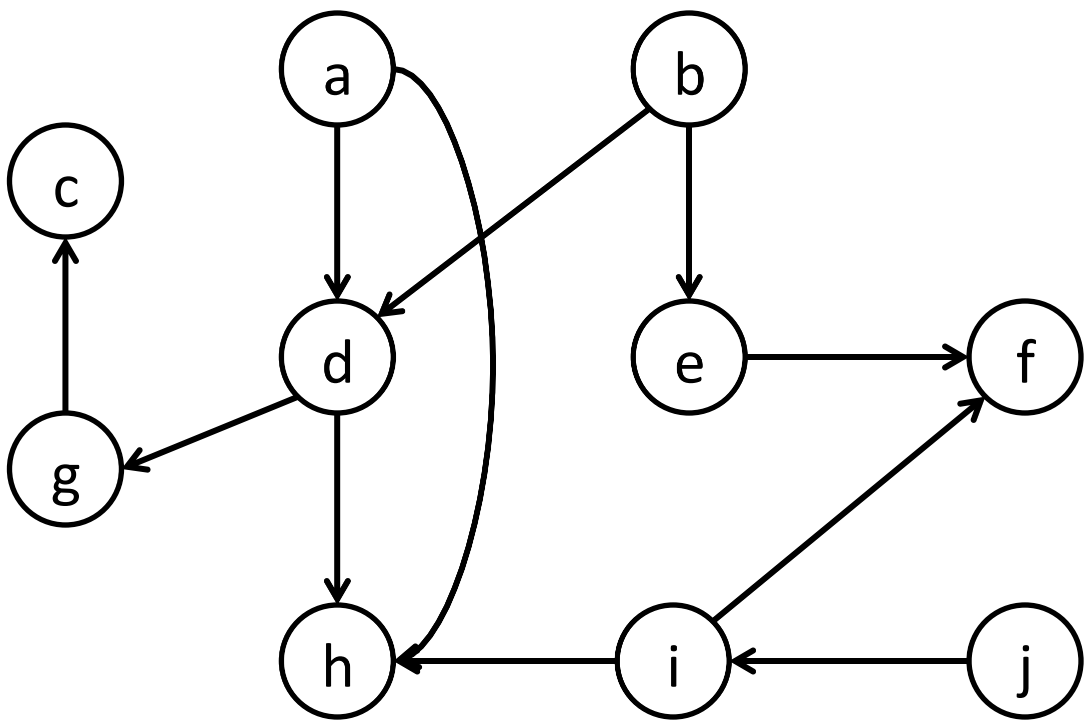
Initialize a set Z containing all vertices x with $in(x) = 0$.

while $\exists x \in Z$ **do**

{delete x from Z; number x;
for $(x, y) \in E$ **do**
{subtract 1 from $in(y)$;
if $in(y) = 0$ then insert y in Z}}

Topological order via alternate algorithm, with *Z* implemented as a queue:

a, b, j, d, e, i, g, h, f, c



Running time = O(n + m)

By choosing each candidate vertex in all possible ways, the alternate algorithm can generate all possible topological orders

Not true of the DFS algorithm: some acyclic graphs have topological orders that cannot be generated by DFS