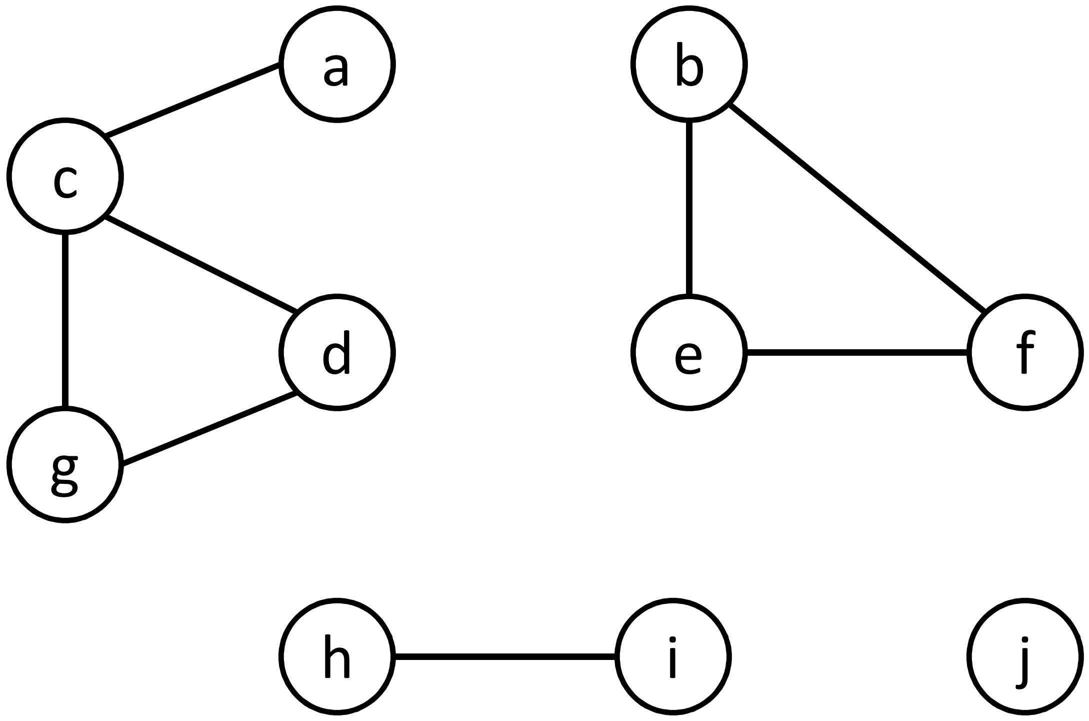
# COS 423 Lecture14
# Graph Search

# 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** $\exists(v, w) \in S$ **do**
        {$S \leftarrow S - (v, w)$;
         $traverse(v, w)$;
         **if** $w$ unvisited **then**
            {$T \leftarrow T \cup \{(v, w)\}$; $visit(w)$}}

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

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

## Vertex-guided search

Maintain a set *U* of vertices to visit

```
explore($V$, $E$):
  {for $v \in V$ do mark $v$ unvisited;
   $T \leftarrow \{\ \}$;
   for $v \in V$ do if $v$ unvisited then
       {$U \leftarrow \{v\}$; search}
```

```
search:
  while $\exists v \in U$ do
    $\{U \leftarrow U - v$; visit($v$);
     for $(v, w) \in E$ do
       $\{$traverse($v, w$);
        if $w$ unvisited then
          $\{T \leftarrow T \cup \{(v, w)\}; U \leftarrow U \cup \{w\}\}\}\}$
```

*visit*($v$): mark $v$ visited

BFS is a vertex-guided search (*U* = queue), *not* DFS: traversals of edges incident to a vertex are not necessarily consecutive

Other types of vertex-guided search:

*Shortest-first*: *U* is a heap, key = cost

*Maximum-cardinality*: *U* is a heap,

key of *v* = #adjacent visited vertices

## 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 $(v, w) \in E$ do
         {$advance(v, w)$ [$traverse(v)$];
          if $w$ unvisited then
              {$T \leftarrow T \cup \{(v, w)\}$; $search(w)$};
          $retreat(v, w)$};
     $postvisit(v)$}
```

*previsit*($v$): mark $v$ visited

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

**Origins**: maze traversal

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

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

**Nesting lemma**: *v* is an ancestor of *w* in the DFS forest iff $v \leq_{pre} w$ and $v \geq_{post} w$

**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.

## BFS

If $(v, w)$ is an arc, $0 \leq d(w) - d(v) \leq 1$, where $d(x)$ is the depth of $x$ in the BFS forest: every edge connects two vertices at the same depth or at adjacent depths.

**Proof**: Make the search vertex-guided. Define *passes*. Pass 0 is the first iteration of the **while** loop. Pass $k + 1$ is all iterations that visit vertices added to $U$ during pass $k$. An induction shows that vertex $v$ is visited during pass $d(v)$. Let $(v, w)$ be an edge that is first traversed as a result of the visit to $v$. Then $w$ will be visited before or during pass $d(v) + 1$. Thus $d(w) \leq d(v) + 1$. Since $(v, w)$ was not traversed before pass $d(v)$, $d(w) \geq d(v)$.

**Distance Lemma**: If $G$ is connected, for each vertex $v$, $d(v)$ is the minimum number of edges on a path from the start vertex to $v$.

Proof: By induction on $d(v)$.

## Application to finding small cutsets

Removing the vertices at any depth breaks the graph into (at least) two connected components: the vertices at smaller depths and those at larger depths.

Removing the edges between any pair of adjacent depths does the same thing.

# 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

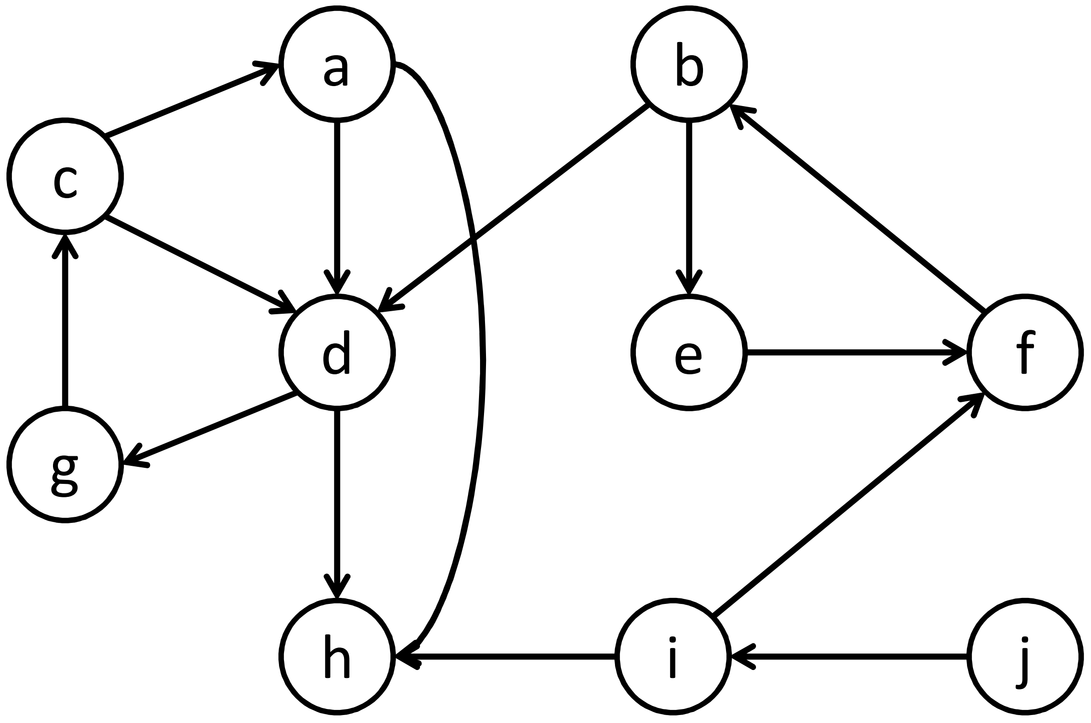
**Proof**: Let (*v*, *w*) be a non-tree arc. Then (*v*, *w*) is first traversed from *v*, between *previsit*(*v*) and *postvisit*(*v*). Since (*v*, *w*) is not a tree arc, *previsit*(*w*) precedes *traverse*(*v*, *w*). Since (*v*, *w*) was not traversed from *w*, *postvisit*(*w*) follows *traverse*(*v*, *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.

# BFS (digraph)

If $(v, w)$ is an arc, $d(w) - d(v) \leq 1$, where $d(x)$ is the depth of $x$ in the BFS forest.

**Proof**: Make the search vertex-guided. Define *passes*. Pass 0 is the first iteration of the **while** loop. Pass $k + 1$ is all iterations that visit vertices added to $U$ during pass $k$. An induction shows that vertex $v$ is visited during pass $d(v)$. Let $(v, w)$ be an edge that is traversed as a result of the visit to $v$. Then $w$ will be visited before or during pass $d(v) + 1$. Thus $d(w) \leq d(v) + 1$.

**Distance Lemma** (digraph): Suppose every vertex is reachable from the start vertex of the first search. For each vertex *v*, *d*(*v*) is the minimum number of edges on a path from the start vertex to *v*.

**Proof**: By induction on *d*(*v*).

## DFS (digraph)

The nesting lemma holds for digraphs by the same proof as for graphs

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

- **tree arc**: $v <_{pre} w$, $v >_{post} w$, $w$ unvisited when $(v, w)$ is traversed
- **forward arc**: $v <_{pre} w$, $v >_{post} w$, $w$ visited when $(v, w)$ is traversed
- **back arc**: $v >_{pre} w$, $v <_{post} w$
- **cross arc**: $v >_{pre} w$, $v >_{post} w$

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

**Preorder lemma**: Let *P* be a path whose first vertex *u* is minimum on P in preorder. 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 $z <_{pre} y$. (Otherwise, *z* is a descendant of *y* and hence of *u*.) But $z >_{pre} u$. 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* is maximum on P in postorder. 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*. Then $y <_{pre} u \leq_{pre} z$. Then *y* is an ancestor of *z*, and hence related to *u*. But $y <_{post} u$ implies *y* is a descendant of *u*.

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

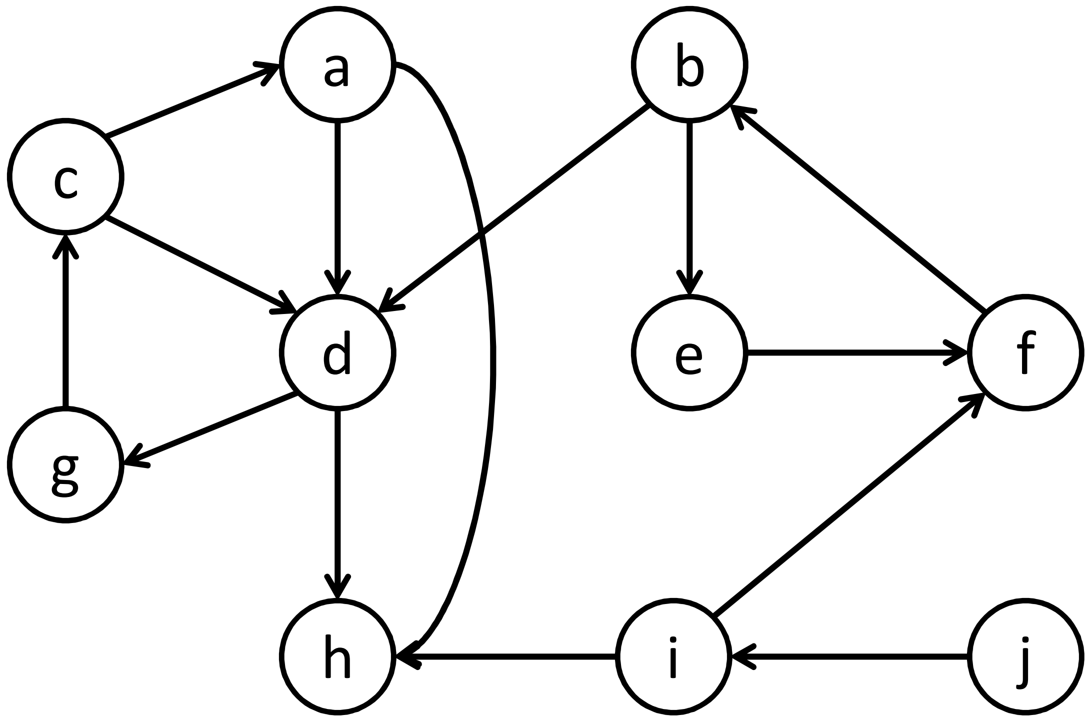
**Proof**: Let $x$ be minimum on P in preorder and $y$ maximum on P in postorder. By the preorder lemma, $x$ is an ancestor of $w$; by the postorder lemma, $y$ is an ancestor of $v$. If $x = y$, the lemma holds. Suppose $x \neq y$. Since $x <_{pre} y$ and $x <_{post} y$, $x$ and $y$ are unrelated. But $w$ a descendant of $x$ and $v$ a descendant of $y$ implies $w <_{pre} v$ and $w <_{post} v$, 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 $v \geq_{rpost} w$, then reverse postorder is a *topological order*: every arc leads from a smaller to a larger vertex. If some arc ($v$, $w$) has $v >_{rpost} w$, 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

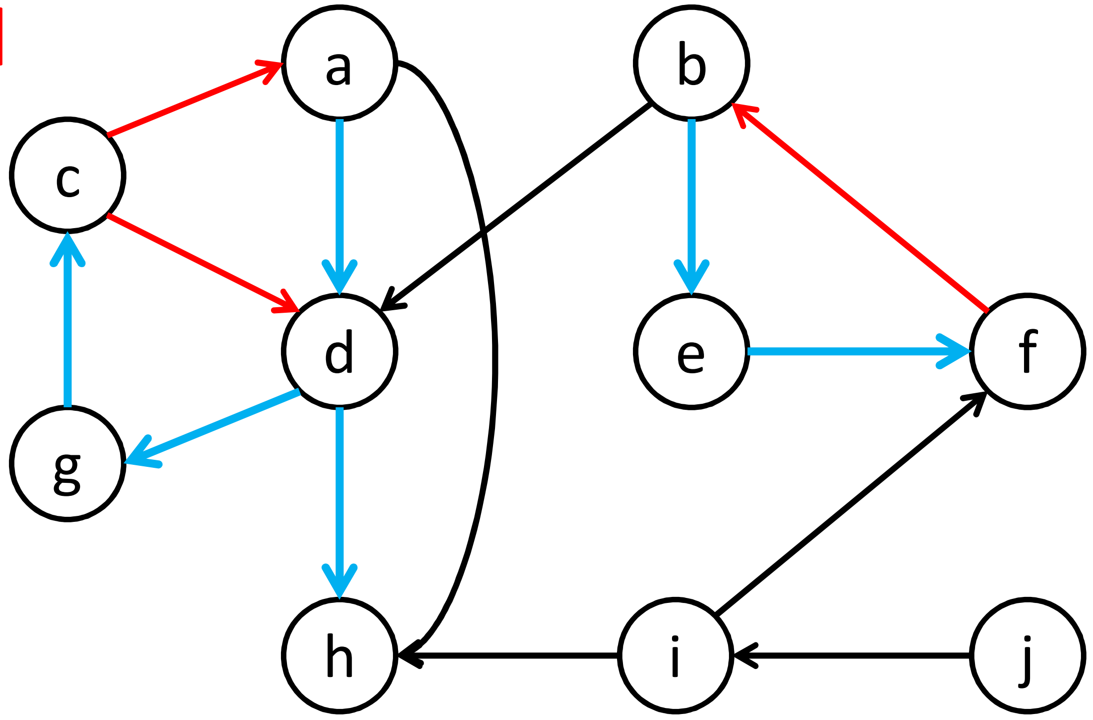


preorder a, d, h, g, c; b, e, f; i; j

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

tree arcs in blue

back arcs in red



## Alternate 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}
```

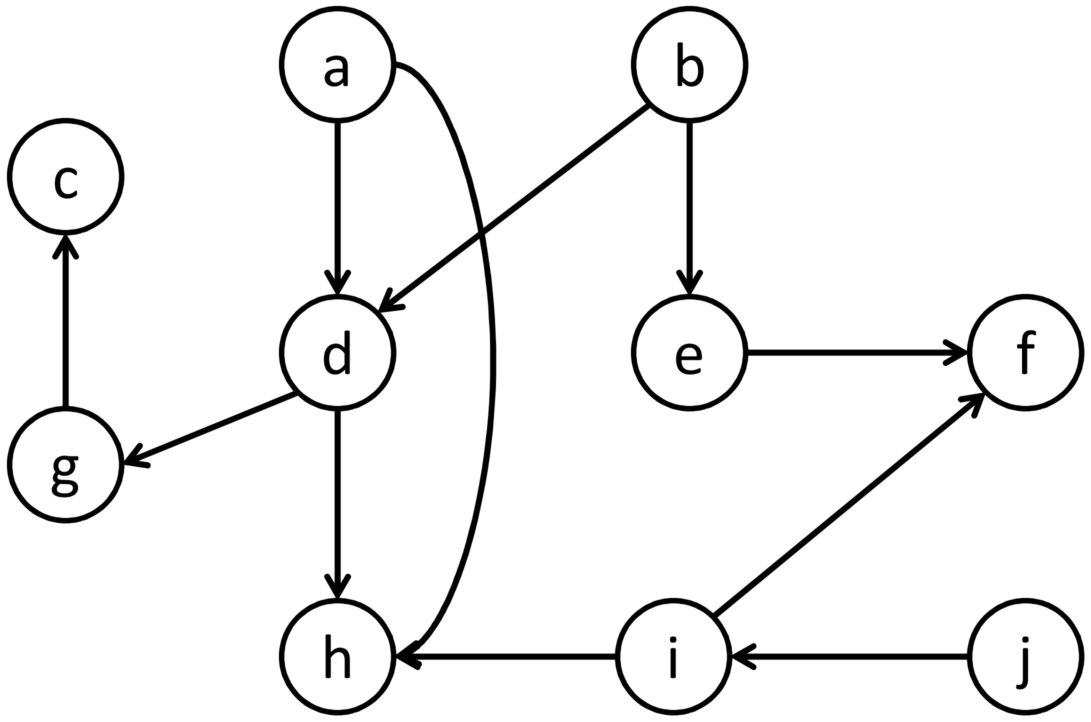
If this algorithm successfully numbers all the vertices, the numbering is topological. If not, every remaining vertex has at least one incoming arc, can find a cycle by doing a DFS backward from any vertex until reaching a previously visited vertex

# 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

## Undirected DFS to find blocks

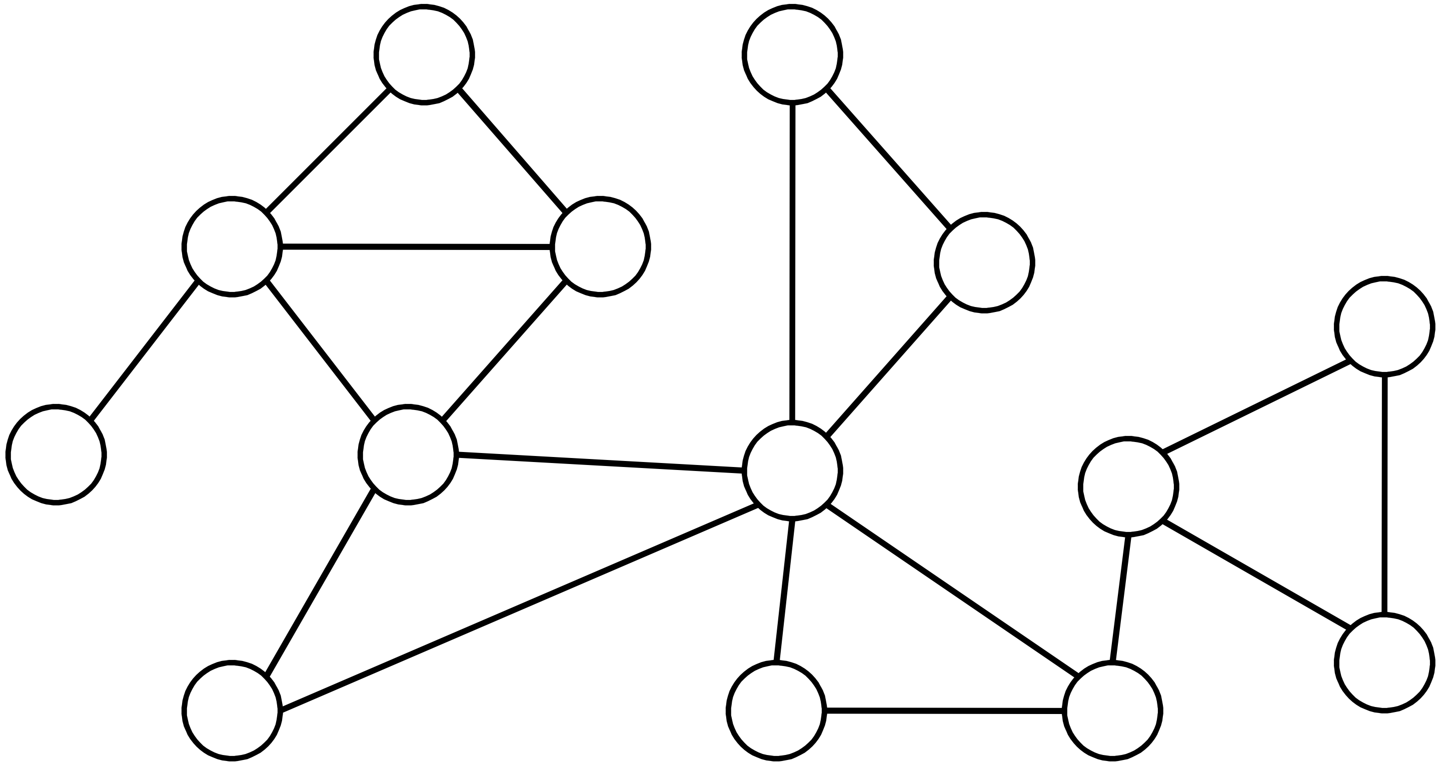
*Cut vertex*: removal disconnects the connected component containing it

*Bridge*: edge whose removal disconnects the connected component containing it

*Block*: Maximal subgraph such that any two arcs are on a simple cycle; the blocks partition the arcs

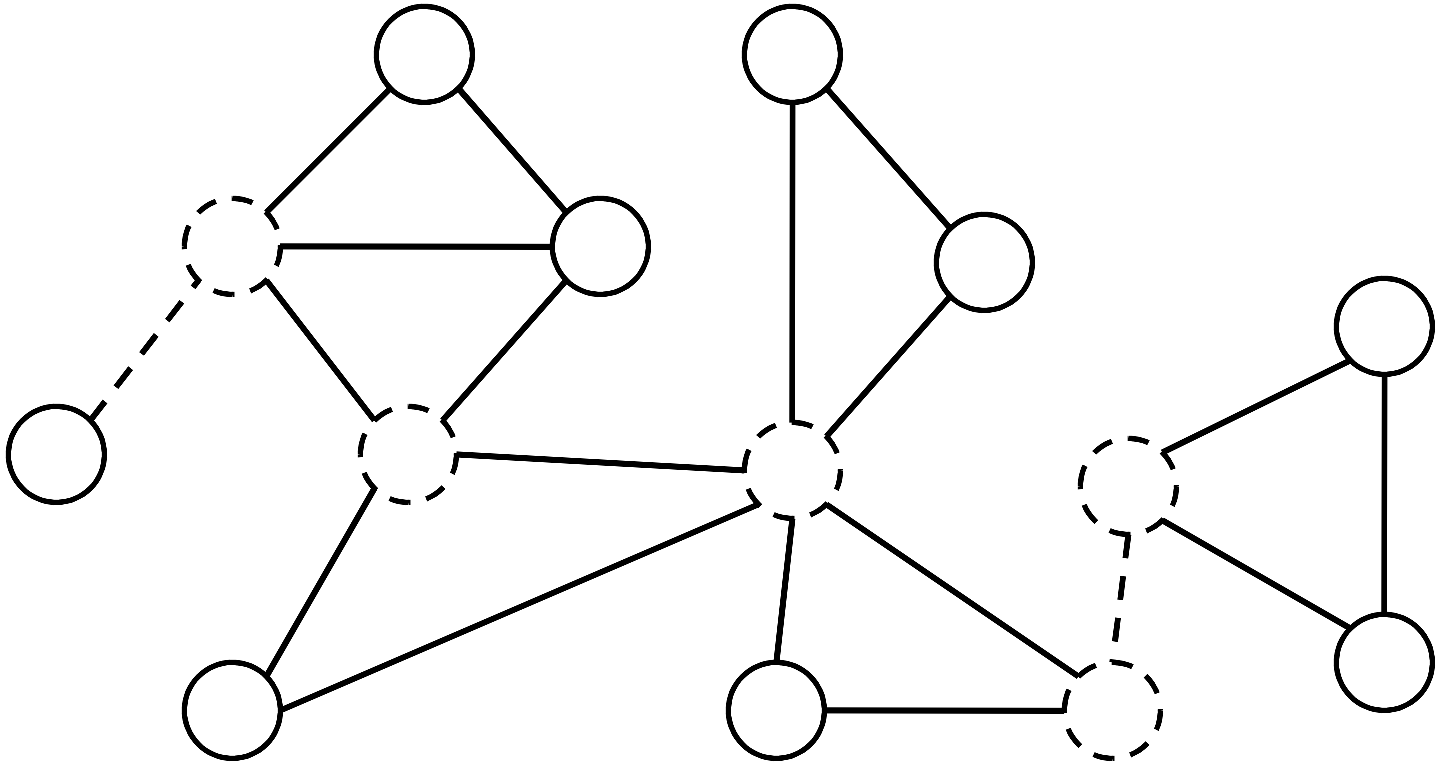
Bridge component: maximal subgraph such that any two vertices are on a cycle (not necessarily simple)

Blocks



## Bridges and cut vertices dashed

## 7 blocks, 3 bridge components



The blocks partition the edges

The bridge components partition the vertices

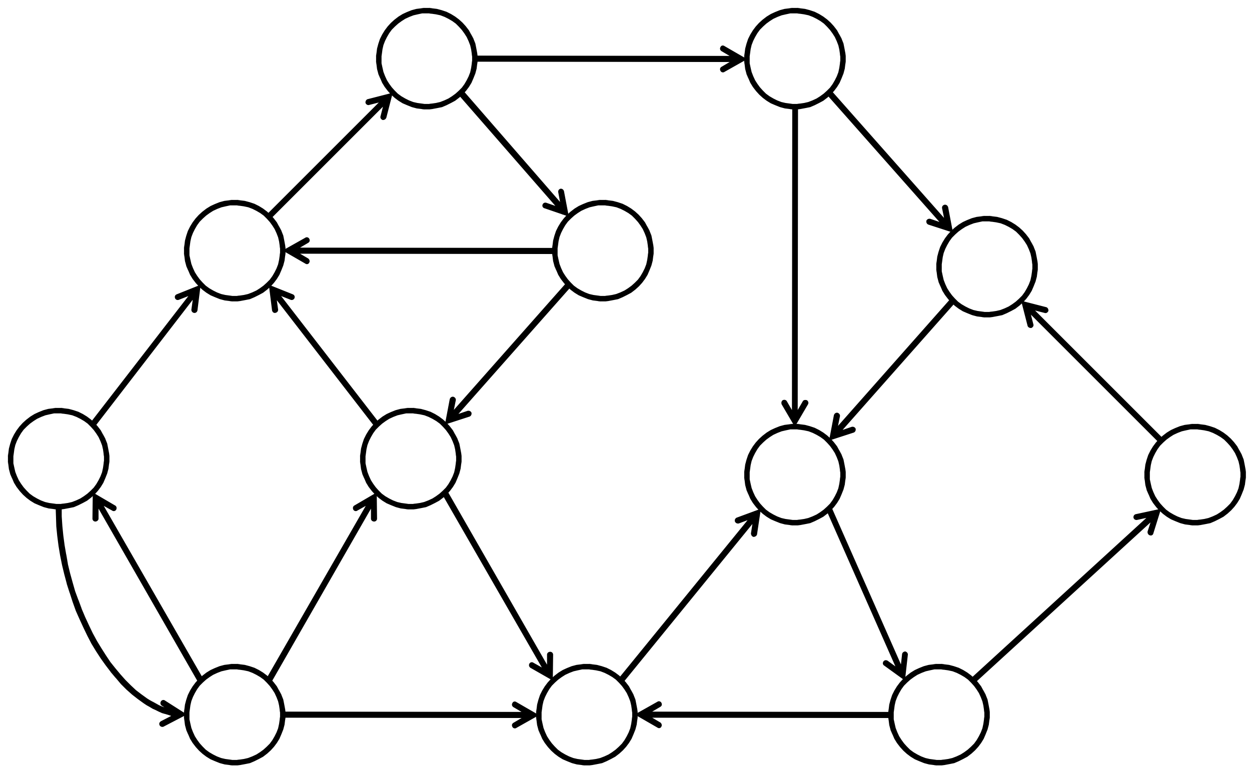
Each bridge is a block

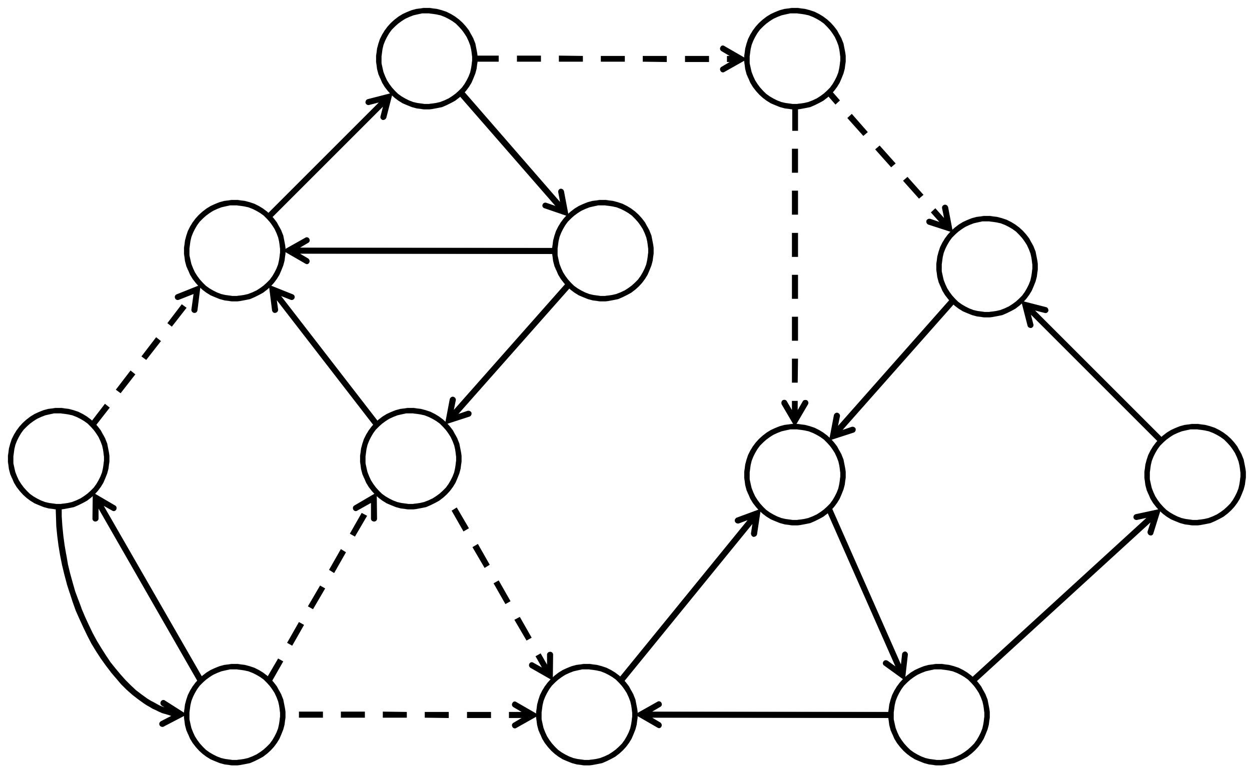
Goal: find cut vertices, bridges, blocks, bridge components, in O($n$ + $m$) time

# Directed DFS to find strong components

Two vertices *v* and *w* are *strongly connected* if there is a path from *v* to *w* and a path from *w* to *v*. Strong connectivity is an equivalence relation. A *strong component* is a subgraph induced by a maximal set of strongly connected vertices. The strong components can be *topologically ordered*: numbered so that each arc is either within a component or leads from a smaller component to a larger.

# Strong components





Goal: find strong components, and a topological order of them, in O($n$ + $m$) time.