Graph Algorithms

Textbook Reading

Chapter 22

Overview

Design principle:

- Learn the structure of the graph by systematic exploration.

Proof technique:

- Proof by contradiction

Problems:

- Connected components
- Bipartiteness testing
- Topological sorting
- Strongly connected components

Graphs, Vertices, and Edges

A graph is an ordered pair $G = (V, E)$.

- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).

Graphs, Vertices, and Edges

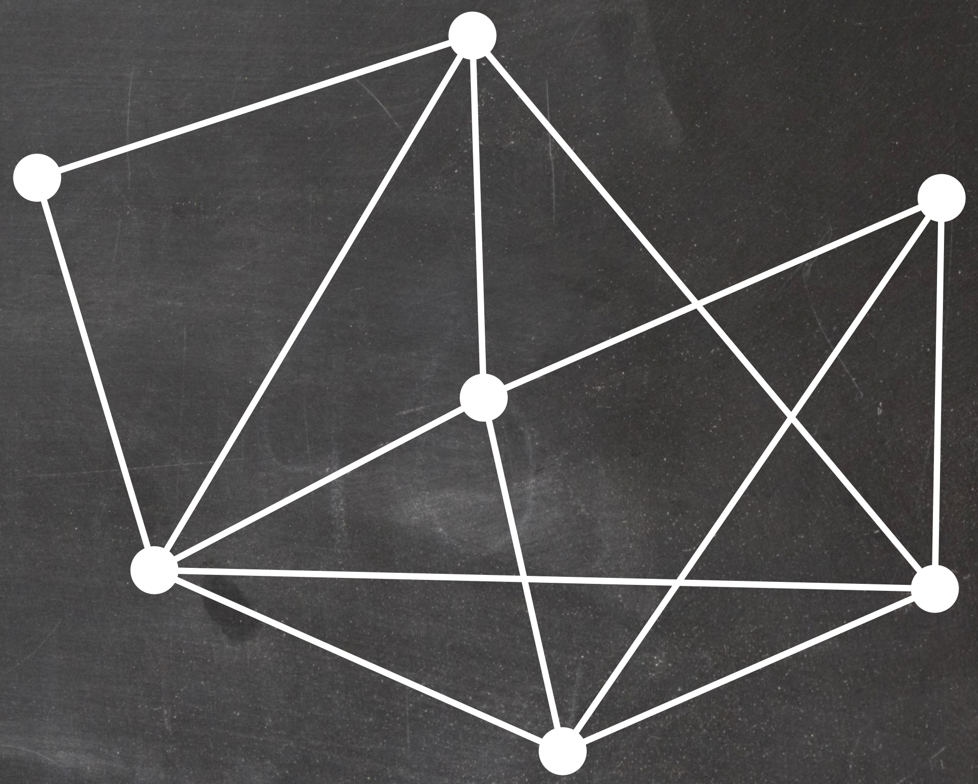
A graph is an ordered pair $G = (V, E)$.

- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).

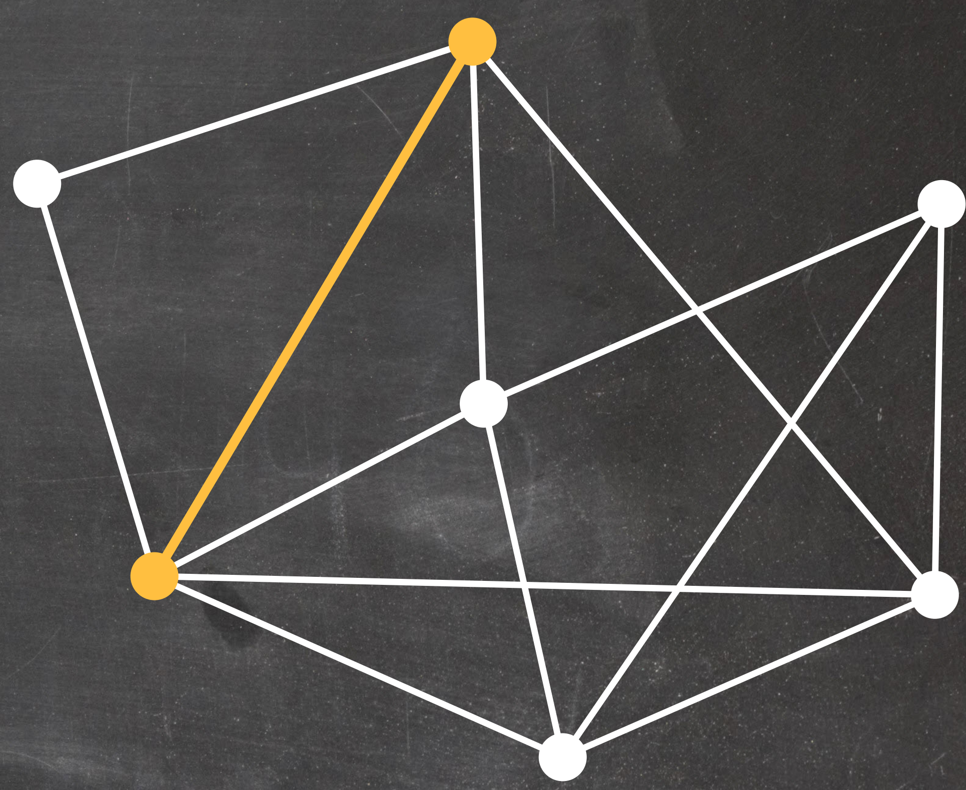
Graphs, Vertices, and Edges

A graph is an ordered pair $G = (V, E)$.

- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).



Graphs, Vertices, and Edges



A graph is an ordered pair $G = (V, E)$.

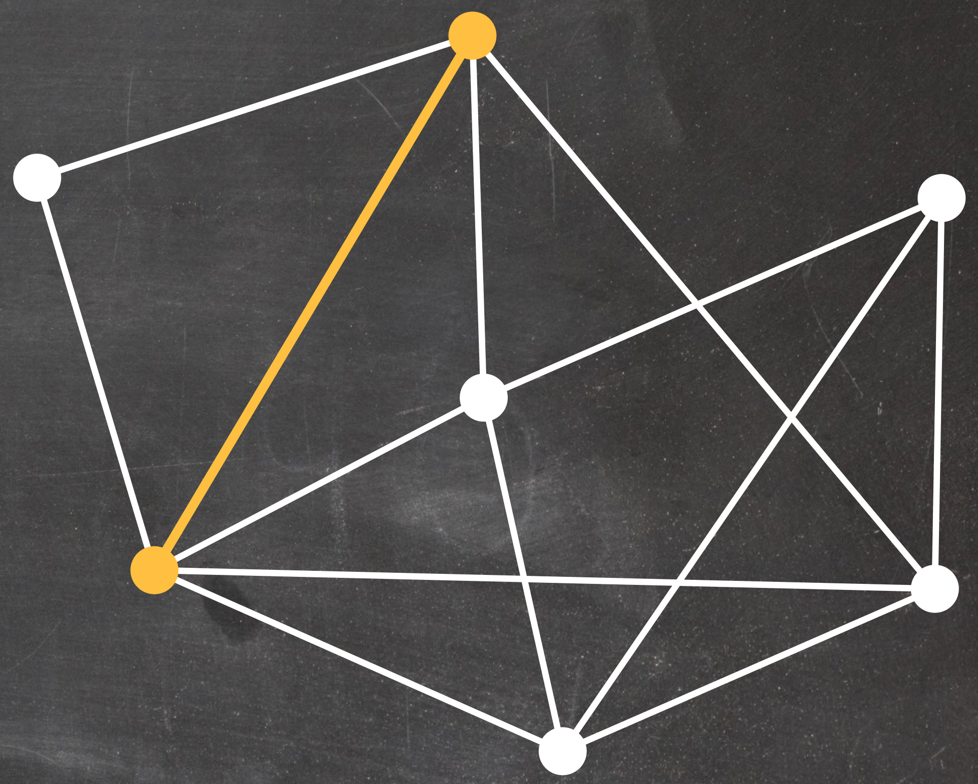
- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).

The endpoints of an edge (v, w) are v and w.

Graphs, Vertices, and Edges

A graph is an ordered pair G = (V, E).

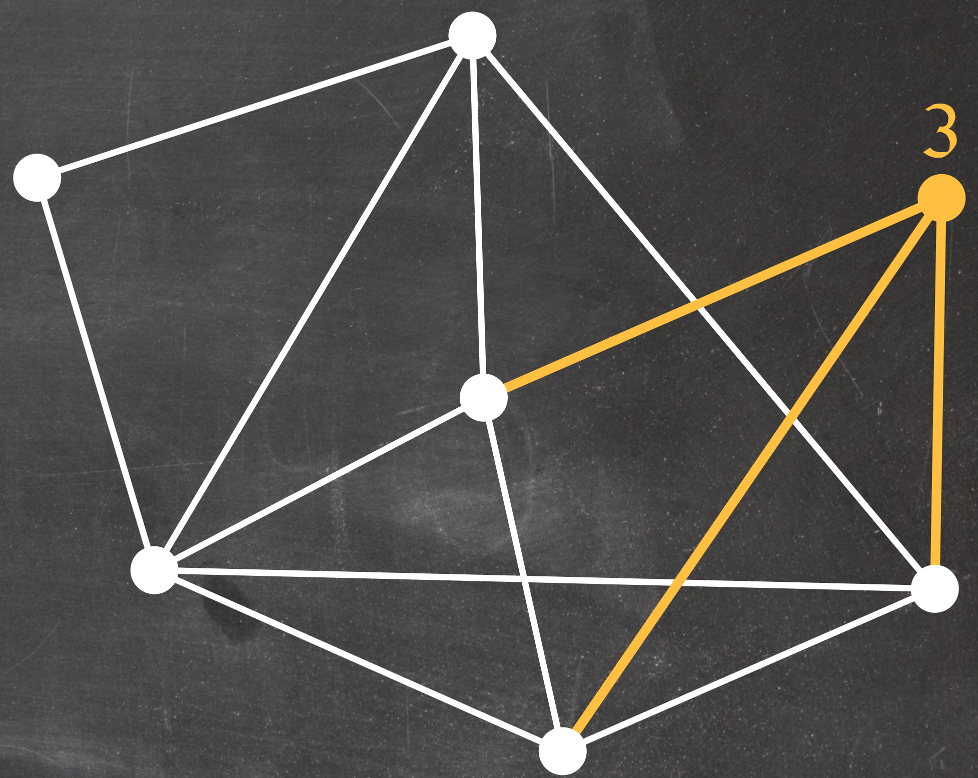
- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).



The endpoints of an edge (v, w) are v and w.

The endpoints of an edge e are said to be adjacent to each other and incident with e.

Graphs, Vertices, and Edges



A graph is an ordered pair $G = (V, E)$.

- V is the set of vertices of G.
- E is the set of edges of G.
- The elements of E are pairs of vertices (v, w).

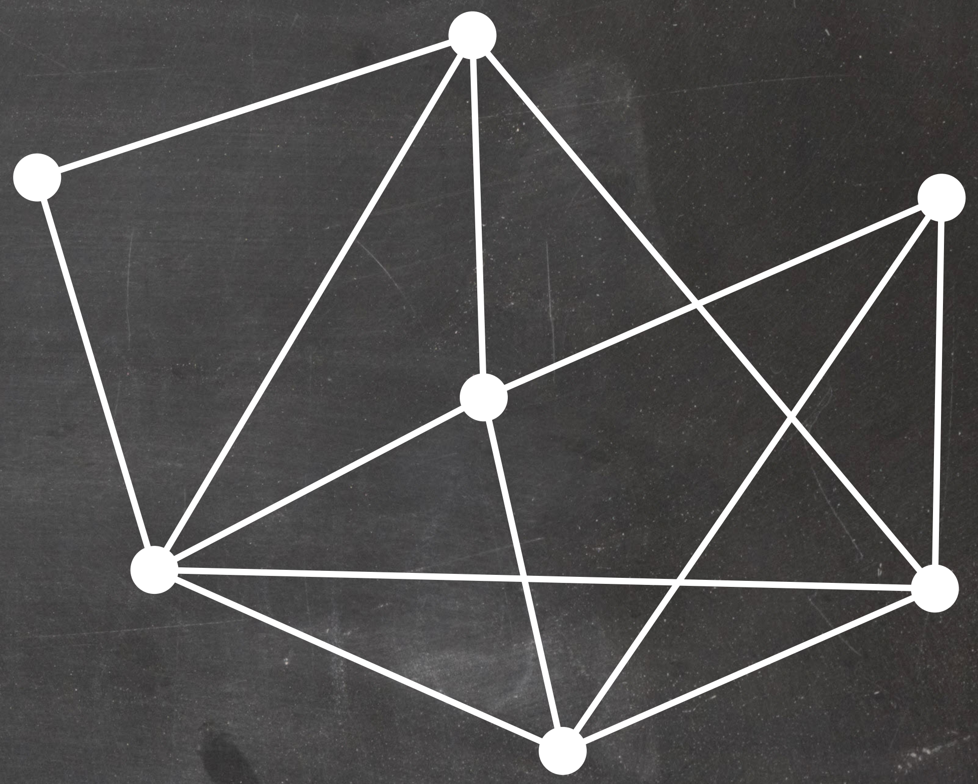
The endpoints of an edge (v, w) are v and w.

The endpoints of an edge e are said to be adjacent to each other and incident with e.

The degree of a vertex is the number of its incident edges.

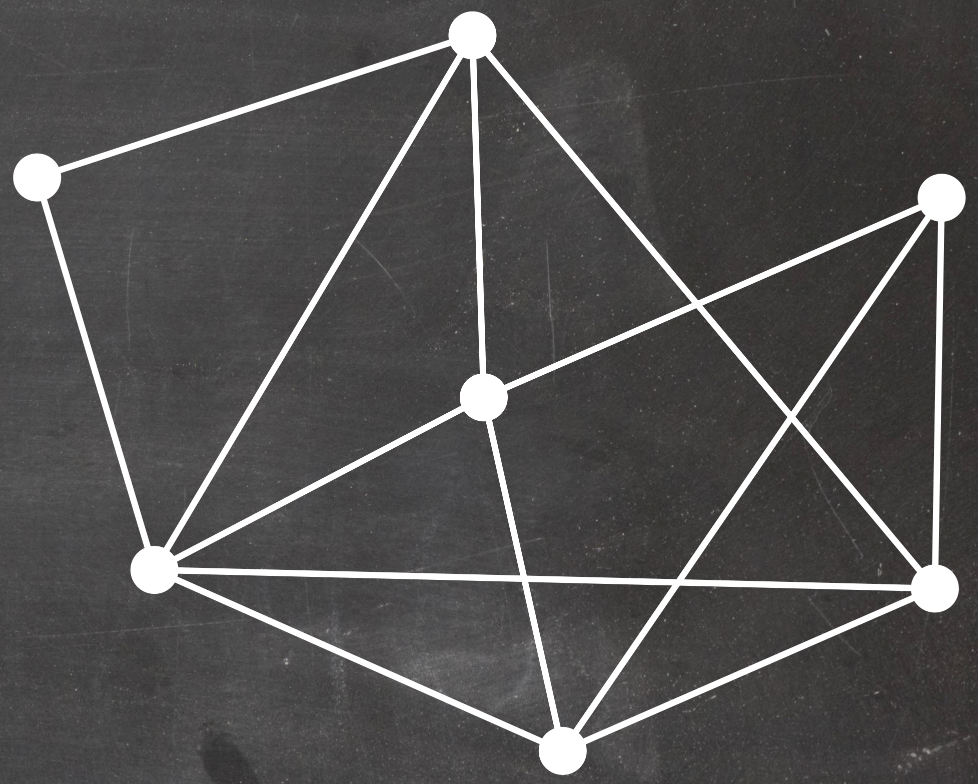
Undirected and Directed Graphs

A graph is undirected if its edges are unordered pairs, that is, $(v, w) = (w, v)$.

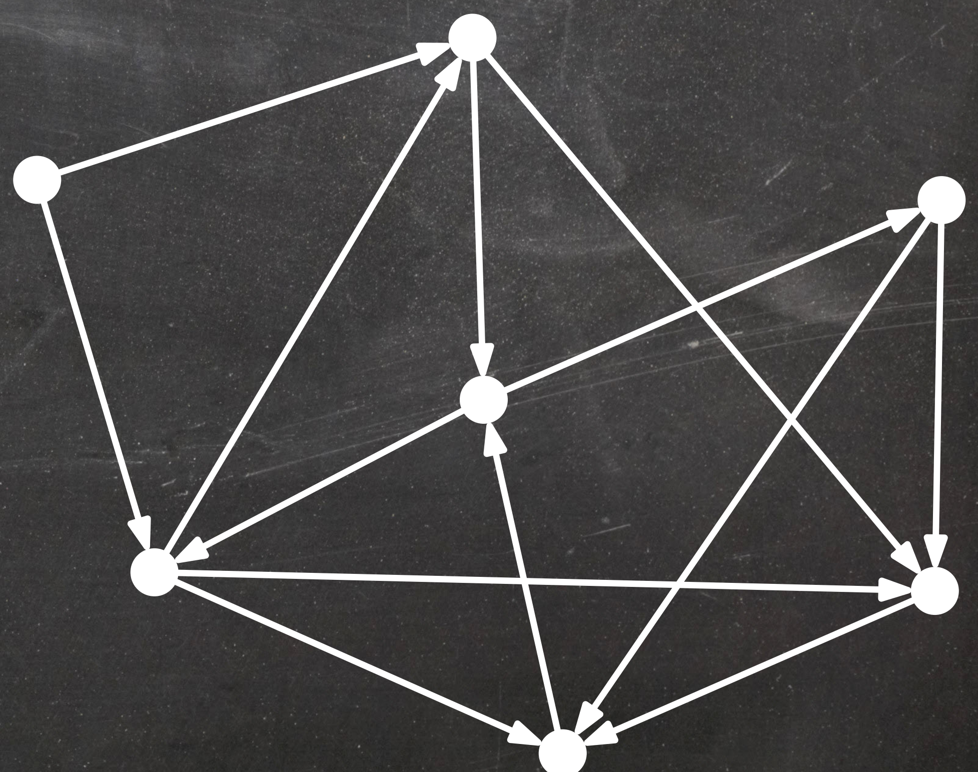


Undirected and Directed Graphs

A graph is undirected if its edges are unordered pairs, that is, $(v, w) = (w, v)$.

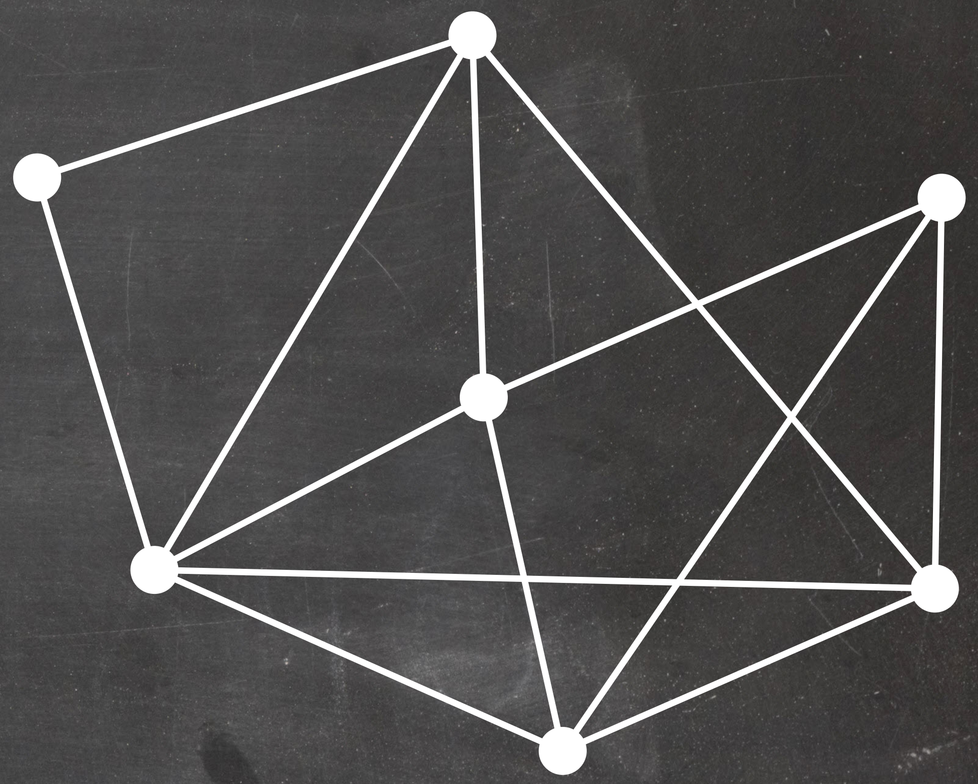


A graph is directed if its edges are ordered pairs, that is, $(v, w) \neq (w, v)$.



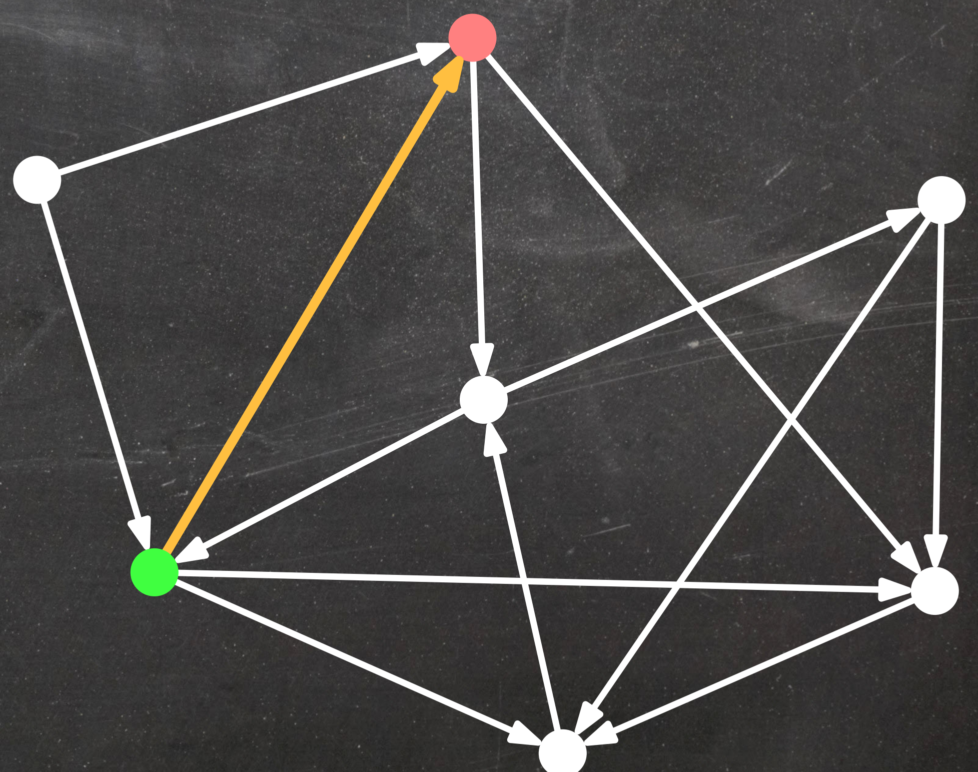
Undirected and Directed Graphs

A graph is undirected if its edges are unordered pairs, that is, (v, w) = (w, v).

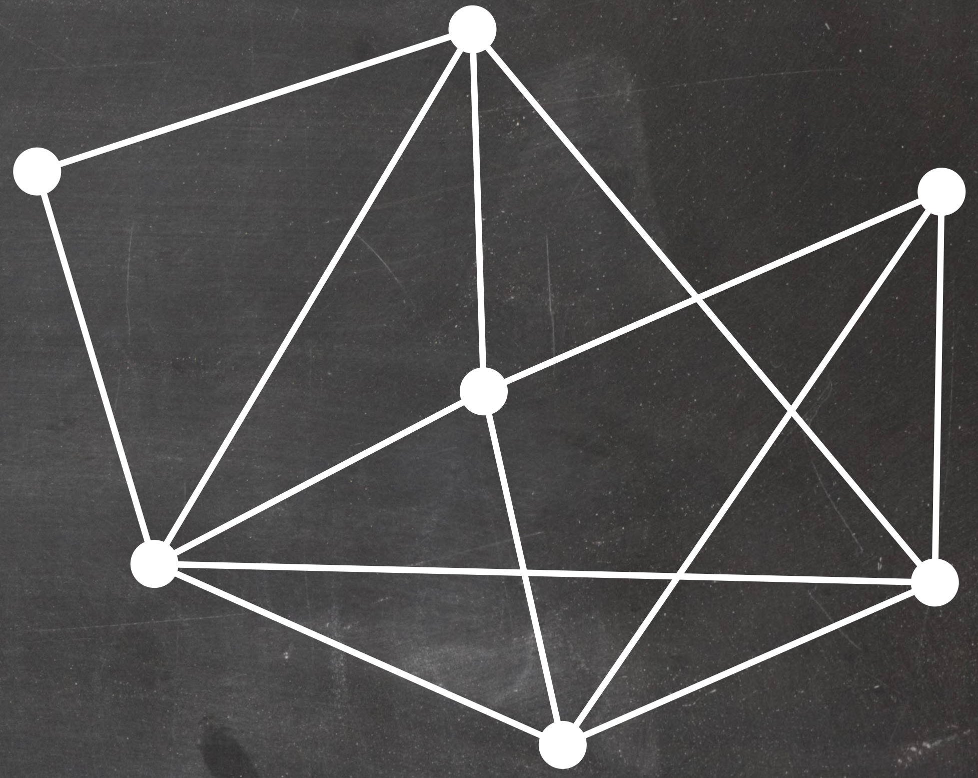


A graph is directed if its edges are ordered pairs, that is, (v, w) ≠ (w, v).

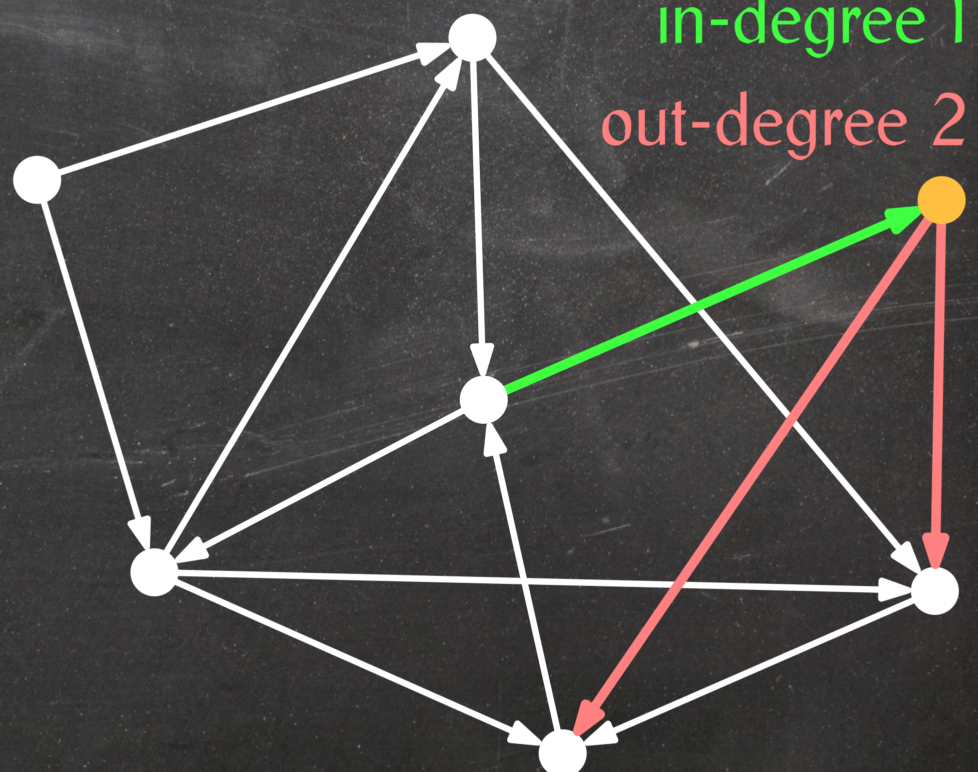
A directed edge (v, w) is an out-edge of v and an in-edge of w.



Undirected and Directed Graphs



A graph is undirected if its edges are unordered pairs, that is, $(v, w) = (w, v)$.



A graph is directed if its edges are ordered pairs, that is, $(v, w) \neq (w, v)$.

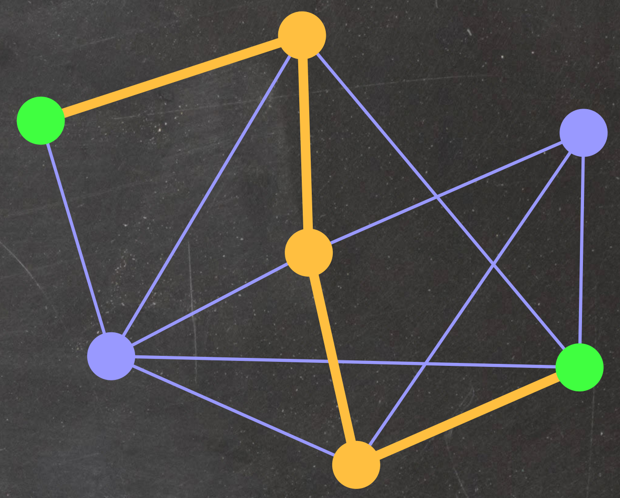
A directed edge (v, w) is an out-edge of v and an in-edge of w.

The in-degree and out-degree of a vertex are the numbers of its in-edges and out-edges, respectively.

Paths and Cycles

A path from a vertex s to a vertex t is a sequence of vertices $\langle x_0, x_1, \ldots, x_k \rangle$ such that

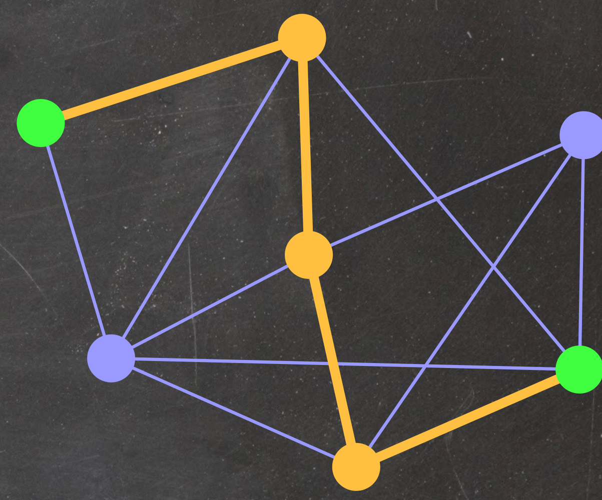
- $x_0 = s$,
- $x_k = t$, and
- for all $1 \leq i \leq k$, (x_{i-1}, x_i) is an edge of G.



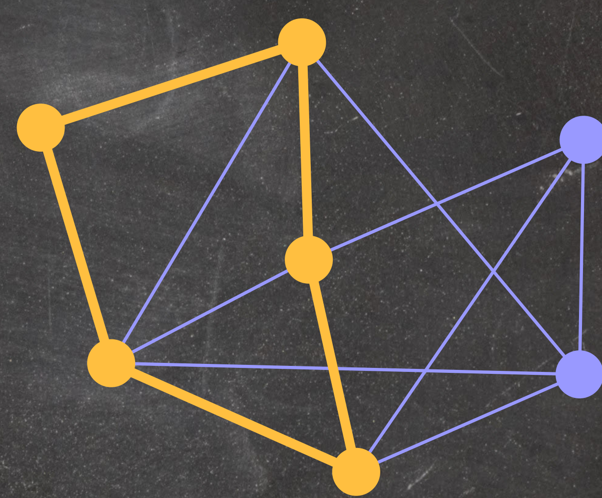
Paths and Cycles

A path from a vertex s to a vertex t is a sequence of vertices $\langle x_0, x_1, \ldots, x_k \rangle$ such that

- $x_0 = s$,
- $x_k = t$, and
- for all $1 \leq i \leq k$, (x_{i-1}, x_i) is an edge of G.



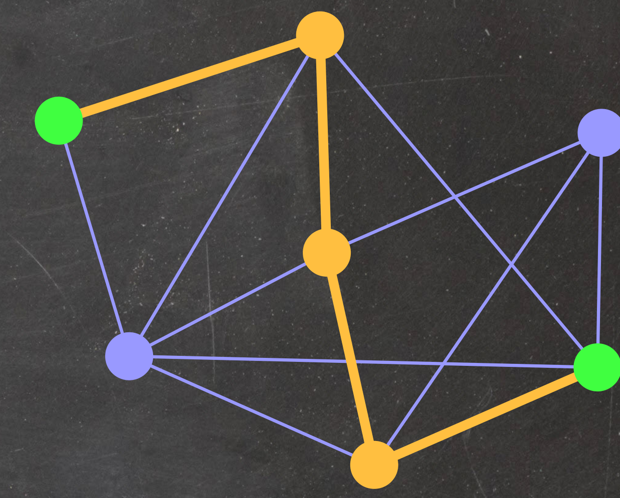
A cycle is a path from a vertex x back to itself.



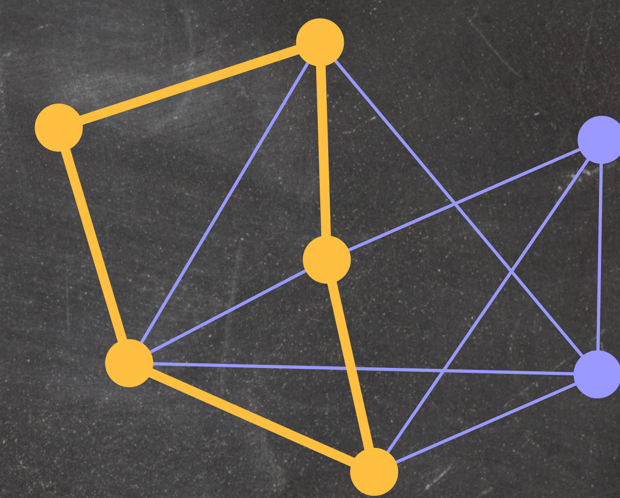
Paths and Cycles

A path from a vertex s to a vertex t is a sequence of vertices $\langle x_0, x_1, \ldots, x_k \rangle$ such that

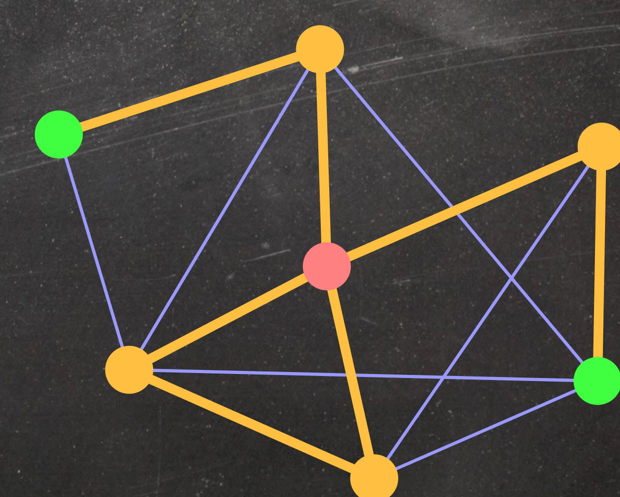
- $x_0 = s$,
- $x_k = t$, and
- for all $1 \leq i \leq k$, (x_{i-1}, x_i) is an edge of G.



A cycle is a path from a vertex x back to itself.

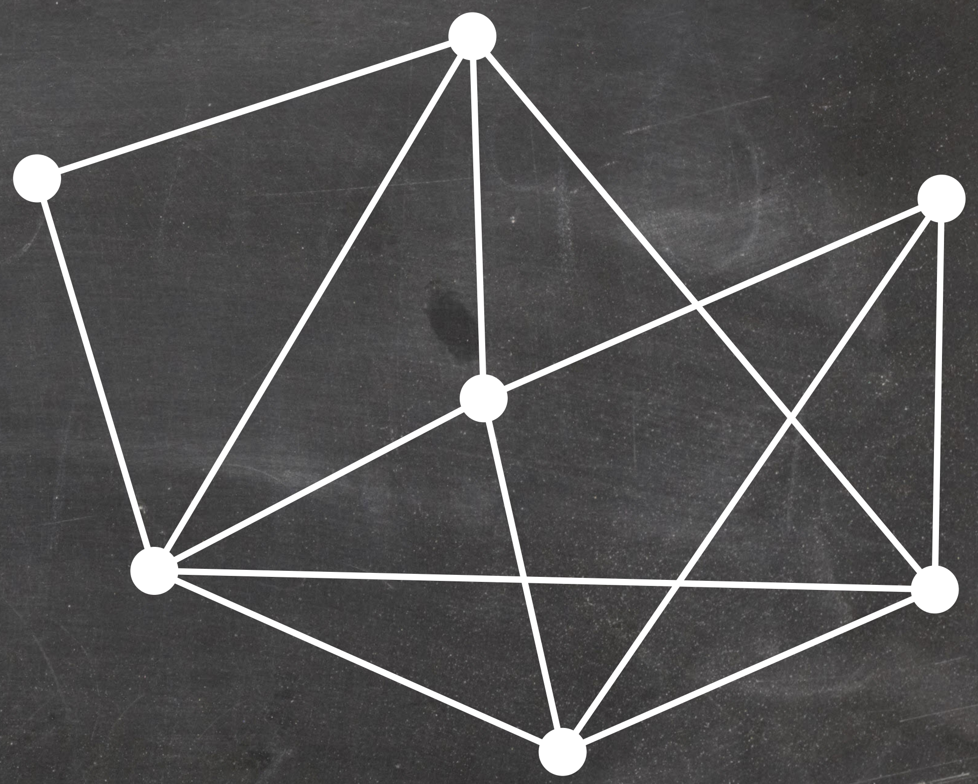


A path or cycle is simple if it contains every vertex of G at most once.



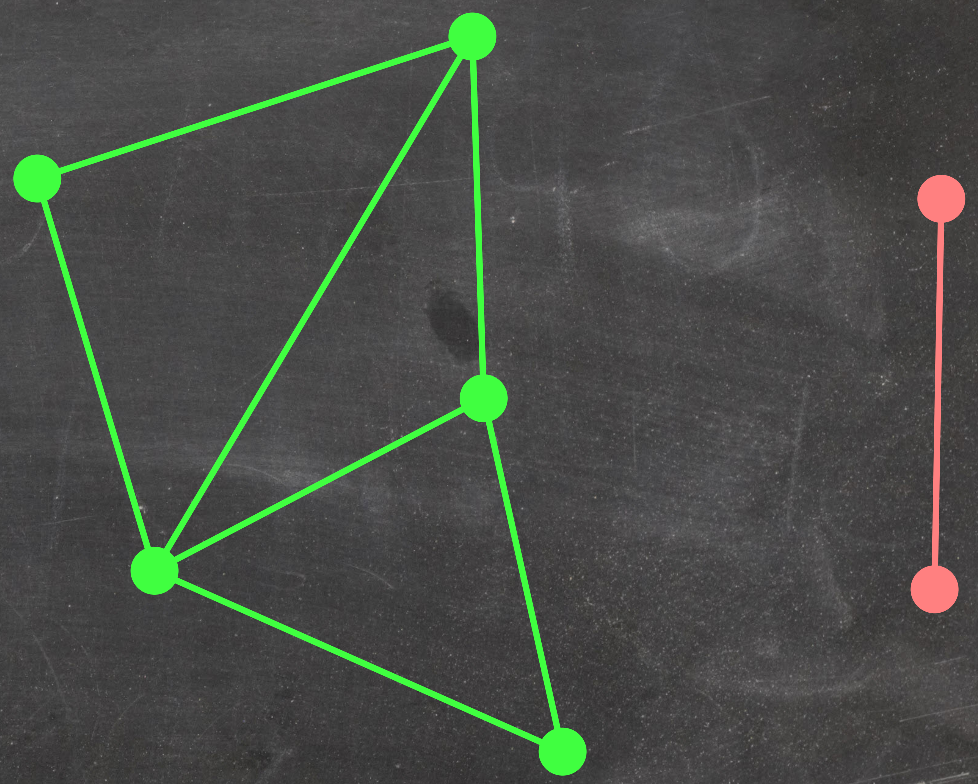
Connected Graphs, Trees, and Forests

A graph is connected if there exists a path between every pair of vertices.



Connected Graphs, Trees, and Forests

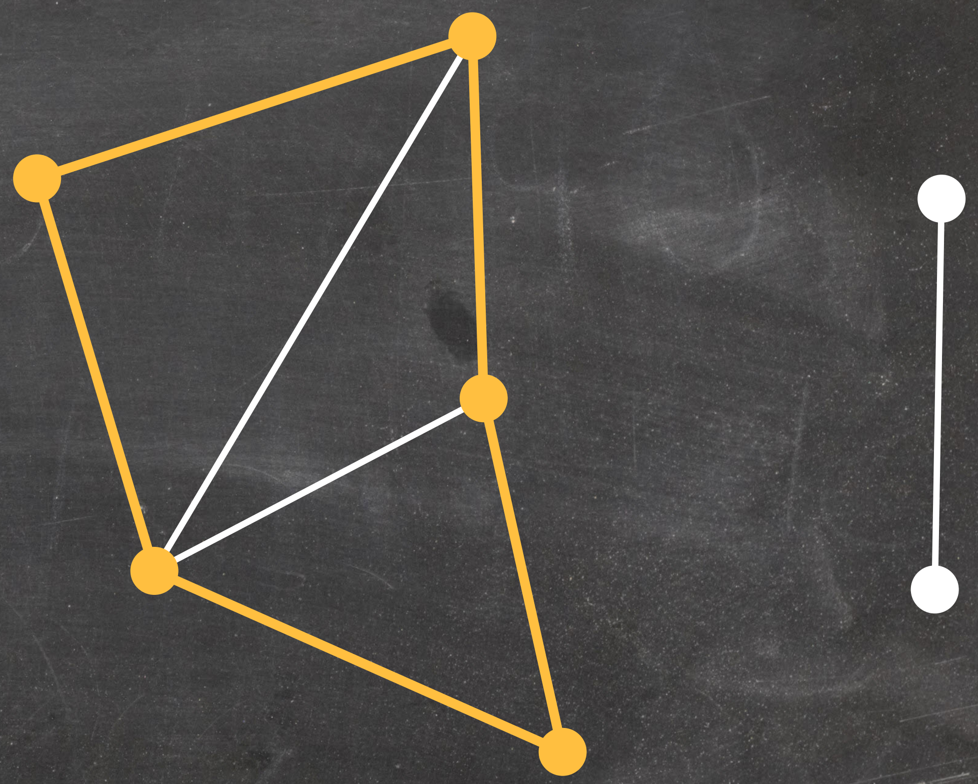
A graph is connected if there exists a path between every pair of vertices.



Connected Graphs, Trees, and Forests

A graph is connected if there exists a path between every pair of vertices.

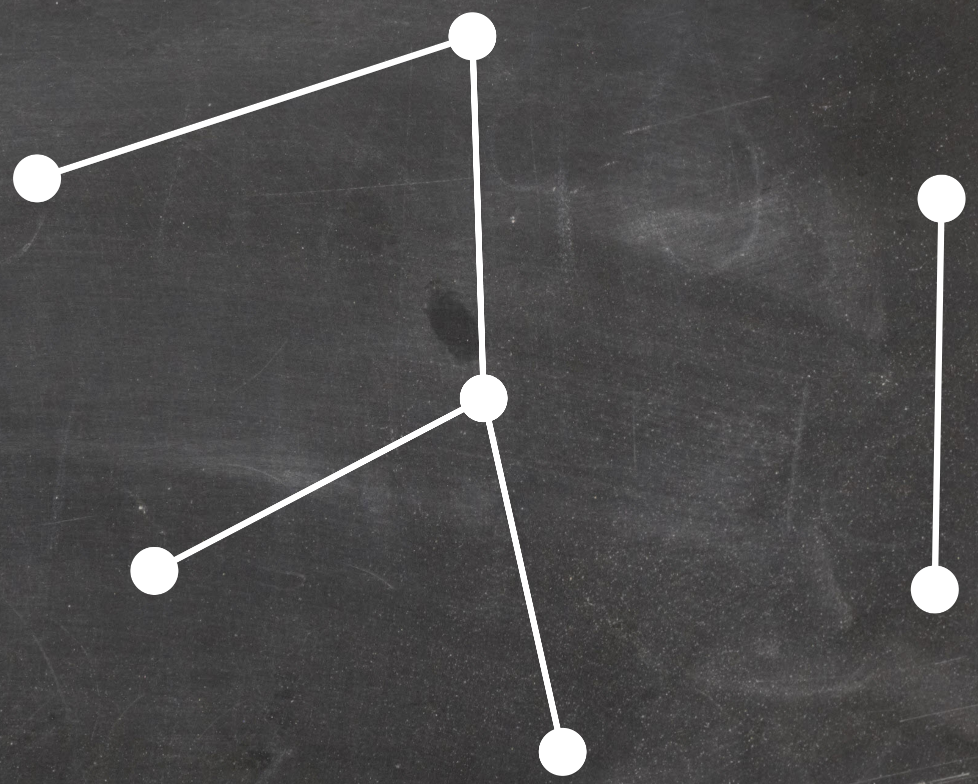
A forest is a graph without cycles.



Connected Graphs, Trees, and Forests

A graph is connected if there exists a path between every pair of vertices.

A forest is a graph without cycles.

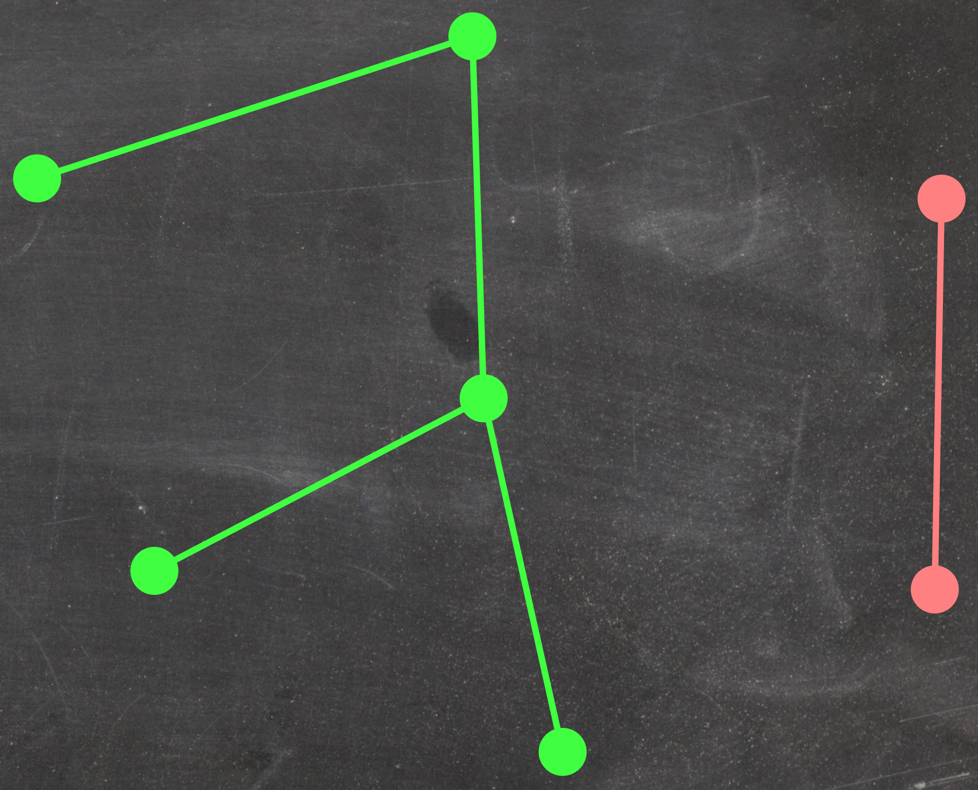


Connected Graphs, Trees, and Forests

A graph is connected if there exists a path between every pair of vertices.

A forest is a graph without cycles.

A tree is a connected forest.

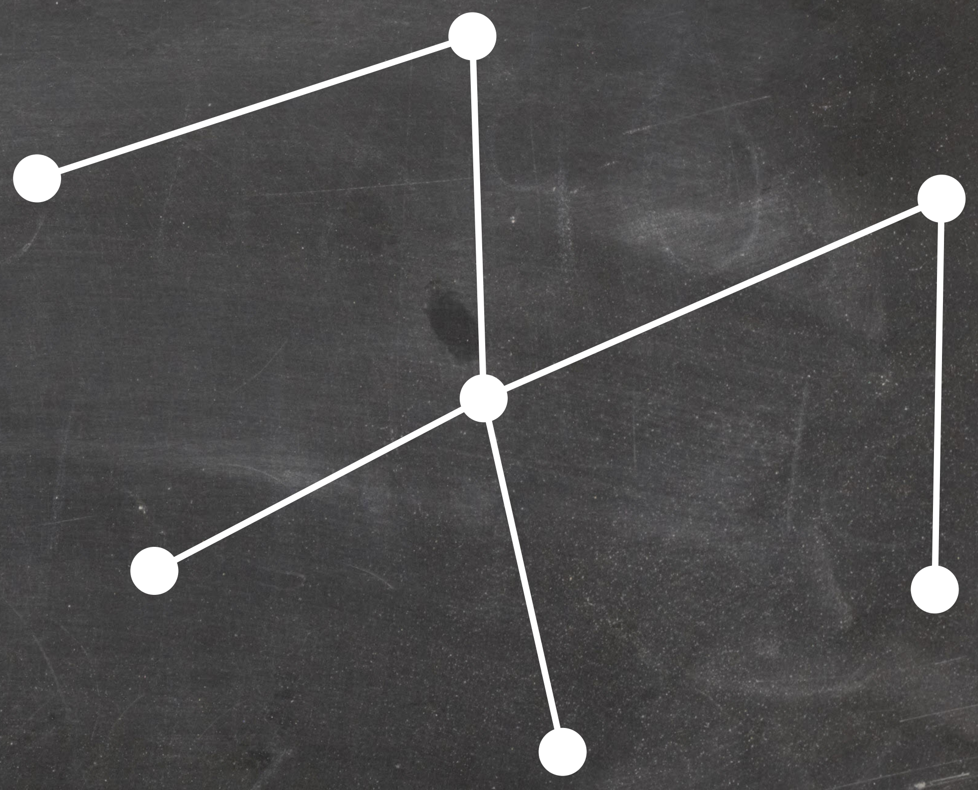


Connected Graphs, Trees, and Forests

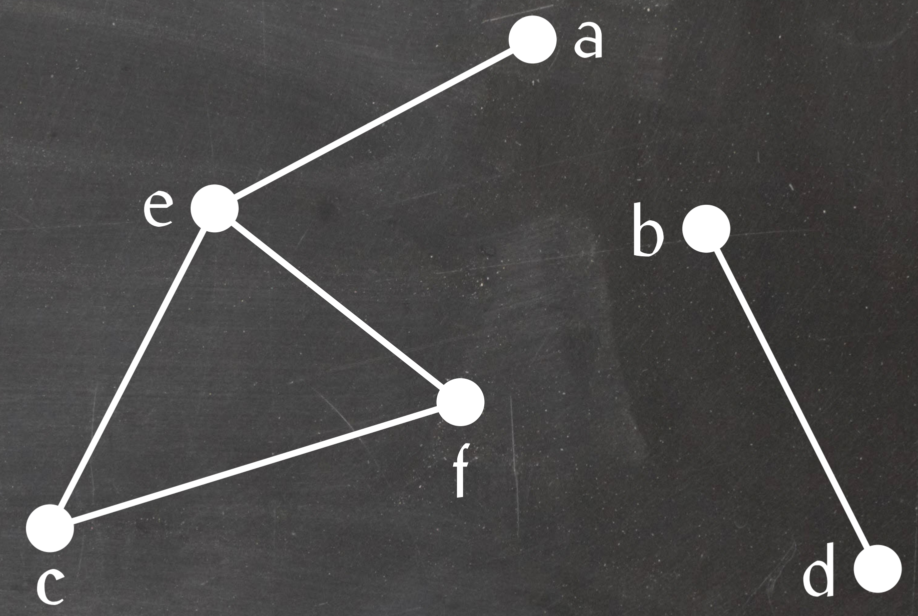
A graph is connected if there exists a path between every pair of vertices.

A forest is a graph without cycles.

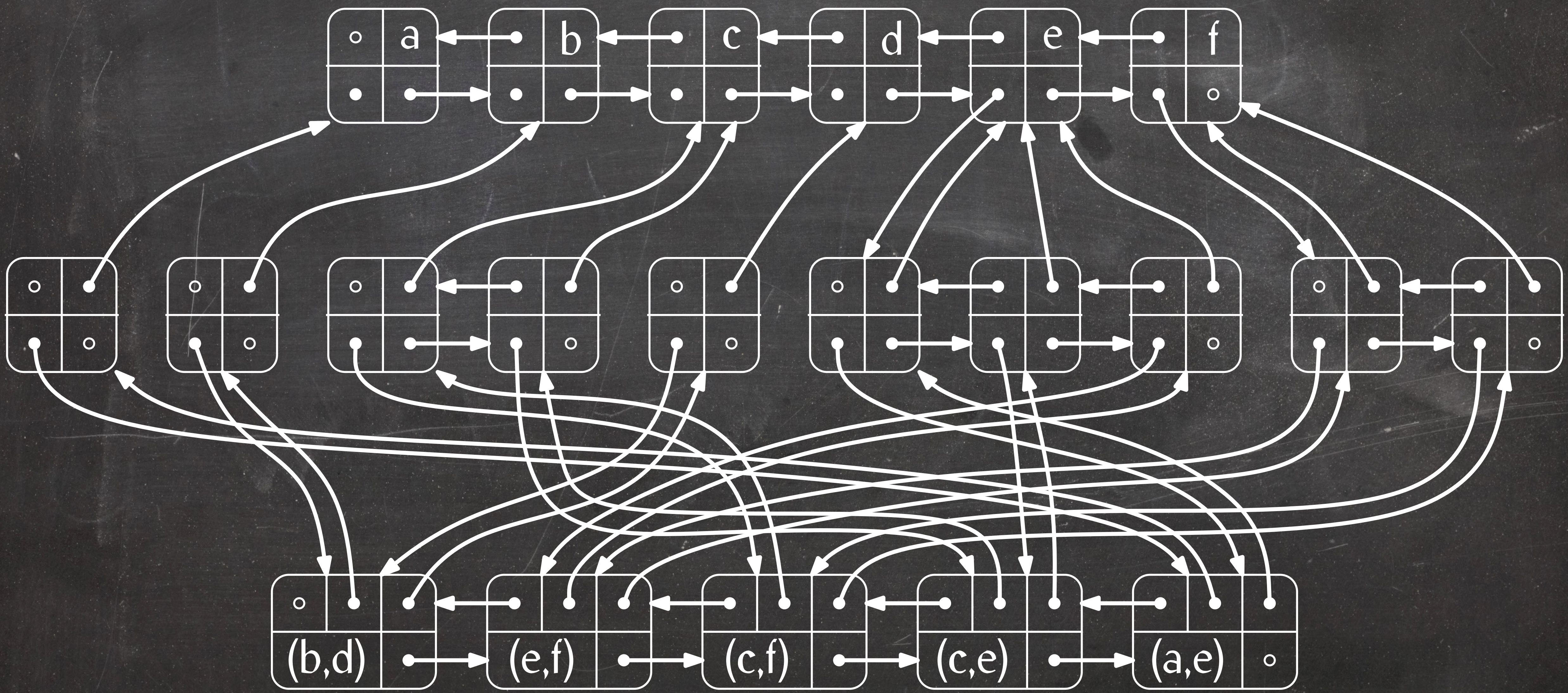
A tree is a connected forest.



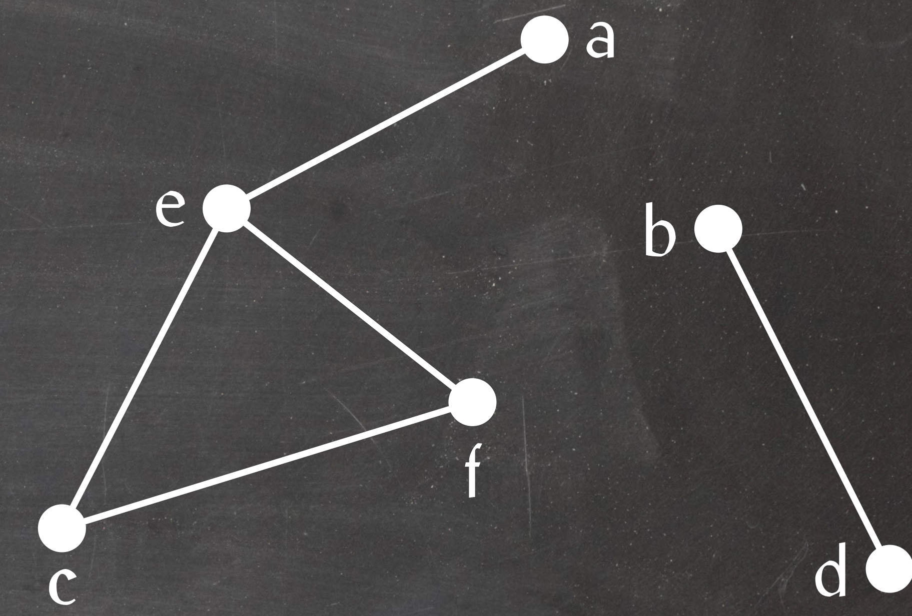
Adjacency List Representation



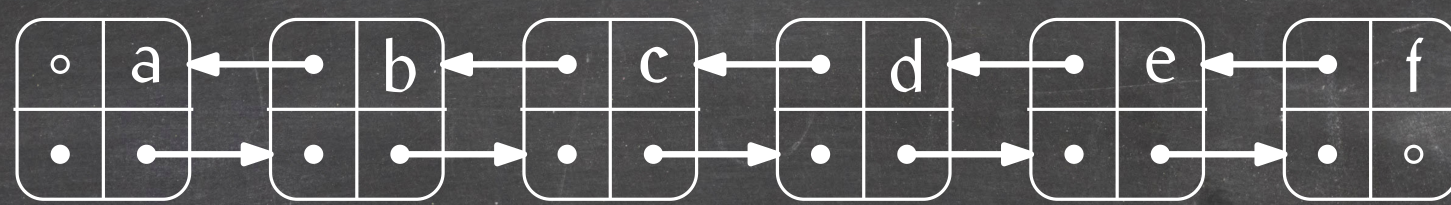
- Doubly-linked list of vertices
- Doubly-linked list of edges
- One doubly-linked adjacency list per vertex
- Pointers from adjacency list entries to vertices
- Cross-pointers between edges and adjacency list entries

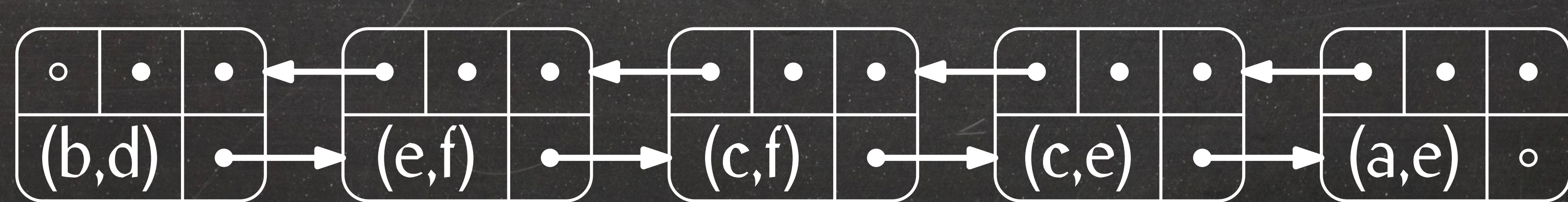


Adjacency List Representation



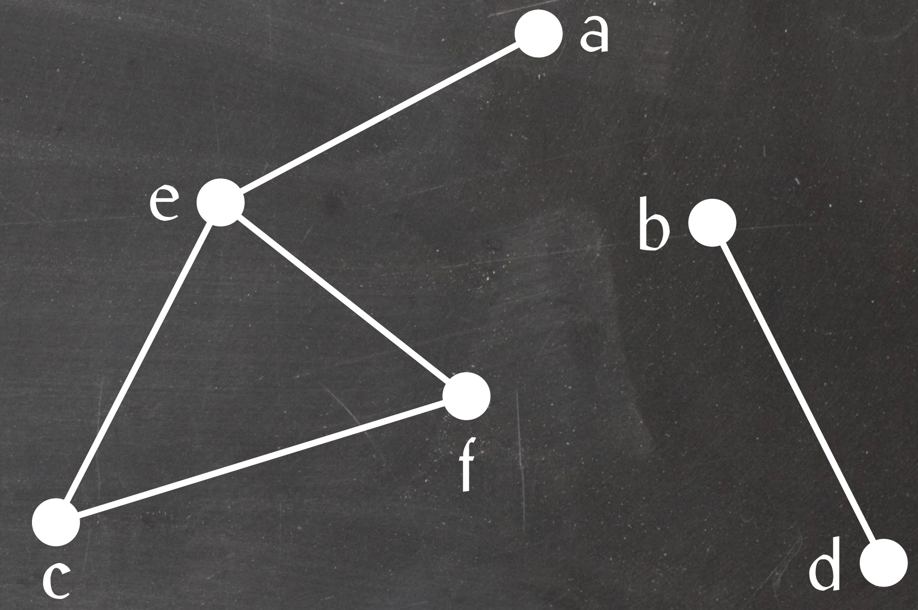
- Doubly-linked list of vertices
- Doubly-linked list of edges
- One doubly-linked adjacency list per vertex
- Pointers from adjacency list entries to vertices
- Cross-pointers between edges and adjacency list entries

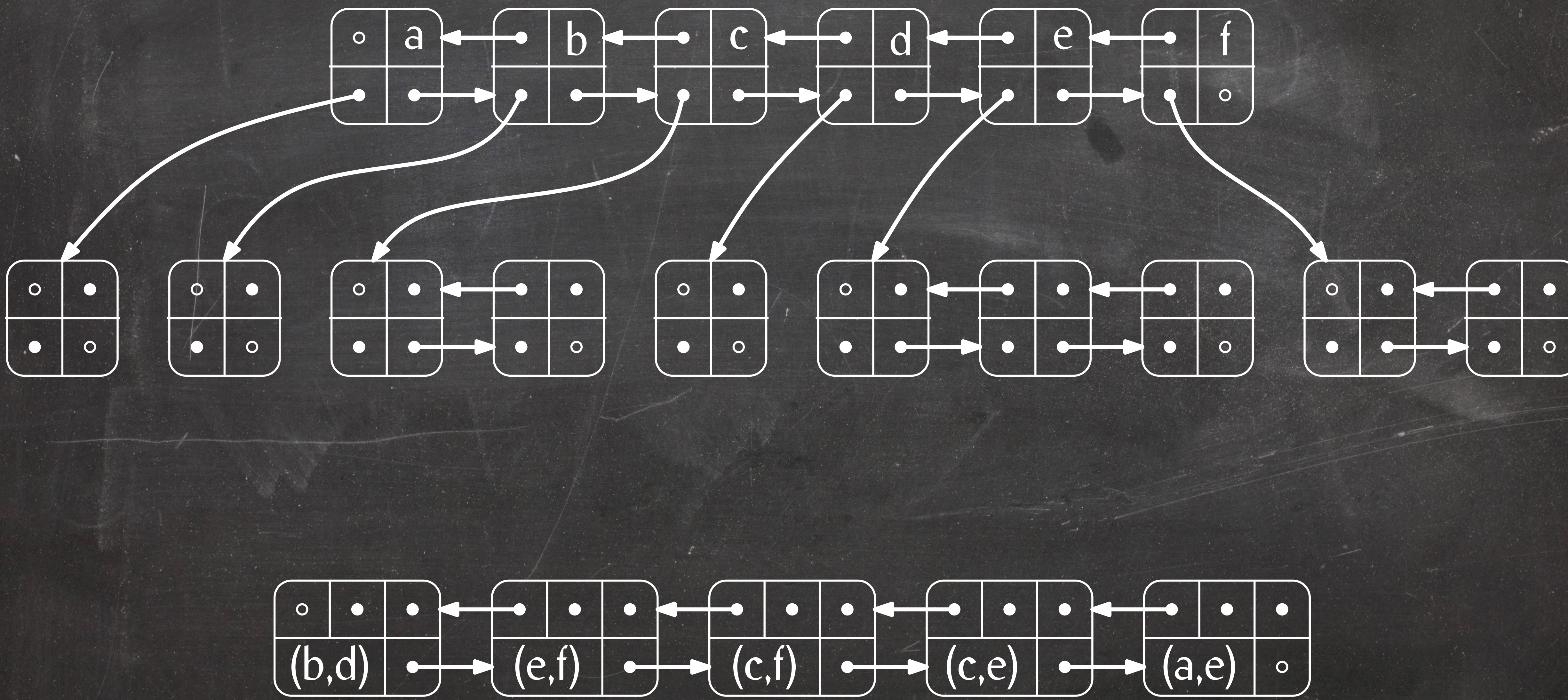




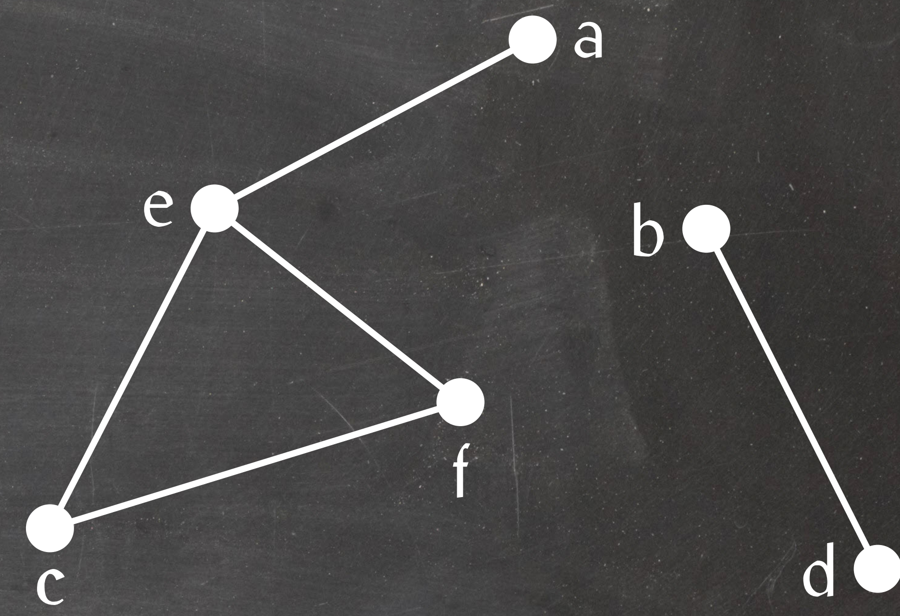
Adjacency List Representation

- Doubly-linked list of vertices
- Doubly-linked list of edges
- One doubly-linked adjacency list per vertex
- Pointers from adjacency list entries to vertices
- Cross-pointers between edges and adjacency list entries

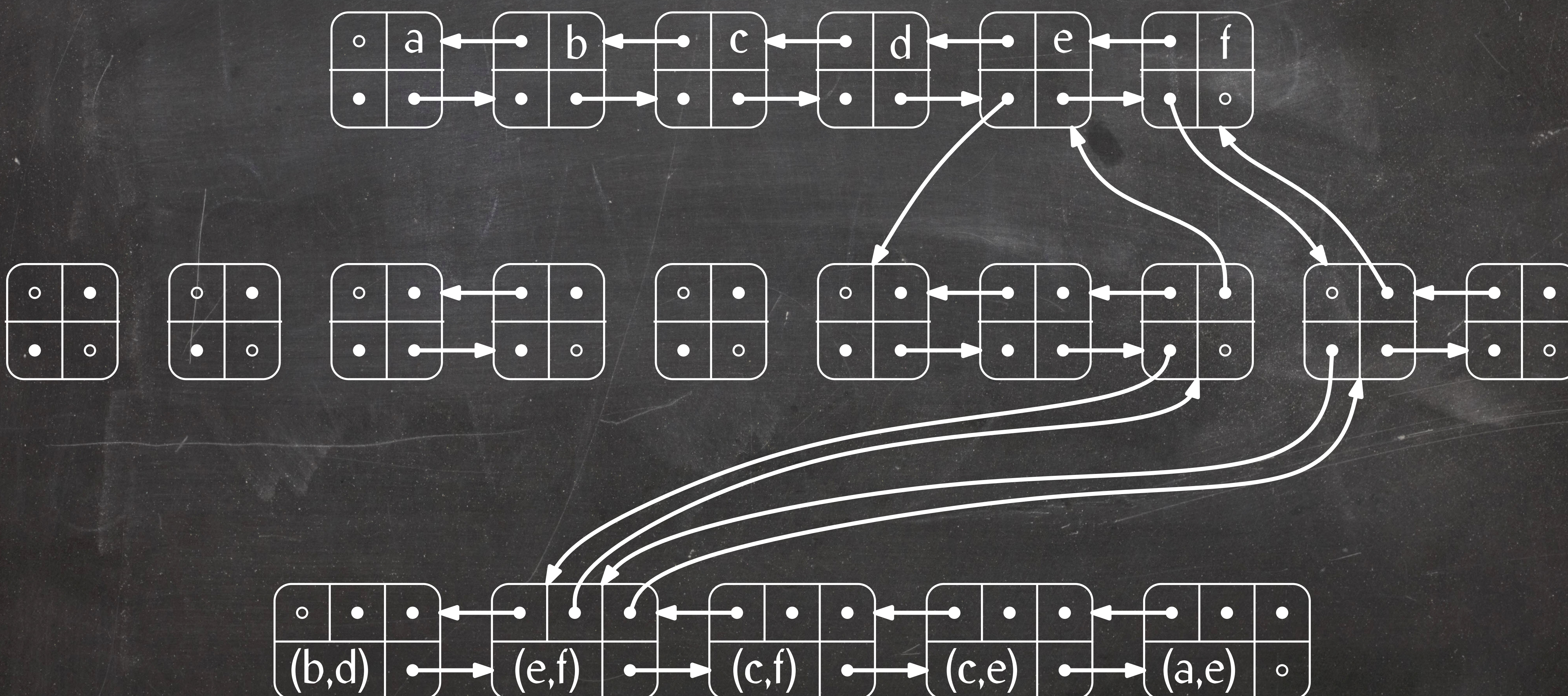




Adjacency List Representation



- Doubly-linked list of vertices
- Doubly-linked list of edges
- One doubly-linked adjacency list per vertex
- Pointers from adjacency list entries to vertices
- Cross-pointers between edges and adjacency list entries

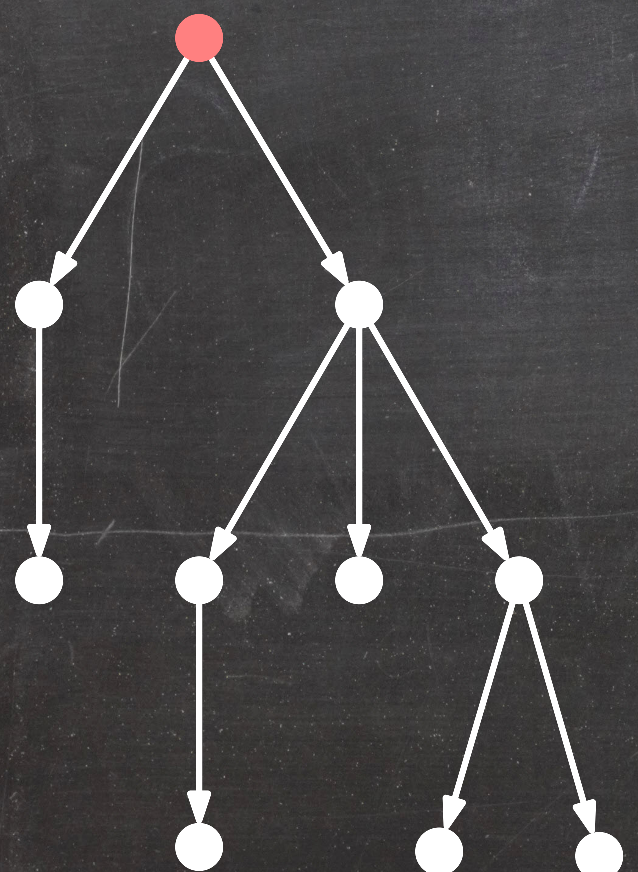


Representing Rooted Trees

A rooted tree T

- is a tree,
- is a directed graph,
- has one of its vertices, r, designated as a root.

There exists a path from r to every vertex in T.

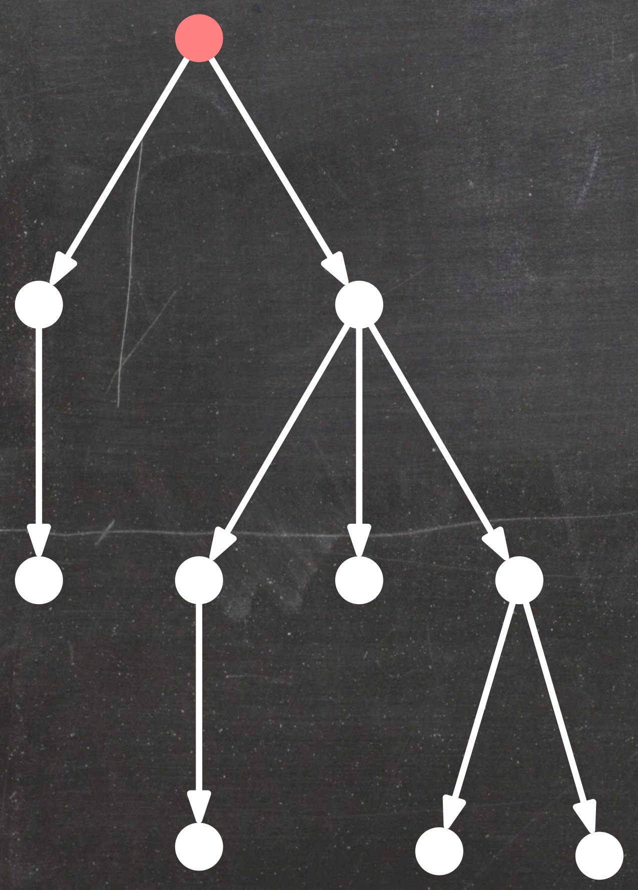


Representing Rooted Trees

A rooted tree T

- is a tree,
- is a directed graph,
- has one of its vertices, r, designated as a root.

There exists a path from r to every vertex in T.



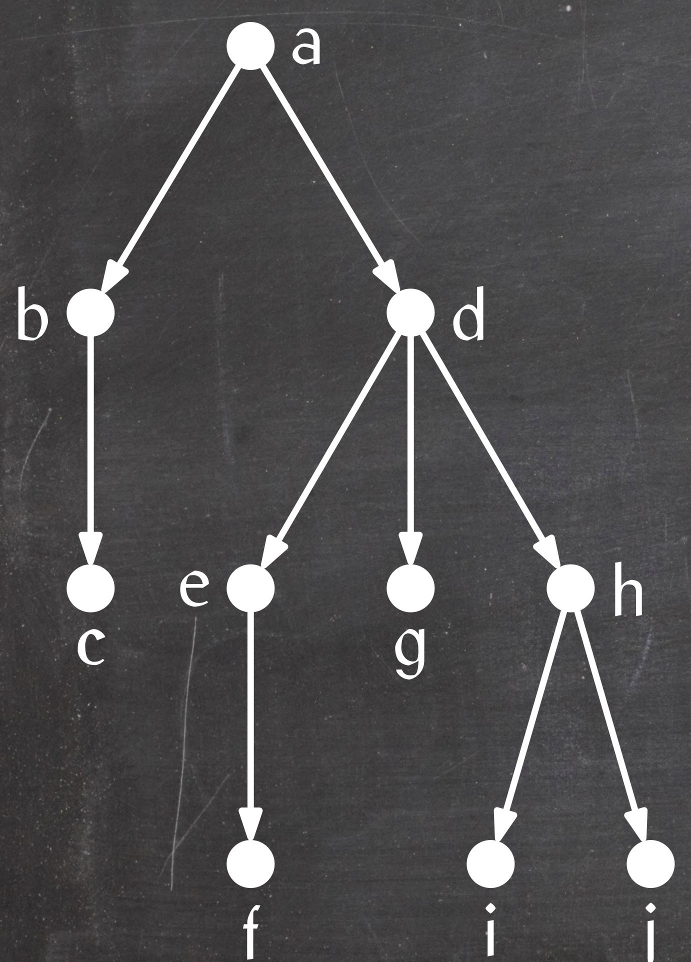
Representation:

Tree = root

Every node stores

- an arbitrary key
- a (doubly-linked) list of its children.

Standard Tree Orderings

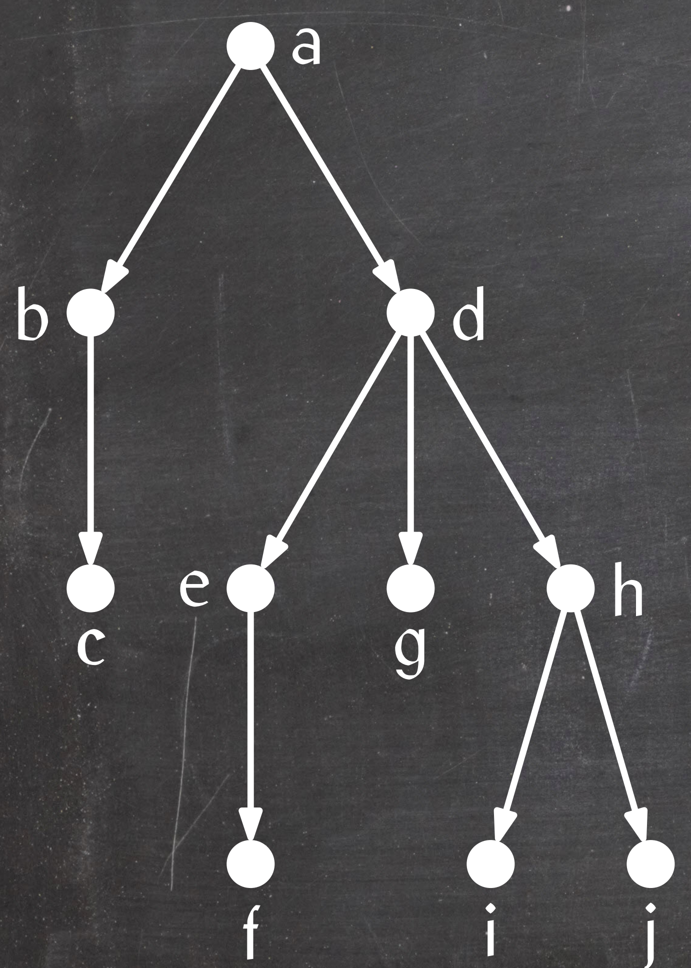


Preorder:

- Every vertex appears before its children.
- Every vertex appears before its right sibling.
- The vertices in each subtree appear consecutively.

$\Rightarrow$ [a, b, c, d, e, f, g, h, i, j]

Standard Tree Orderings



Preorder:

- Every vertex appears before its children.
- Every vertex appears before its right sibling.
- The vertices in each subtree appear consecutively.

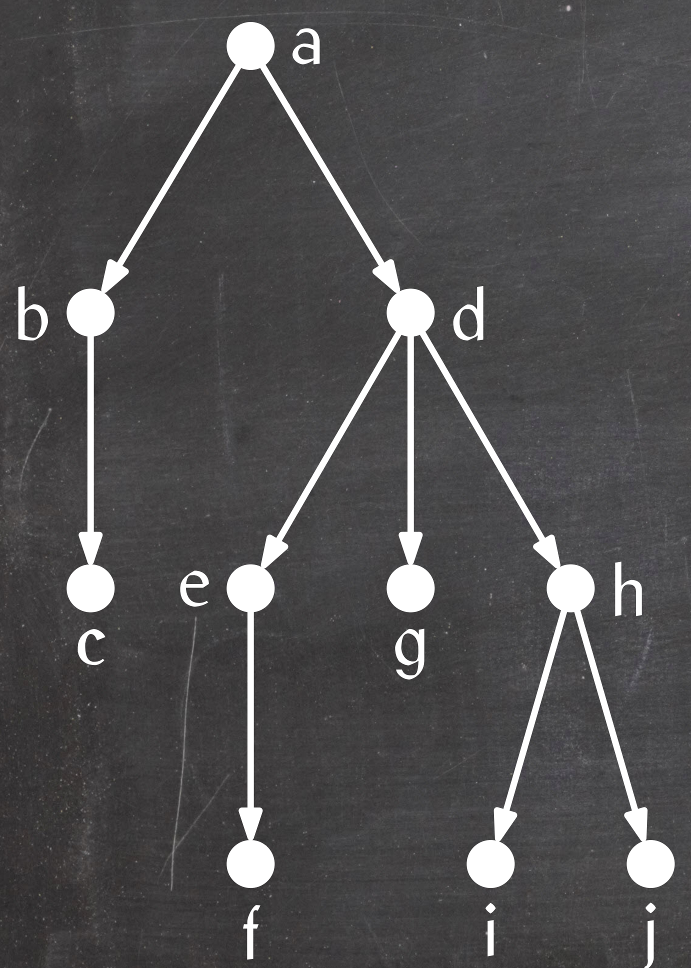
$\Rightarrow$ [a, b, c, d, e, f, g, h, i, j]

Postorder:

- Every vertex appears after its children.
- Every vertex appears before its right sibling.
- The vertices in each subtree appear consecutively.

$\Rightarrow$ [c, b, f, e, g, i, j, h, d, a]

Standard Tree Orderings



Preorder:

- Every vertex appears before its children.
- Every vertex appears before its right sibling.
- The vertices in each subtree appear consecutively.

$\Rightarrow$ [a, b, c, d, e, f, g, h, i, j]

Postorder:

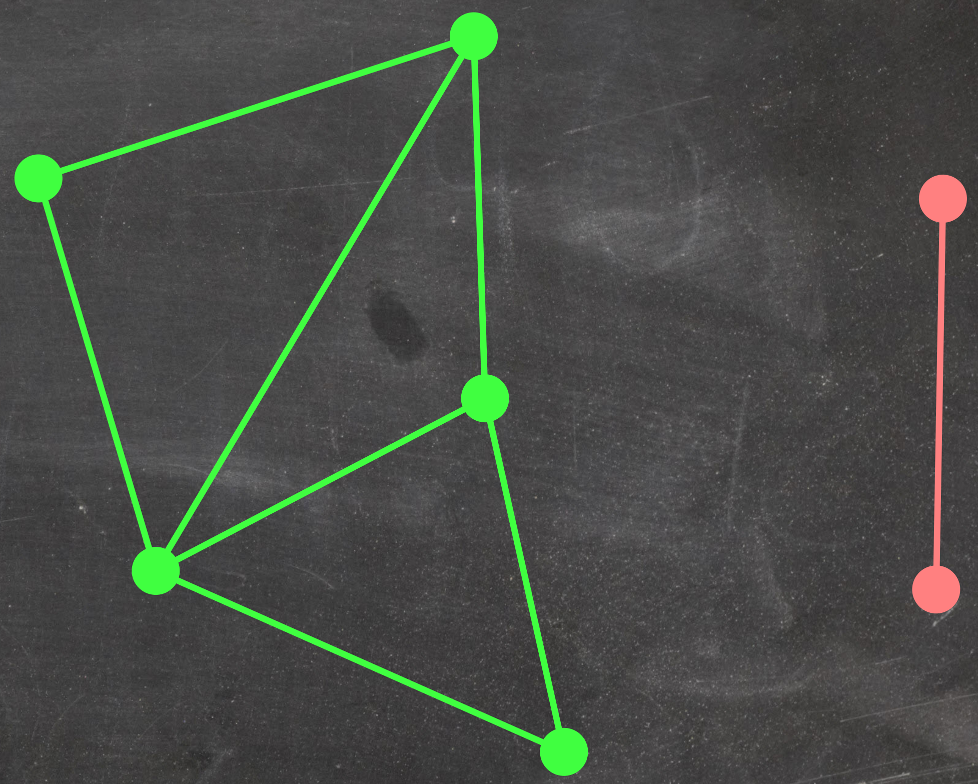
- Every vertex appears after its children.
- Every vertex appears before its right sibling.
- The vertices in each subtree appear consecutively.

$\Rightarrow$ [c, b, f, e, g, i, j, h, d, a]

Lemma: It takes linear time to arrange the vertices of a forest in preorder or postorder.

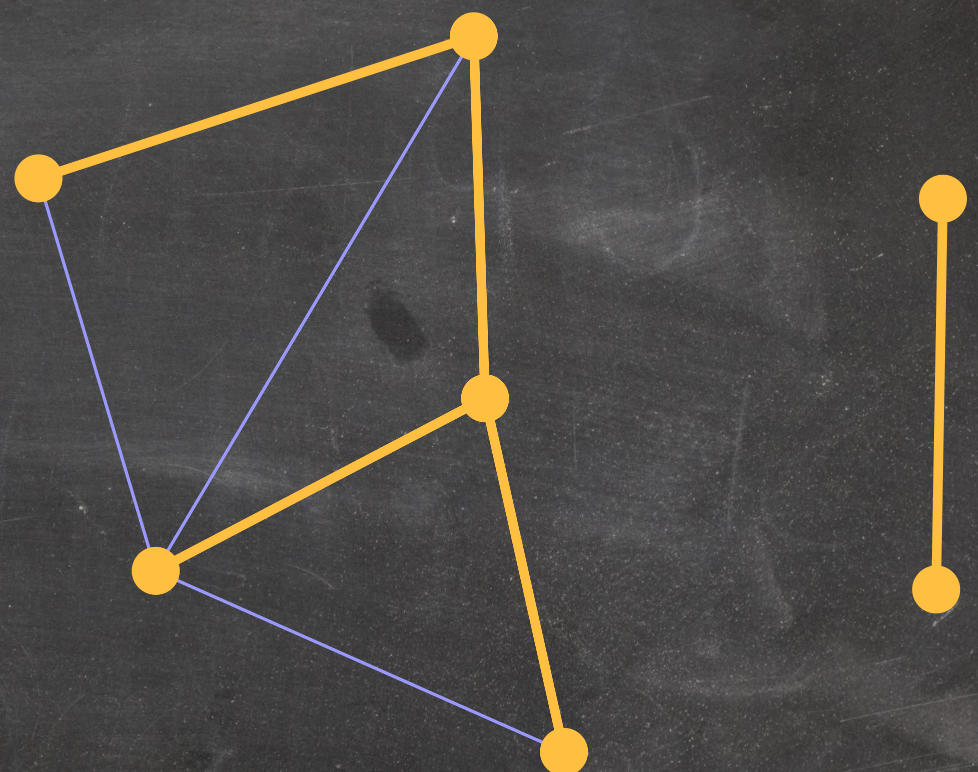
Connected Components and Spanning Forests

The connected components of a graph G are its maximal connected subgraphs.



Connected Components and Spanning Forests

The connected components of a graph G are its maximal connected subgraphs.



A spanning forest of a graph G is a subgraph $F \subseteq G$ with the same number of connected components and which is a forest.

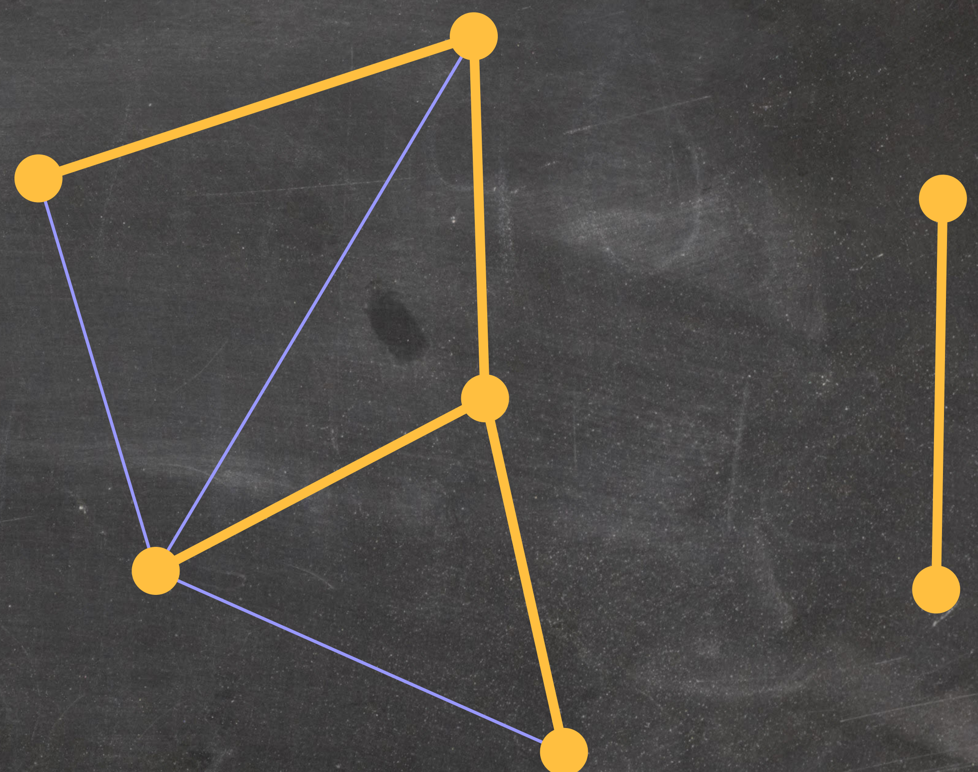
Connected Components and Spanning Forests

The connected components of a graph G are its maximal connected subgraphs.

Representation:

- List of graphs or
- Labelling of vertices with component IDs

A spanning forest of a graph G is a subgraph $F \subseteq G$ with the same number of connected components and which is a forest.



Connected Components and Spanning Forests

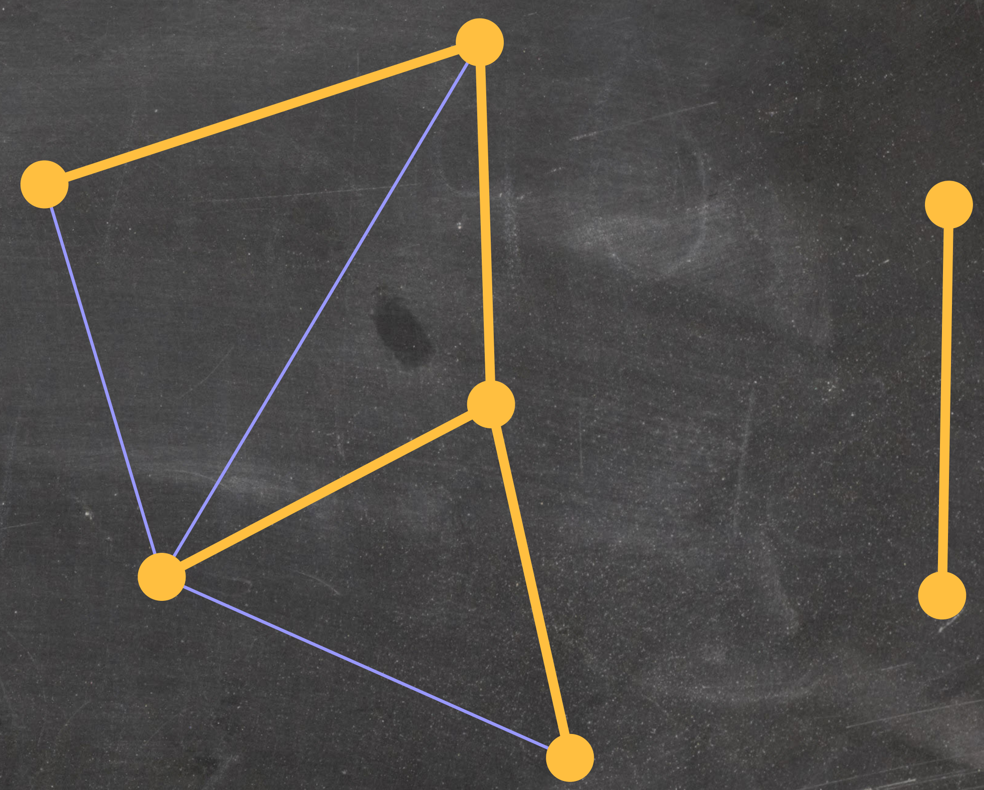
The connected components of a graph G are its maximal connected subgraphs.

Representation:

- List of graphs or
- Labelling of vertices with component IDs

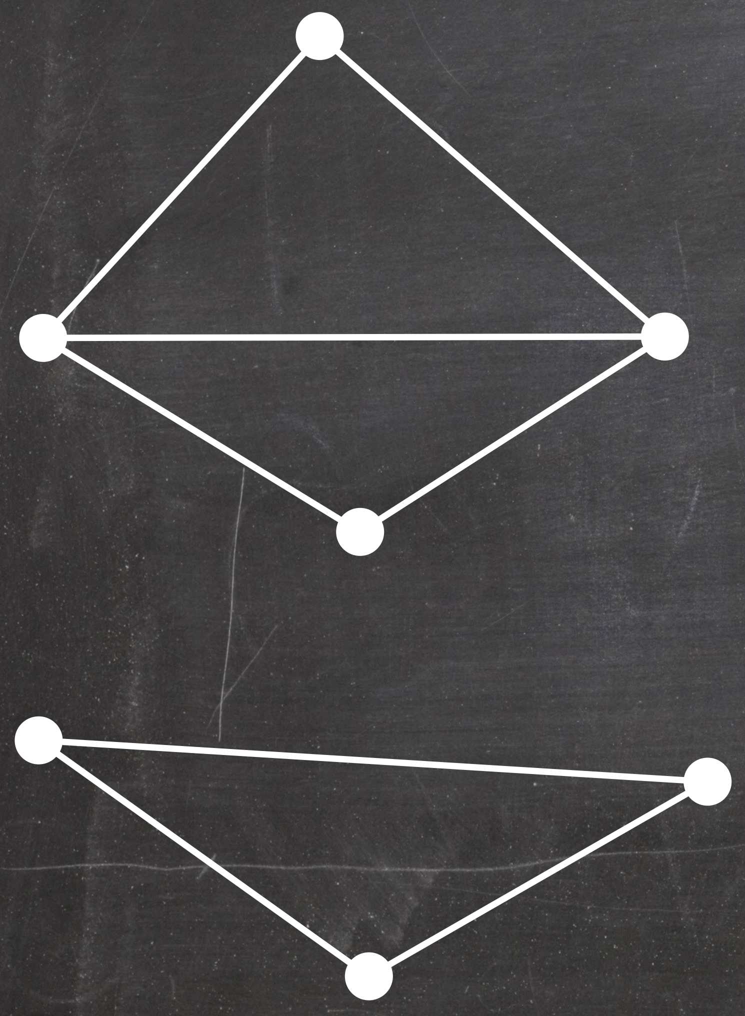
A spanning forest of a graph G is a subgraph $F \subseteq G$ with the same number of connected components and which is a forest.

Representation: List of rooted trees



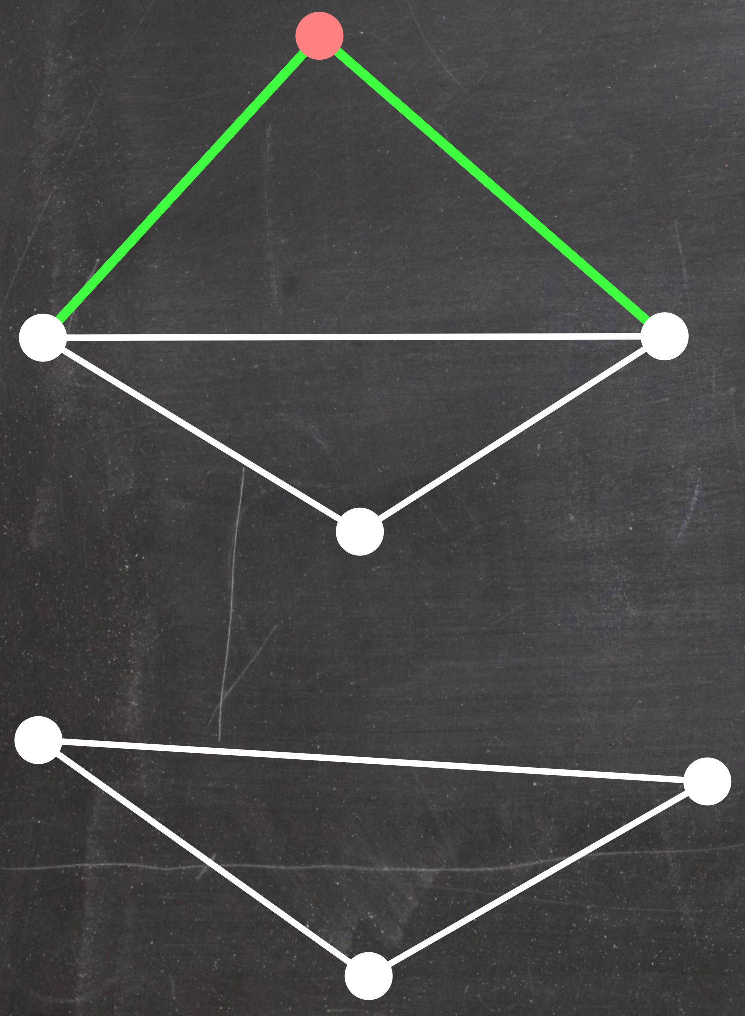
Graph Traversal

We use graph traversal to build a spanning forest of G.



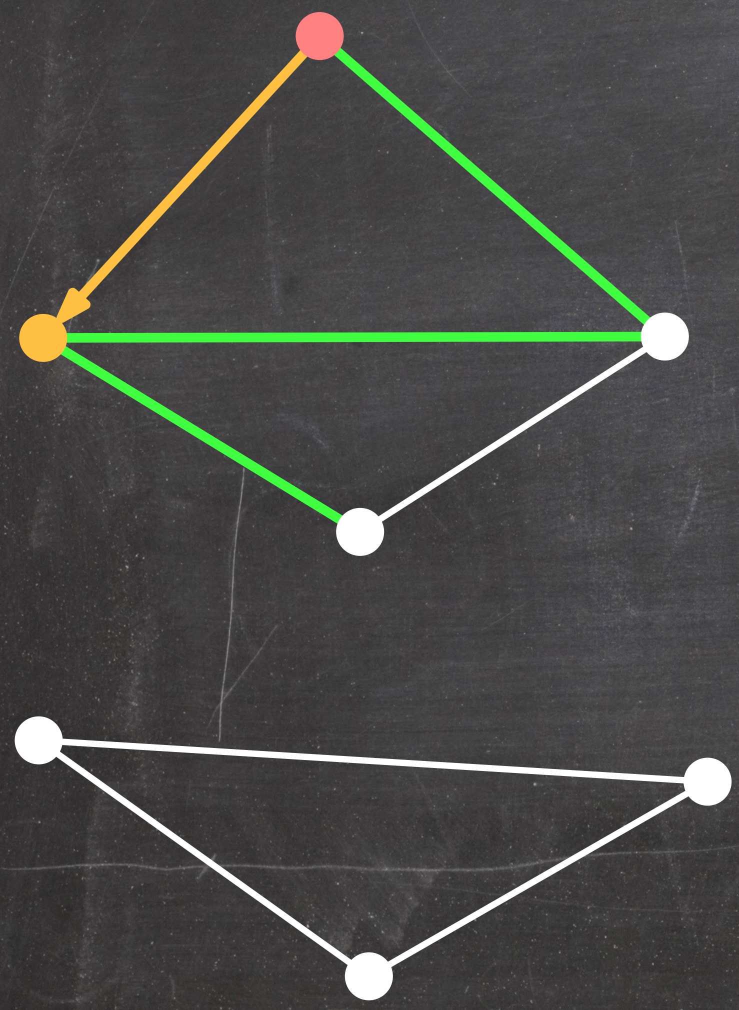
Graph Traversal

We use graph traversal to build a spanning forest of G.



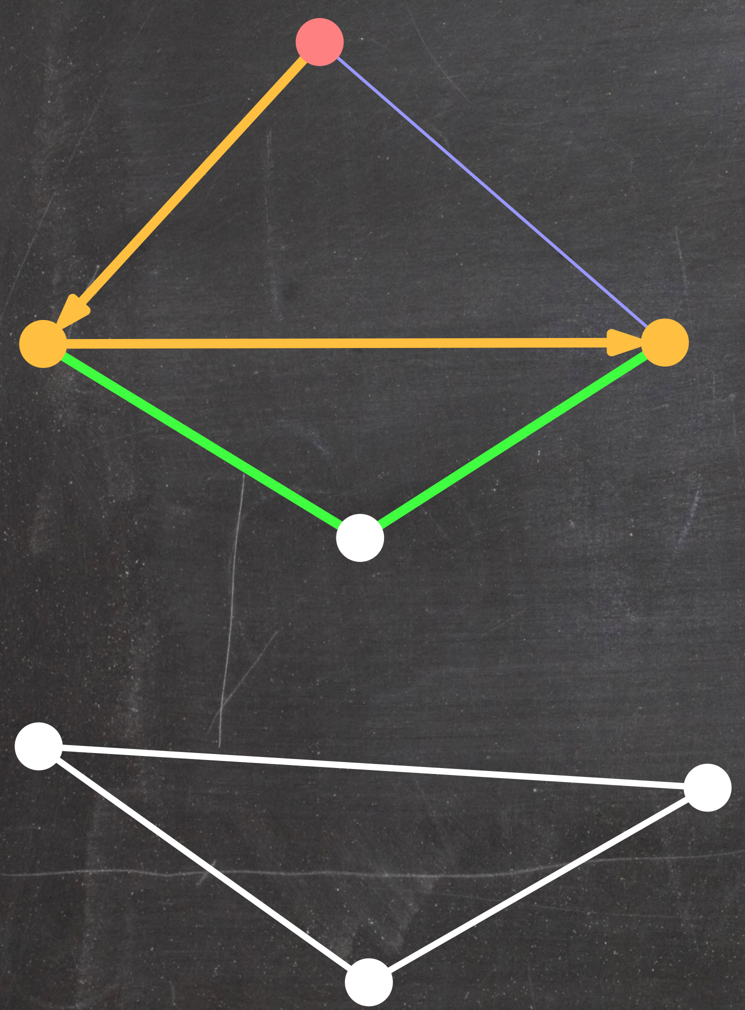
Graph Traversal

We use graph traversal to build a spanning forest of G.



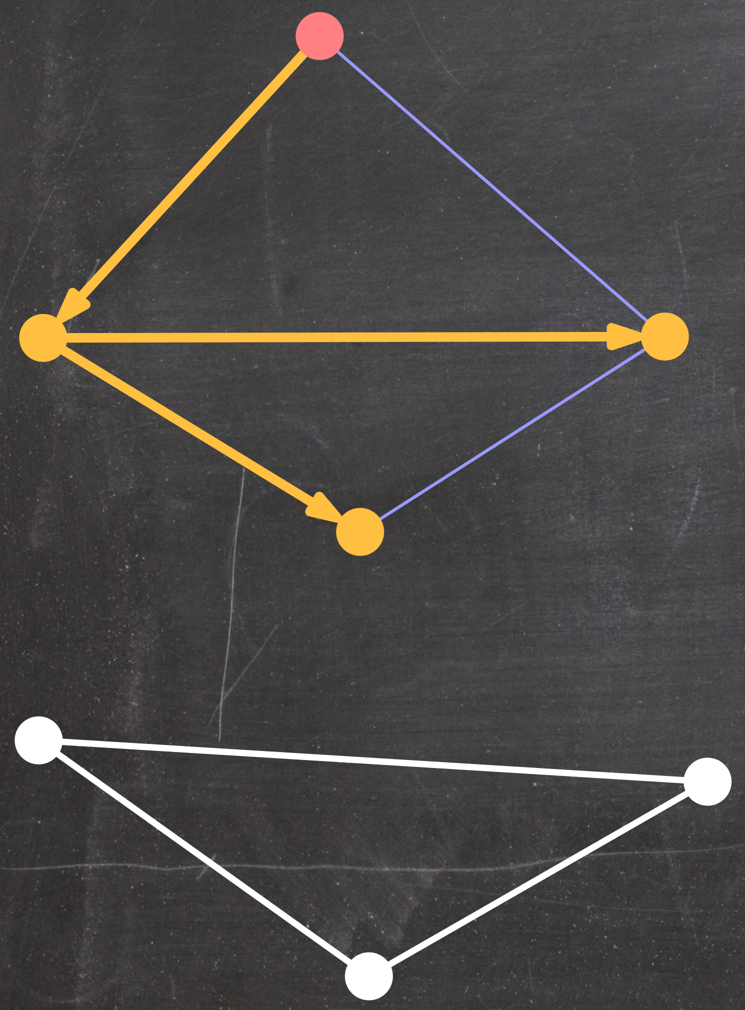
Graph Traversal

We use graph traversal to build a spanning forest of G.



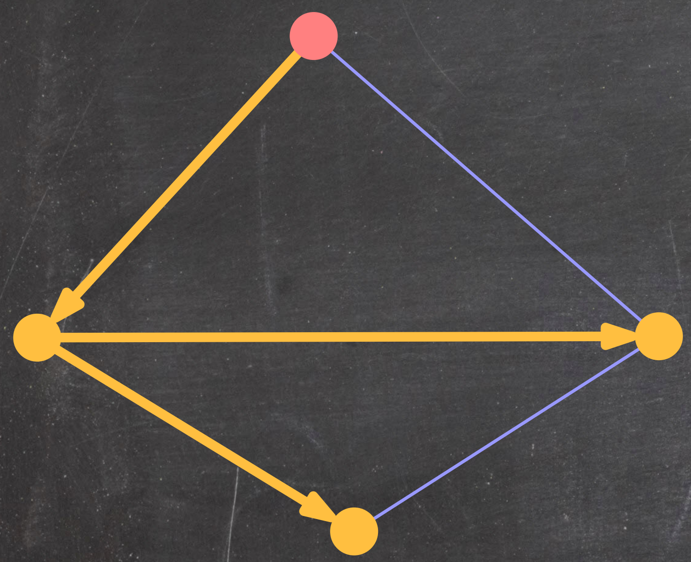
Graph Traversal

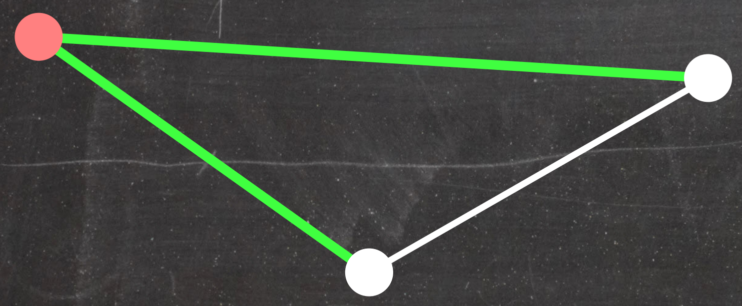
We use graph traversal to build a spanning forest of G.



Graph Traversal

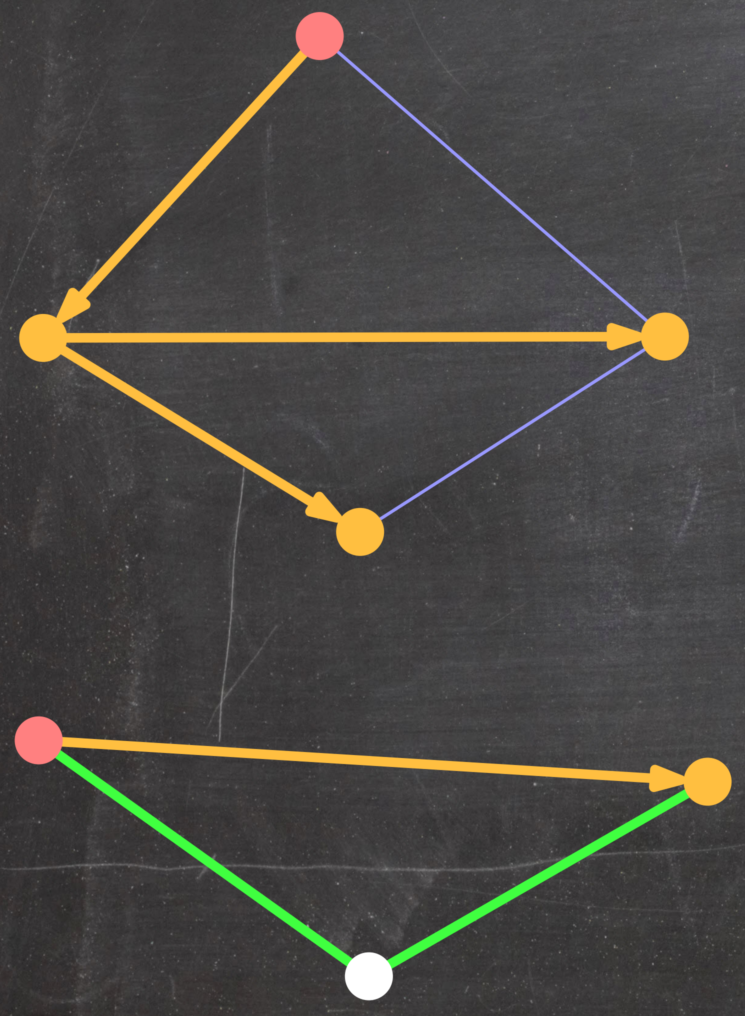
We use graph traversal to build a spanning forest of G.





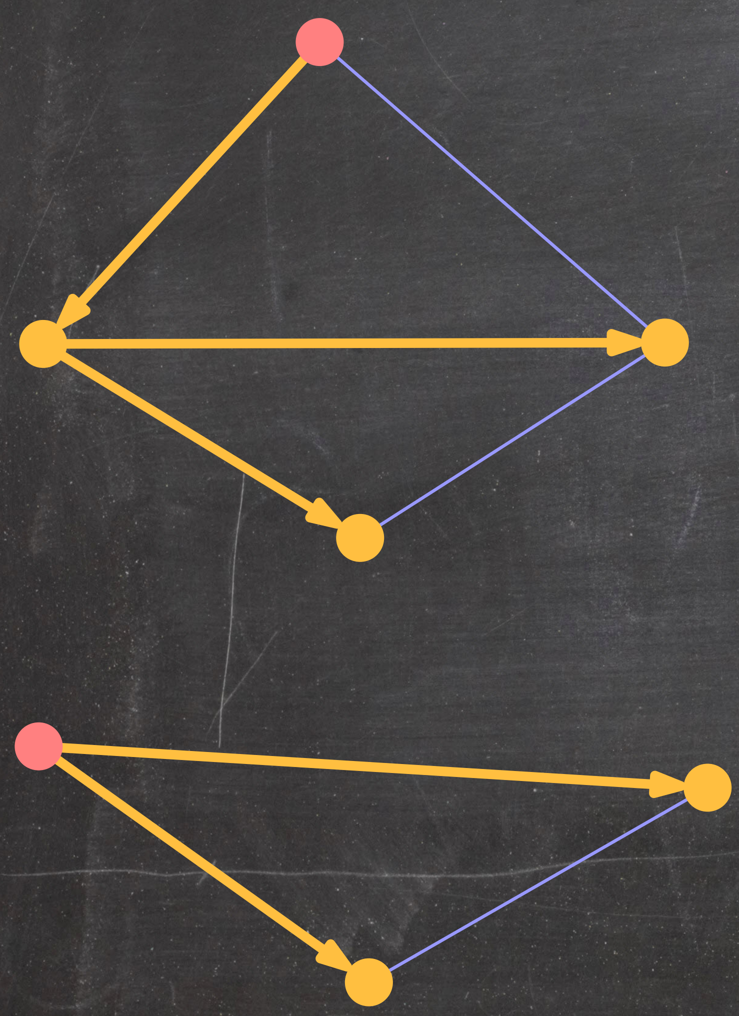
Graph Traversal

We use graph traversal to build a spanning forest of G.



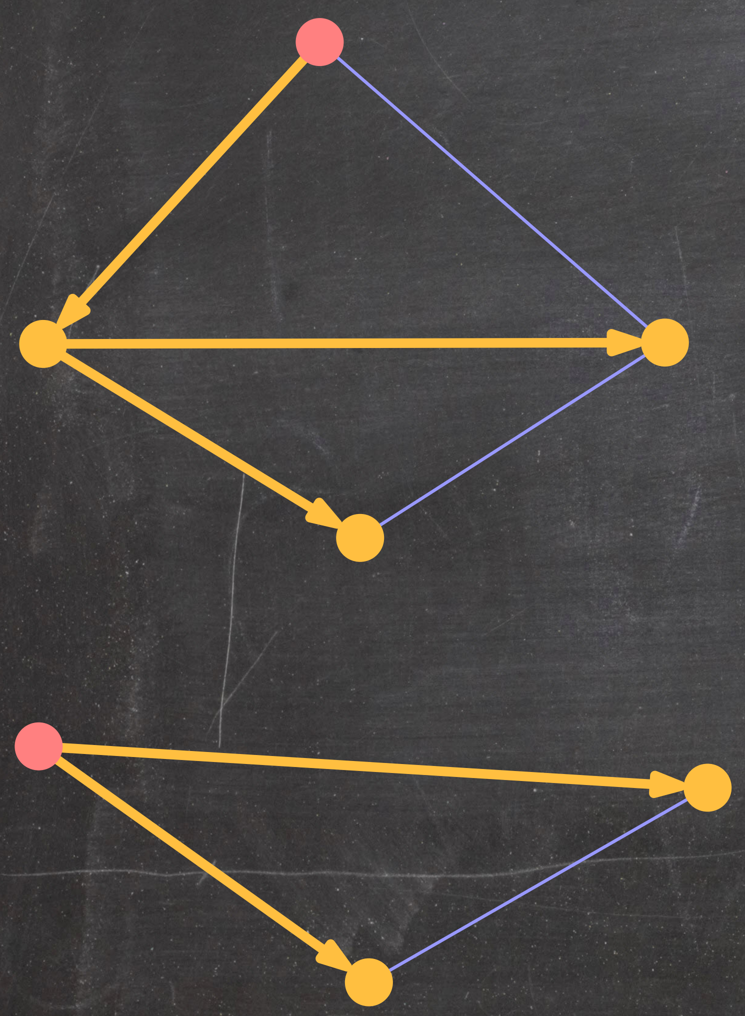
Graph Traversal

We use graph traversal to build a spanning forest of G.



Graph Traversal

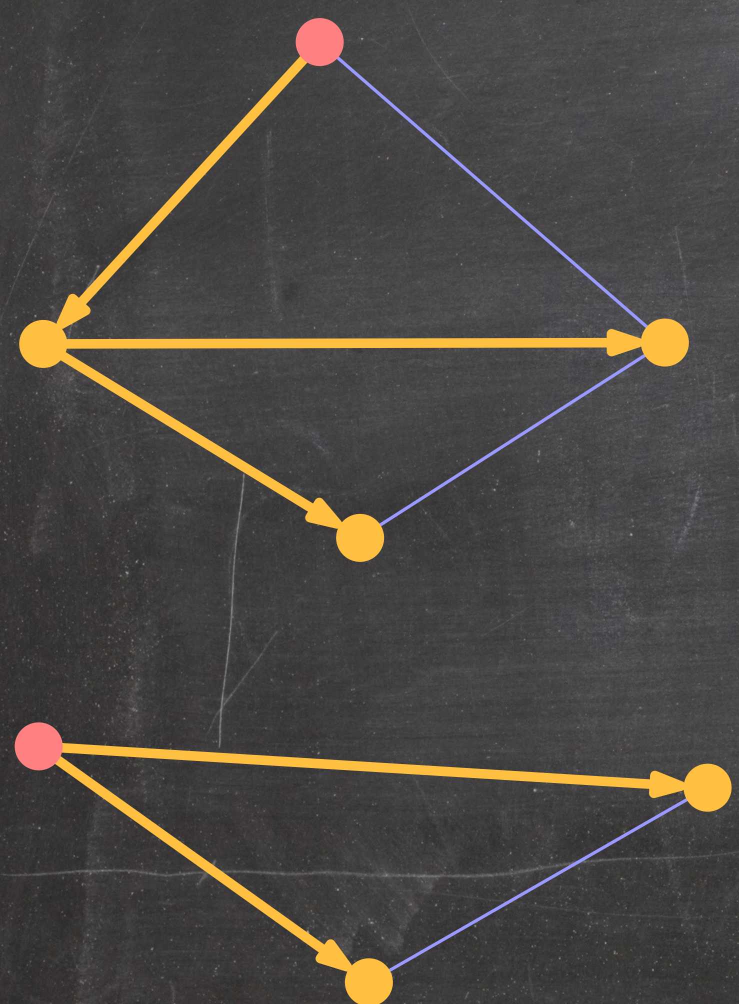
We use graph traversal to build a spanning forest of G.



Different traversal strategies lead to different spanning forests:

- Breadth-first search
- Depth-first search
- Prim's algorithm for computing minimum spanning trees
- Dijkstra's algorithm for computing shortest paths

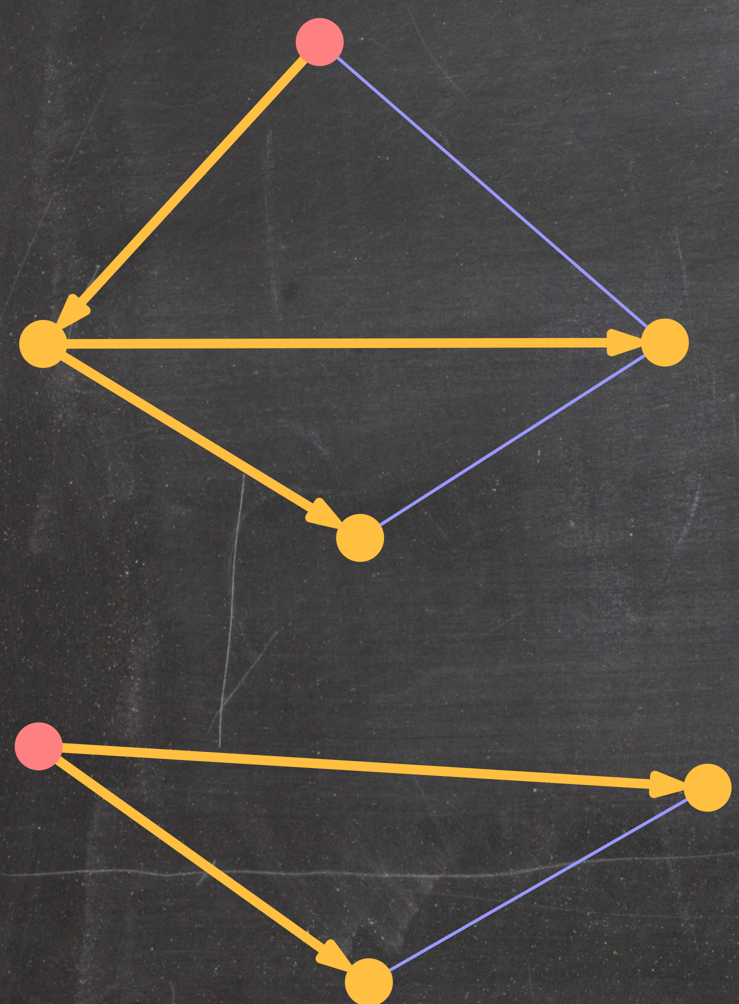
Graph Traversal



TraverseGraph(G)

```
Mark every vertex of G as unexplored
F = []
for every vertex u ∈ G
    do if not u.explored
            then F.append(TraverseFromVertex(G, u))
return F
```

Graph Traversal

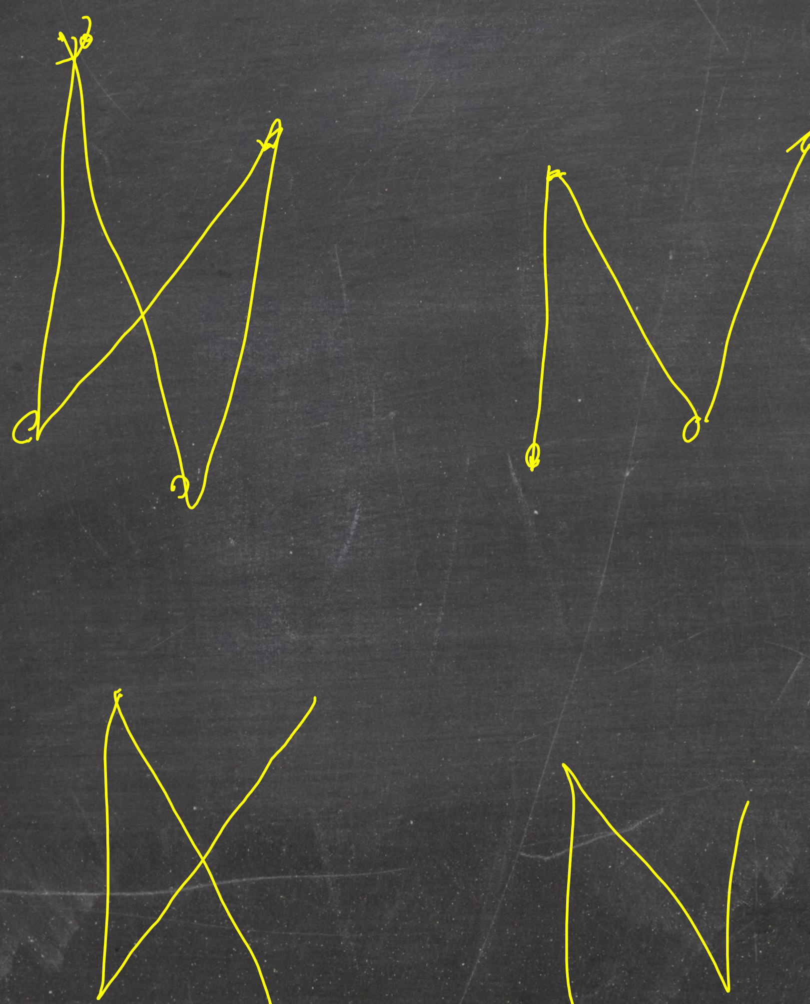


TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
    do Q.add((u, v))
while not Q.isEmpty()
    do (v, w) = Q.remove()
        if not w.explored
            then w.explored = True
                 w.tree = Node(w, [ ])
                 v.tree.children.append(w.tree)
                 for every out-edge (w, x) of v
                     do Q.add((w, x))
return u.tree
```

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.



Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
    do Q.add((u, v))
while not Q.isEmpty()
    do (v, w) = Q.remove()
        if not w.explored
            then w.explored = True
                 w.tree = Node(w, [ ])
                 v.tree.children.append(w.tree)
                 for every out-edge (w, x) of v
                     do Q.add((w, x))
return u.tree
```

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G), then $u \sim_{CC(F)} v$.

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G), then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of CC(G)) then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
    do Q.add((u, v))
while not Q.isEmpty()
    do (v, w) = Q.remove()
        if not w.explored
            then w.explored = True
                 w.tree = Node(w, [ ])
                 v.tree.children.append(w.tree)
                 for every out-edge (w, x) of v
                     do Q.add((w, x))
return u.tree
```

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G), then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

Corollary: Both endpoints of every edge in F are explored.

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G) then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

Corollary: Both endpoints of every edge in F are explored.

TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
   do Q.add((u, v))
while not Q.isEmpty()
   do (v, w) = Q.remove()
        if not w.explored
           then w.explored = True
                w.tree = Node(w, [ ])
                v.tree.children.append(w.tree)
                for every out-edge (w, x) of v
                   do Q.add((w, x))
return u.tree
```

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G), then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

Corollary: Both endpoints of every edge in F are explored.

Corollary: F contains no cycle.

Graph Traversal Computes a Spanning Forest

It computes a subgraph of G because it only adds edges of G to F.

$\Rightarrow$ F has at least as many connected components as G.

To prove:

- F contains no cycle.
- If $u \sim_{CC(G)} v$ (u and v belong to the same component of G), then $u \sim_{CC(F)} v$.

Observation: Every edge (u, v) in Q has at least one explored endpoint, namely u.

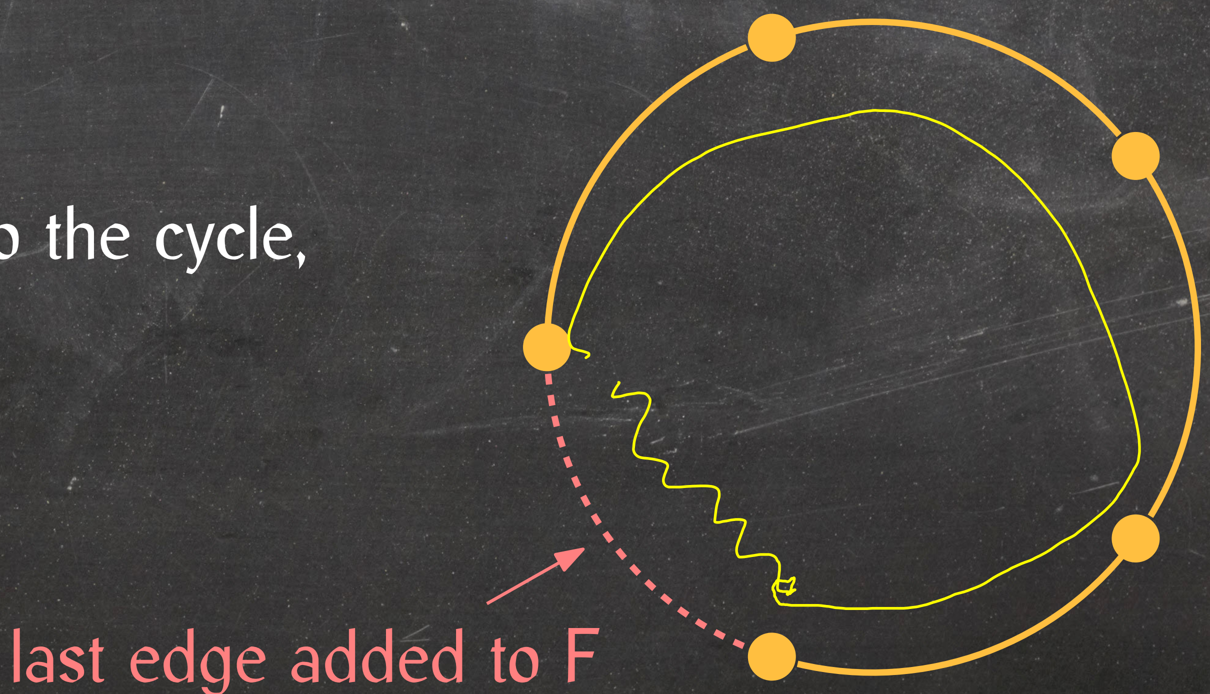
Corollary: Both endpoints of every edge in F are explored.

Corollary: F contains no cycle.

Proof by contradiction:

By the time we add the last edge to the cycle, both its endpoints are explored.

$\Rightarrow$ We would not have added it.



Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:

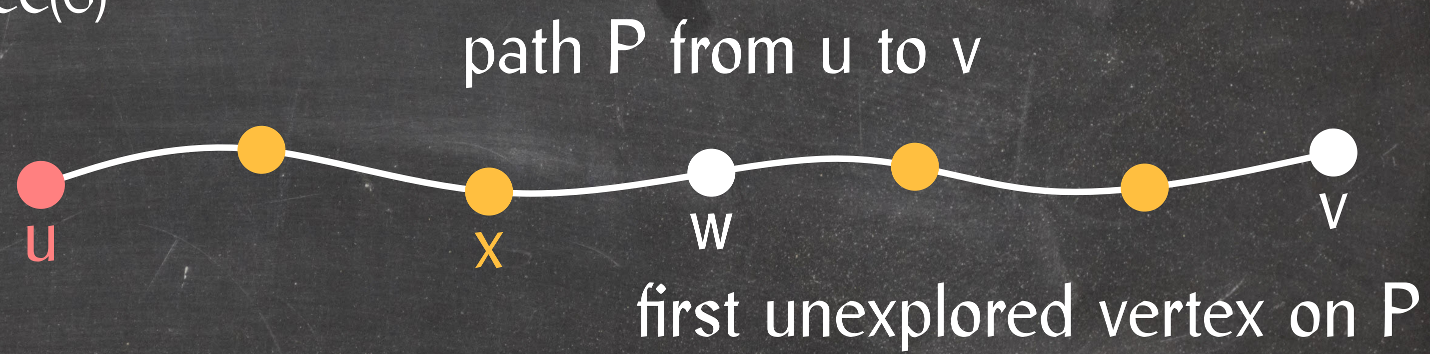
Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:



Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

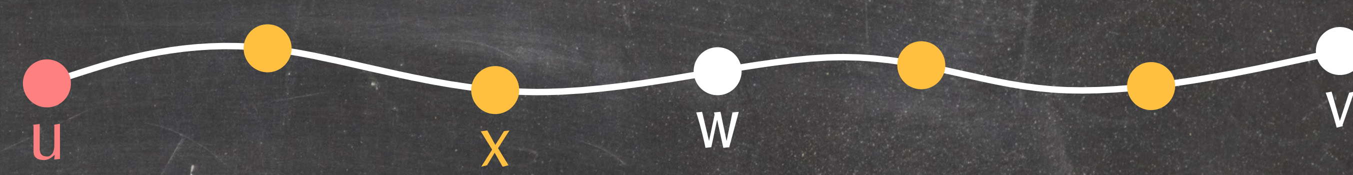
Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:

x adds (x, w) to Q.

$\Rightarrow$ We'd visit w.



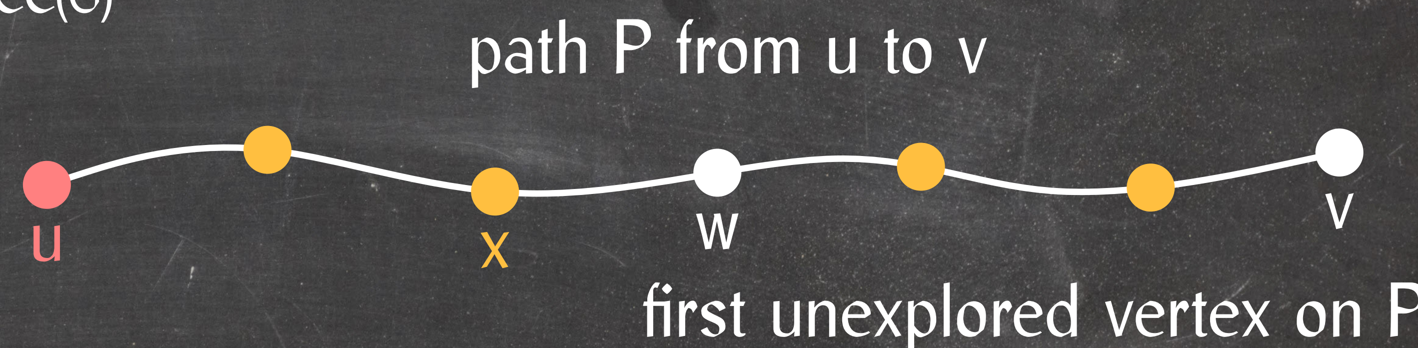
Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:



> x adds (x, w) to Q.
>
> $\Rightarrow$ We'd visit w.

We do not visit a vertex v such that $u \not\sim_{CC(G)} v$:

Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

Proof: By induction on the number of invocations of TraverseFromVertex made so far.

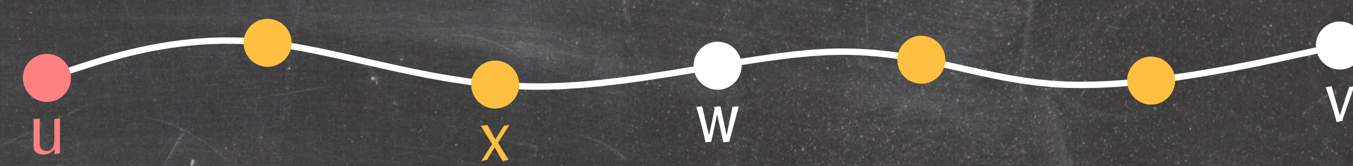
When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:

path P from u to v

x adds (x, w) to Q.

$\Rightarrow$ We'd visit w.



first unexplored vertex on P

We do not visit a vertex v such that $u \not\sim_{CC(G)} v$:

first explored vertex such that $u \not\sim_{CC(G)} v$.

Graph Traversal Computes a Spanning Forest

Lemma: TraverseFromVertex(G, u) visits all vertices v such that $u \sim_{CC(G)} v$ and only those.

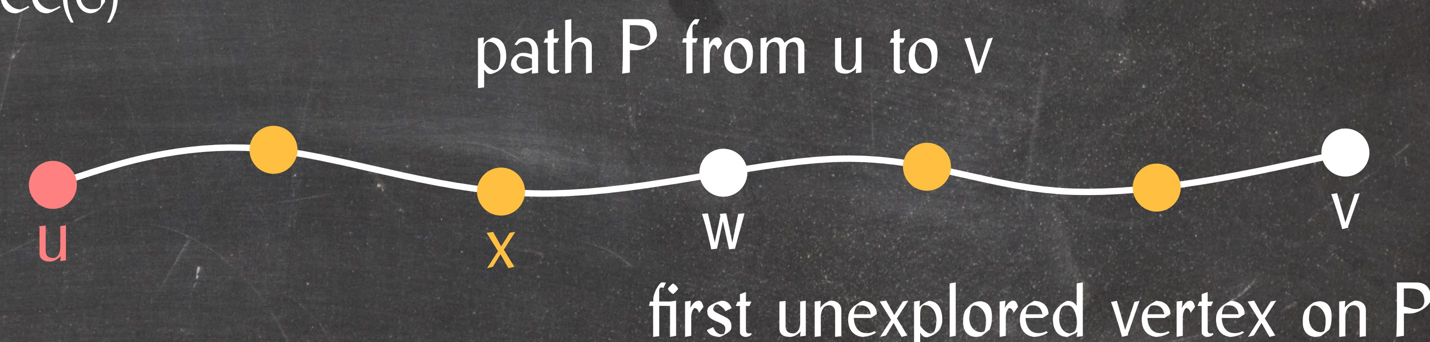
Proof: By induction on the number of invocations of TraverseFromVertex made so far.

When TraverseFromVertex(G, u) is called, every vertex v such that $u \sim_{CC(G)} v$ is unexplored.

We visit all vertices v such that $u \sim_{CC(G)} v$:

x adds (x, w) to Q.

$\Rightarrow$ We'd visit w.

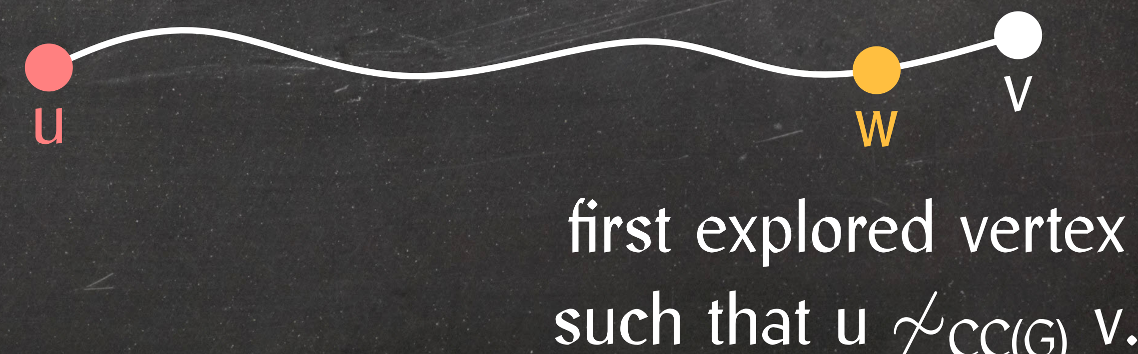


We do not visit a vertex v such that $u \not\sim_{CC(G)} v$:

- v explored because of edge $(w, v) \in Q$.
- w explored before v.

$\Rightarrow$ $w \sim_{CC(G)} u$.

$\Rightarrow$ $v \sim_{CC(G)} u$.



The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

TraverseGraph(G)

```
Mark every vertex of G as unexplored
F = [ ]
for every vertex u ∈ G
   do if not u.explored
         then F.append(TraverseFromVertex(G, u))
return F
```

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

Every edge is added to Q at most once.
$\Rightarrow$ The cost of the for-loops in TraverseFromVertex is $O(m \cdot (1 + t_a))$.

The Cost of Graph Traversal

TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
    do Q.add((u, v))
while not Q.isEmpty()
    do (v, w) = Q.remove()
       if not w.explored
          then w.explored = True
               w.tree = Node(w, [ ])
               v.tree.children.append(w.tree)
               for every out-edge (w, x) of v
                   do Q.add((w, x))
return u.tree
```

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

Every edge is added to Q at most once.

$\Rightarrow$ The cost of the for-loops in TraverseFromVertex is $O(m \cdot (1 + t_a))$.

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

Every edge is added to Q at most once.
$\Rightarrow$ The cost of the for-loops in TraverseFromVertex is $O(m \cdot (1 + t_a))$.

Every edge that is removed must be added first.
$\Rightarrow$ The cost of the while-loop in TraverseFromVertex is $O(m \cdot (1 + t_r))$.

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

Every edge is added to Q at most once.
⇒ The cost of the for-loops in TraverseFromVertex is $O(m \cdot (1 + t_a))$.

Every edge that is removed must be added first.
⇒ The cost of the while-loop in TraverseFromVertex is $O(m \cdot (1 + t_r))$.

TraverseFromVertex(G, u)

```
u.explored = True
u.tree = Node(u, [ ])
Q = an empty edge collection
for every out-edge (u, v) of u
    do Q.add((u, v))
while not Q.isEmpty()
    do (v, w) = Q.remove()
       if not w.explored
          then w.explored = True
               w.tree = Node(w, [ ])
               v.tree.children.append(w.tree)
               for every out-edge (w, x) of v
                  do Q.add((w, x))
return u.tree
```

The Cost of Graph Traversal

Lemma: TraverseGraph takes $O(n + m + m \cdot (t_a + t_r))$ time, where t_a and t_r are the costs of adding and removing an edge from Q, respectively.

TraverseGraph itself takes $O(n)$ time.

Every edge is added to Q at most once.
$\Rightarrow$ The cost of the for-loops in TraverseFromVertex is $O(m \cdot (1 + t_a))$.

Every edge that is removed must be added first.
$\Rightarrow$ The cost of the while-loop in TraverseFromVertex is $O(m \cdot (1 + t_r))$.

Computing Connected Components

- Compute a spanning forest F.
- Collect vertices of trees in F.
- Compute representation of connected components.

Computing Connected Components

- Compute a spanning forest F.
- Collect vertices of trees in F.
- Compute representation of connected components.

CollectComponentVertices(F)

```
L = []
for every tree T ∈ F
    do L.append(CollectDescendantVertices(T))
return L
```

Computing Connected Components

- Compute a spanning forest F.
- Collect vertices of trees in F.
- Compute representation of connected components.

CollectComponentVertices(F)

```
L = []
for every tree T ∈ F
    do L.append(CollectDescendantVertices(T))
return L
```

CollectDescendantVertices(T)

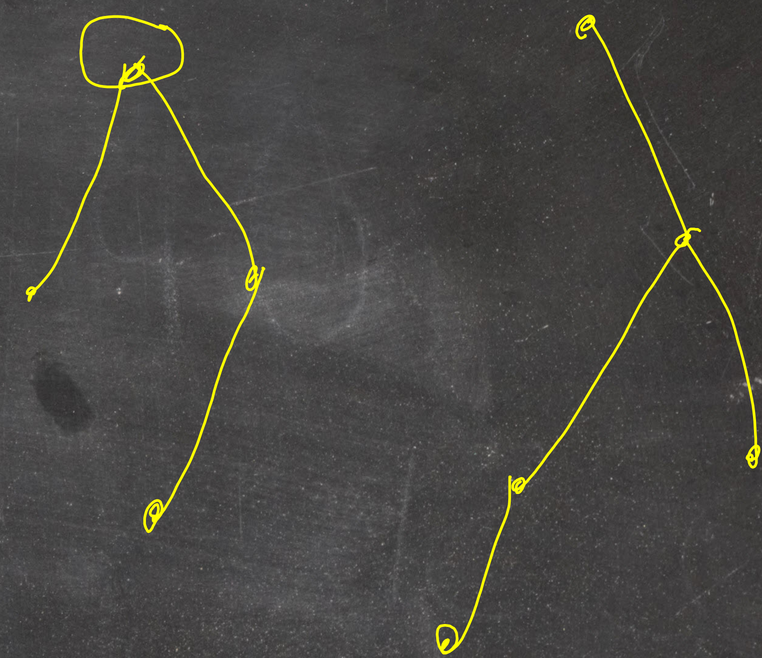
```
L = [T.key]
for every child T′ of T
    do L.concat(CollectDescendantVertices(T′))
return L
```

Computing Connected Components

- Compute a spanning forest F.
- Collect vertices of trees in F.
- Compute representation of connected components.

CollectComponentVertices(F)

```
L = []
for every tree T ∈ F
    do L.append(CollectDescendantVertices(T))
return L
```



CollectDescendantVertices(T)

```
L = [T.key]
for every child T′ of T
    do L.concat(CollectDescendantVertices(T′))
return L
```

Lemma: Collecting the vertices of all components takes $O(n)$ time.

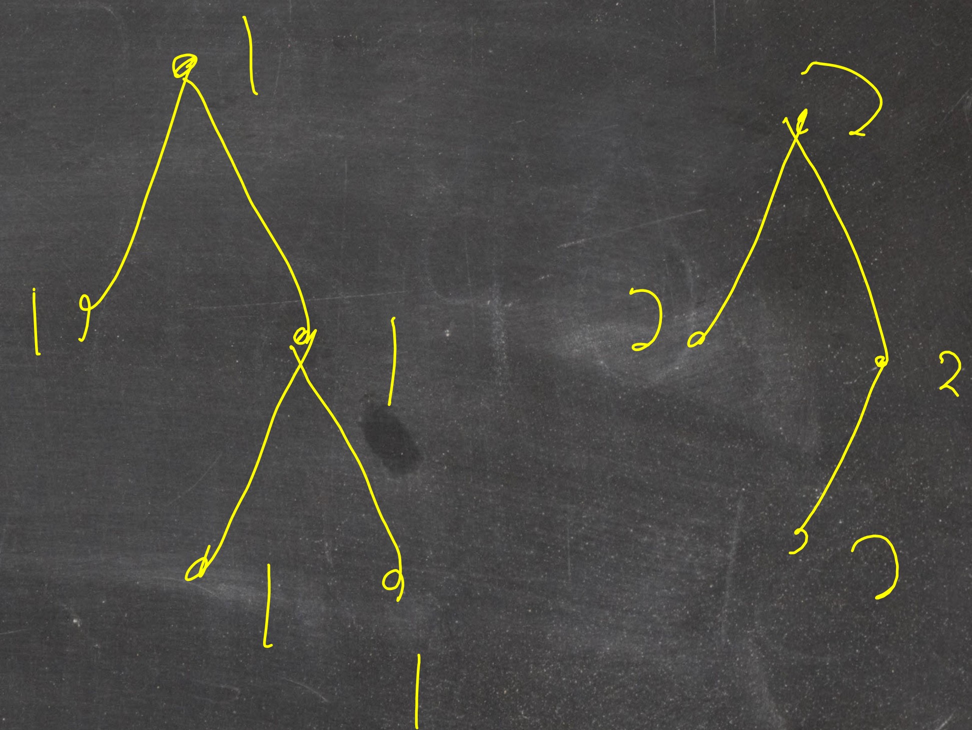
Computing Connected Components

Representation using vertex labels:

ComponentLabels(L)

```
i = 0
for every list L′ ∈ L
    do i = i + 1
       for every vertex v ∈ L′
           do v.cc = i
```

Cost: $O(n)$



Computing Connected Components

Representation as list of graphs:

We already have the right adjacency lists for the vertices.
Need to partition the vertex and edge lists into vertex and edge lists for the components.

Computing Connected Components

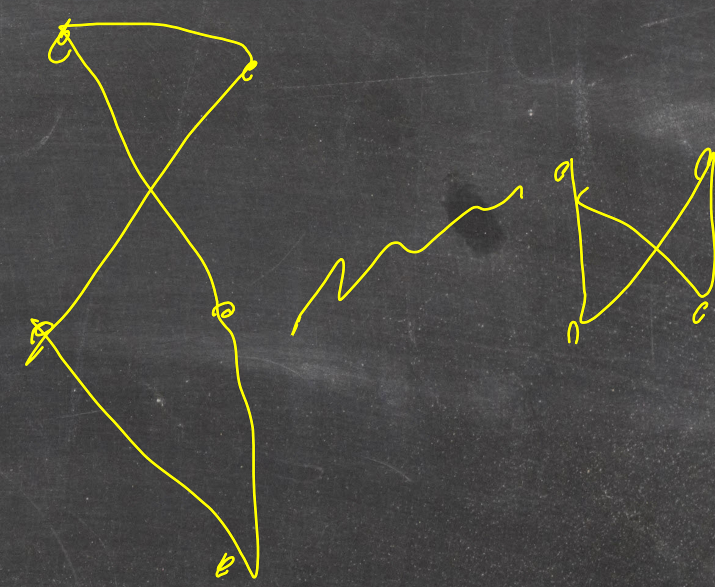
Representation as list of graphs:

We already have the right adjacency lists for the vertices.
Need to partition the vertex and edge lists into vertex and edge lists for the components.

Vertex lists:

BuildVertexLists(L)

```
VL = []
for every list L′ ∈ L
    do VL′ = []
       for every vertex v ∈ L′
           do VL′.append(v)
       VL.append(VL′)
return VL
```

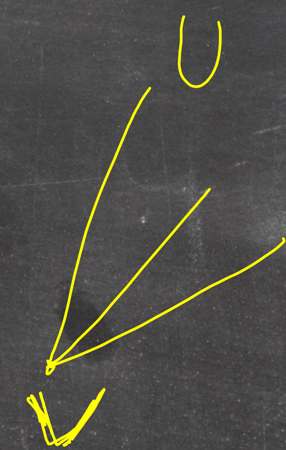


Computing Connected Components

Edge lists:

BuildEdgeLists(G, L)

```
EL = [ ]
for every edge e ∈ G
    do e.collected = False
for every list L′ ∈ L
    do EL′ = [ ]
        for every vertex v ∈ L′
            do for every edge e incident with v
                do if not e.collected
                    then e.collected = True
                         EL′.append(e)
        EL.append(EL′)
return EL
```



Computing Connected Components

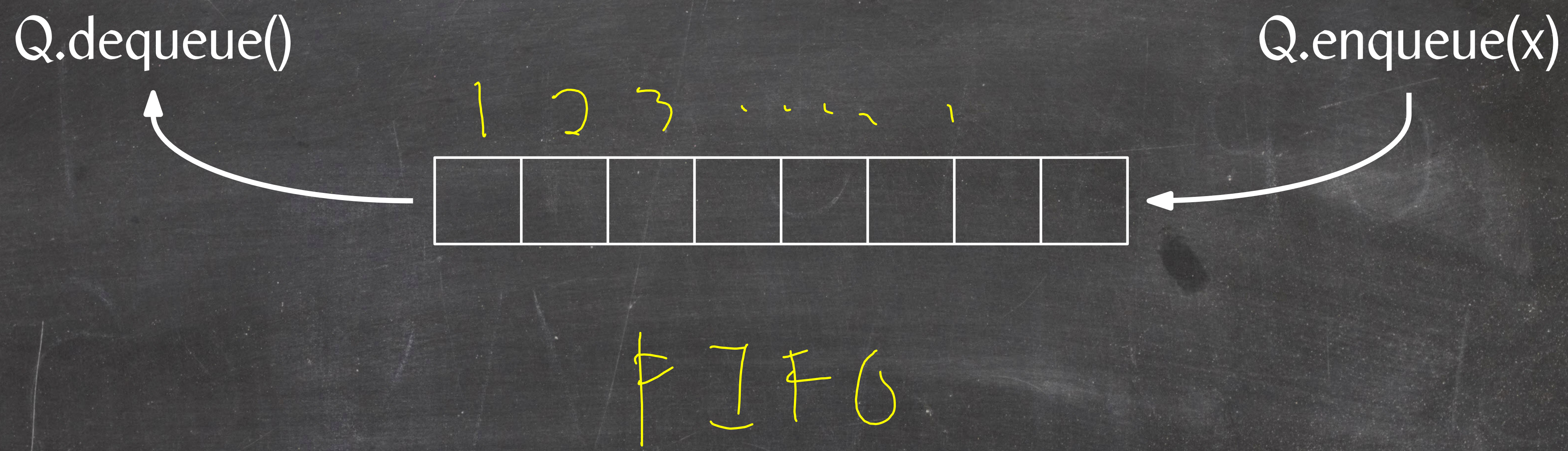
Lemma: The connected components of a graph can be computed in $O(n + m)$ time.

- Building a spanning forest takes $O(n + m + m \cdot (t_a + t_r))$ time.
- Computing the vertex labelling or list of graphs then takes $O(n + m)$ time.
- Using a stack or queue to represent Q, we get $t_a \in O(1)$ and $t_r \in O(1)$.

Breadth-First Search

Breadth-first search (BFS) = graph traversal using a queue to implement Q.

Queue:



Breadth-First Search

Breadth-first search (BFS) = graph traversal using a queue to implement Q.

Queue:

Constant-time implementations:

- Doubly-linked list
- Singly-linked list with tail pointer
- "Circular" array (amortized constant cost)
- Pair of singly-linked lists (functional)

Breadth-First Search

Breadth-first search (BFS) = graph traversal using a queue to implement Q.

Queue:

Constant-time implementations:

- Doubly-linked list
- Singly-linked list with tail pointer
- "Circular" array (amortized constant cost)
- Pair of singly-linked lists (functional)

Lemma: Breadth-first search takes $O(n + m)$ time.

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.

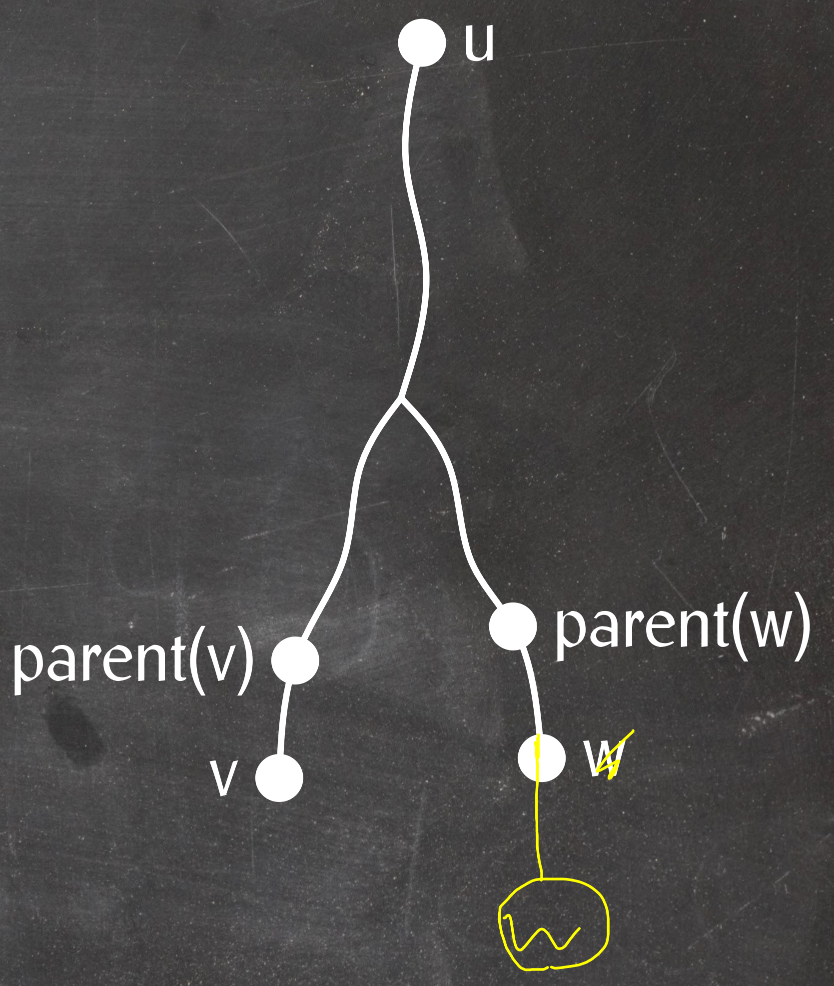
A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.

Assume $d_F(v) < d_F(w)$ and w is visited before v.



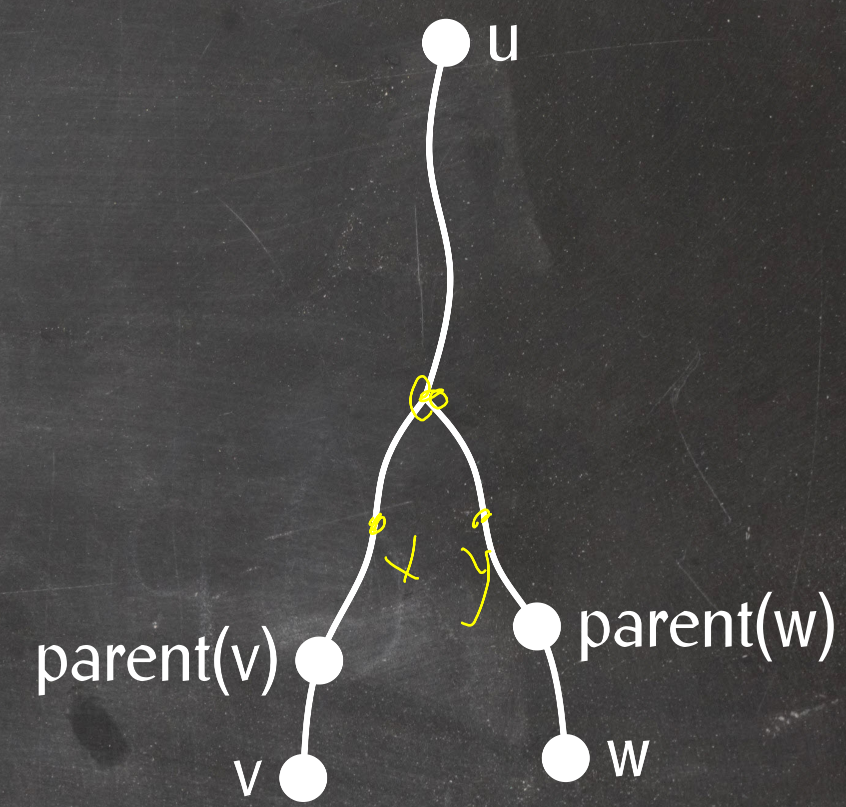
A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.

Assume $d_F(v) < d_F(w)$ and w is visited before v.
Choose such a pair (v, w) so that $d_F(w)$ is minimized.



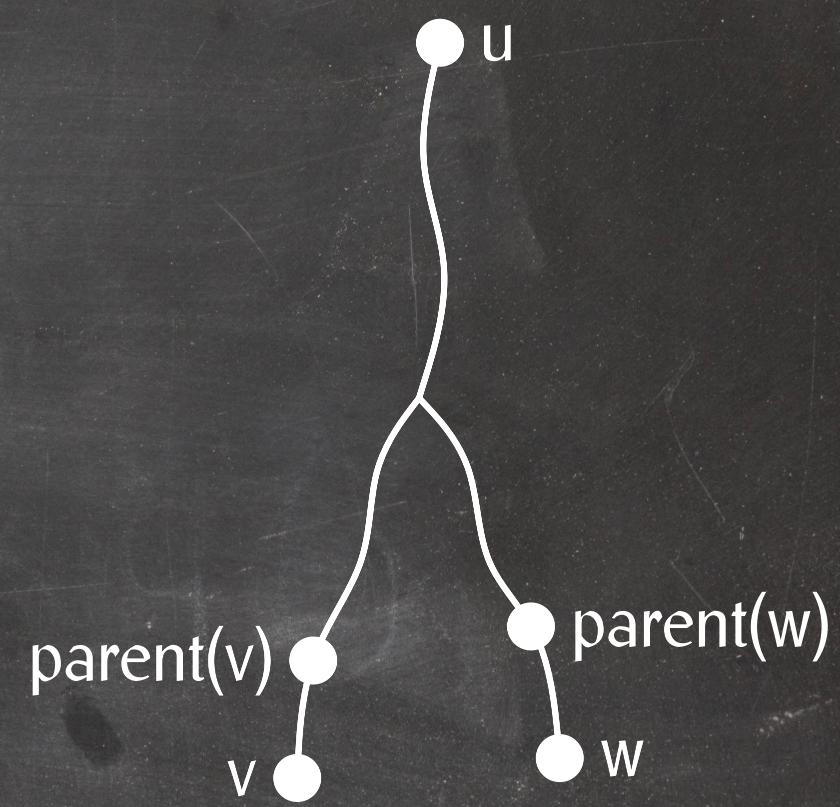
A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.

Assume $d_F(v) < d_F(w)$ and w is visited before v.
Choose such a pair (v, w) so that $d_F(w)$ is minimized.
$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

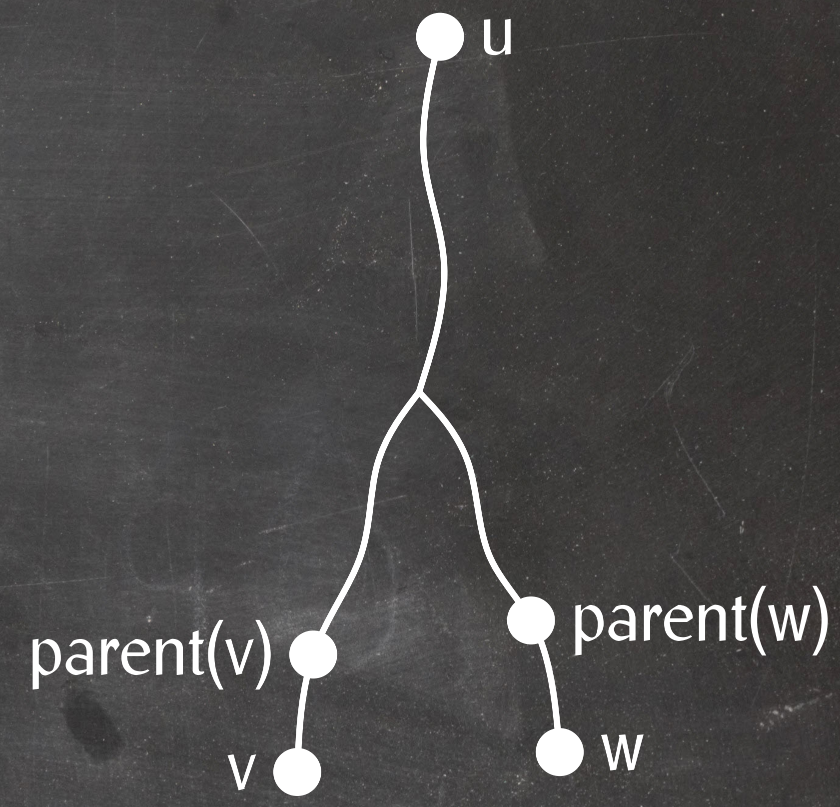


A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

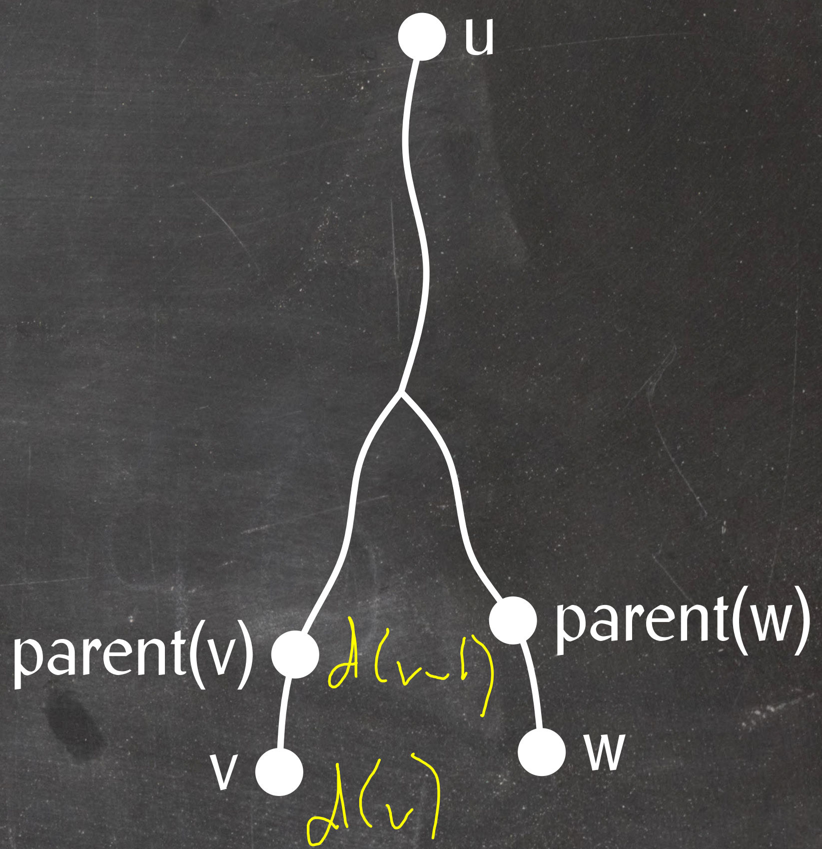
$v \neq u$ because u is visited before any other vertex in the same tree.

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

$v \neq u$ because u is visited before any other vertex in the same tree.

$\Rightarrow$ parent(v) and parent(w) exist and

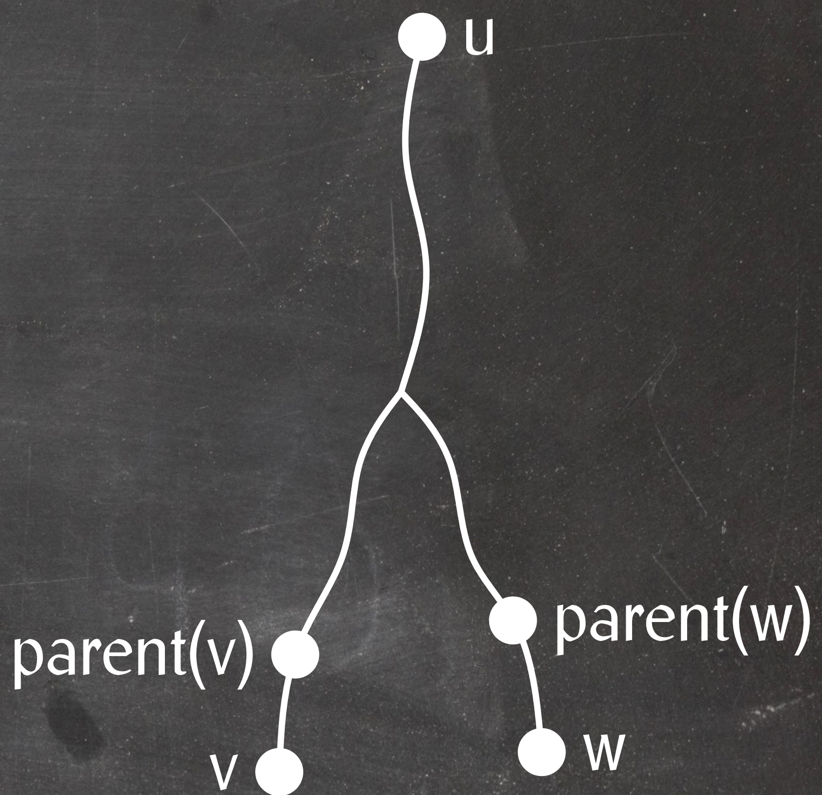
$d_F(\text{parent}(v)) = d_F(v) - 1 < d_F(w) - 1 = d_F(\text{parent}(w))$.

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

$v \neq u$ because u is visited before any other vertex in the same tree.

$\Rightarrow$ parent(v) and parent(w) exist and
$d_F(\text{parent}(v)) = d_F(v) - 1 < d_F(w) - 1 = d_F(\text{parent}(w))$.

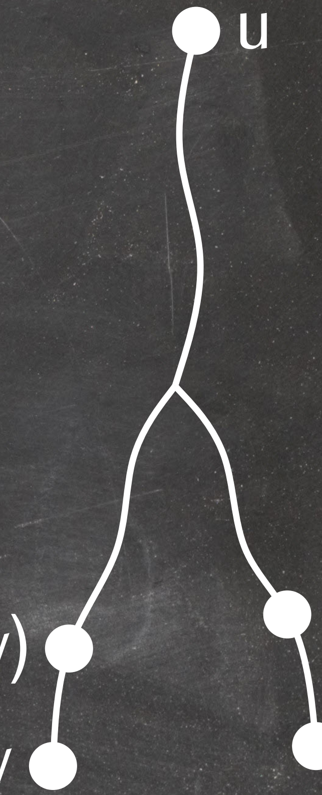
$\Rightarrow$ parent(v) is visited before parent(w).

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

$v \neq u$ because u is visited before any other vertex in the same tree.

$\Rightarrow$ parent(v) and parent(w) exist and
$d_F(\text{parent}(v)) = d_F(v) - 1 < d_F(w) - 1 = d_F(\text{parent}(w))$.

$\Rightarrow$ parent(v) is visited before parent(w).

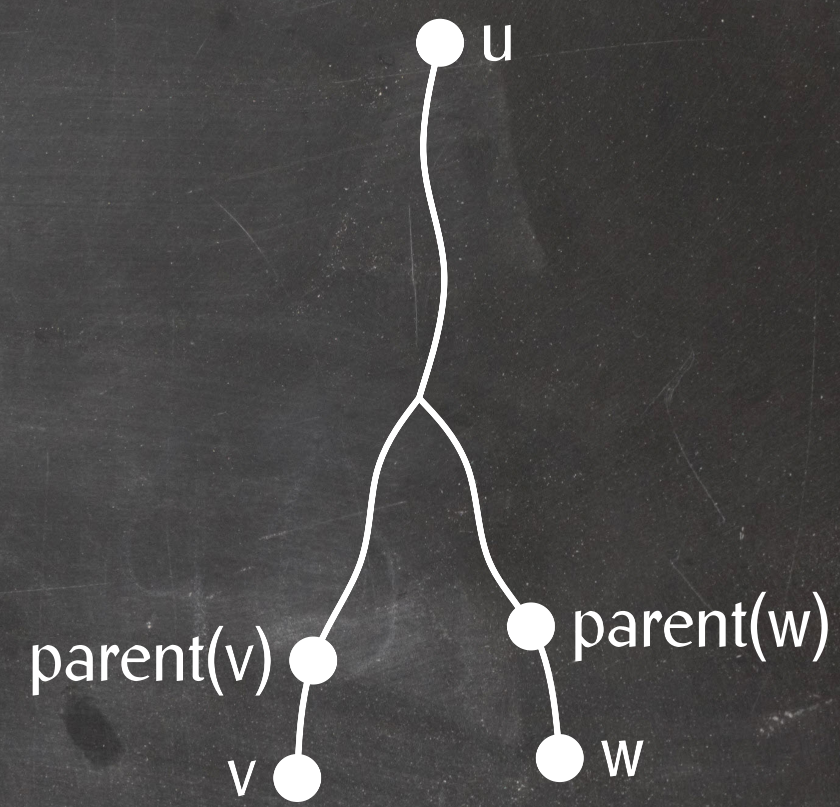
$\Rightarrow$ The edge (parent(v), v) is enqueued before the edge (parent(w), w).

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

$v \neq u$ because u is visited before any other vertex in the same tree.

$\Rightarrow$ parent(v) and parent(w) exist and $d_F(\text{parent}(v)) = d_F(v) - 1 < d_F(w) - 1 = d_F(\text{parent}(w))$.

$\Rightarrow$ parent(v) is visited before parent(w).

$\Rightarrow$ The edge (parent(v), v) is enqueued before the edge (parent(w), w).

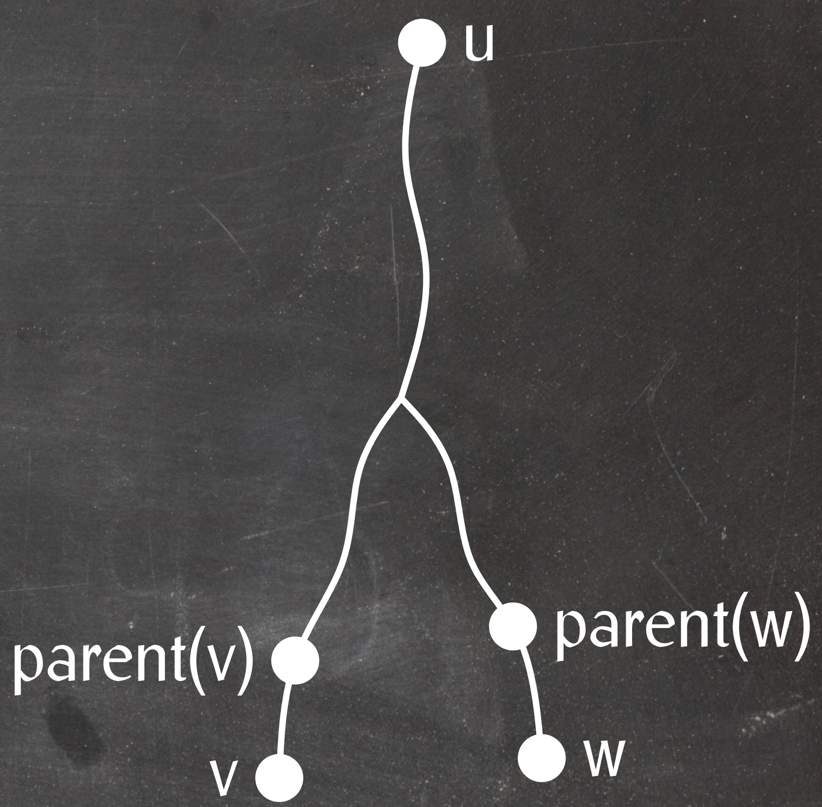
$\Rightarrow$ The edge (parent(v), v) is dequeued before the edge (parent(w), w).

A Property of Undirected BFS Forests

BFS forest = spanning forest computed using BFS

Let the depth $d_F(v)$ of a vertex v in a rooted forest F be the distance from the root of its tree.

Lemma: BFS visits the vertices of each component of F in order of increasing depth.



Assume $d_F(v) < d_F(w)$ and w is visited before v.

Choose such a pair (v, w) so that $d_F(w)$ is minimized.

$w \neq u$ because $d_F(w) > d_F(v) \geq 0$ and $d_F(u) = 0$.

$v \neq u$ because u is visited before any other vertex in the same tree.

$\Rightarrow$ parent(v) and parent(w) exist and $d_F(\text{parent}(v)) = d_F(v) - 1 < d_F(w) - 1 = d_F(\text{parent}(w))$.

$\Rightarrow$ parent(v) is visited before parent(w).

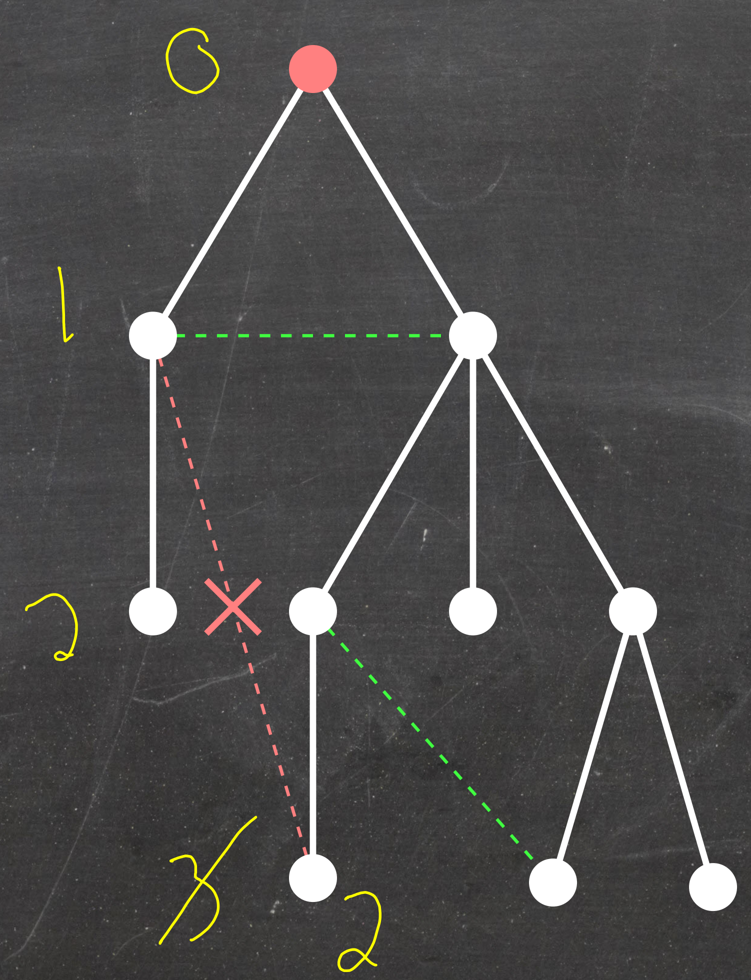
$\Rightarrow$ The edge (parent(v), v) is enqueued before the edge (parent(w), w).

$\Rightarrow$ The edge (parent(v), v) is dequeued before the edge (parent(w), w).

$\Rightarrow$ v is visited before w, a contradiction.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.



$$d(v) = \begin{cases} d(w) - 1 \\ d(w) \\ d(w) + 1 \end{cases}$$

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before $\text{parent}(w)$.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before parent(w).

$\Rightarrow$ The edge (v, w) is enqueued before the edge $(\text{parent}(w), w)$.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before parent(w).

$\Rightarrow$ The edge (v, w) is enqueued before the edge $(\text{parent}(w), w)$.

$\Rightarrow$ The edge (v, w) is dequeued before the edge $(\text{parent}(w), w)$.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before parent(w).

$\Rightarrow$ The edge (v, w) is enqueued before the edge (parent(w), w).

$\Rightarrow$ The edge (v, w) is dequeued before the edge (parent(w), w).

w is unexplored when the edge (parent(w), w) is dequeued.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before parent(w).

$\Rightarrow$ The edge (v, w) is enqueued before the edge (parent(w), w).

$\Rightarrow$ The edge (v, w) is dequeued before the edge (parent(w), w).

w is unexplored when the edge (parent(w), w) is dequeued.

$\Rightarrow$ w is unexplored when the edge (v, w) is dequeued.

A Property of Undirected BFS Forests

Lemma: For every edge (v, w) of G and any BFS forest F of G, the depths of v and w in F differ by at most one.

Assume $d_F(w) > d_F(v) + 1$.

$\Rightarrow$ $d_F(\text{parent}(w)) > d_F(v)$.

$\Rightarrow$ v is visited before parent(w).

$\Rightarrow$ The edge (v, w) is enqueued before the edge (parent(w), w).

$\Rightarrow$ The edge (v, w) is dequeued before the edge (parent(w), w).

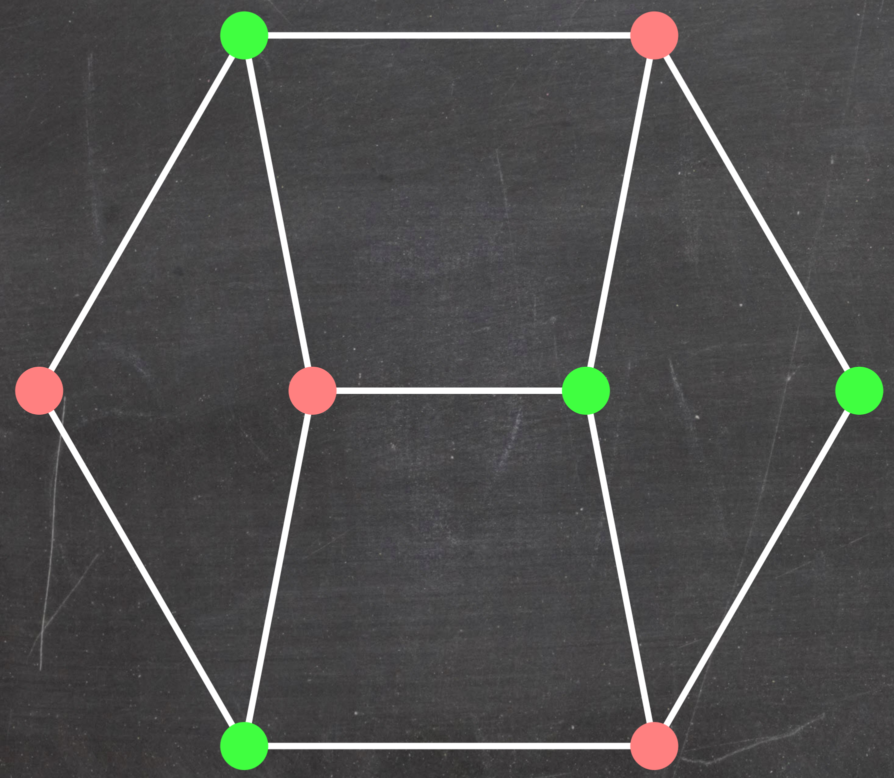
w is unexplored when the edge (parent(w), w) is dequeued.

$\Rightarrow$ w is unexplored when the edge (v, w) is dequeued.

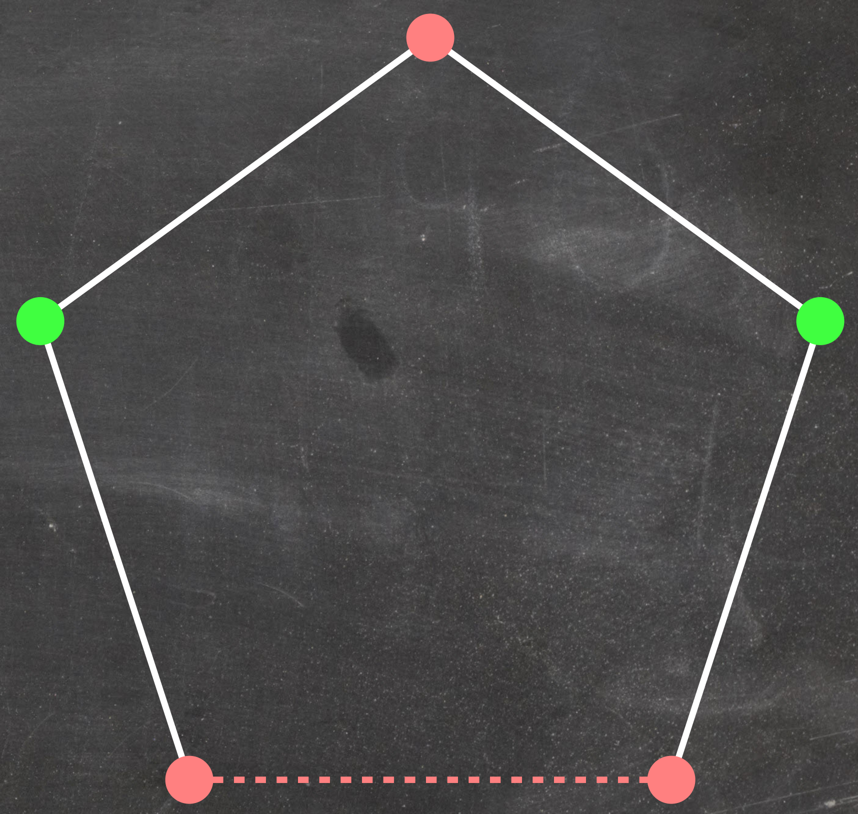
$\Rightarrow$ w would be added to the list of v's children, a contradiction.

Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.



bipartite



not bipartite

Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

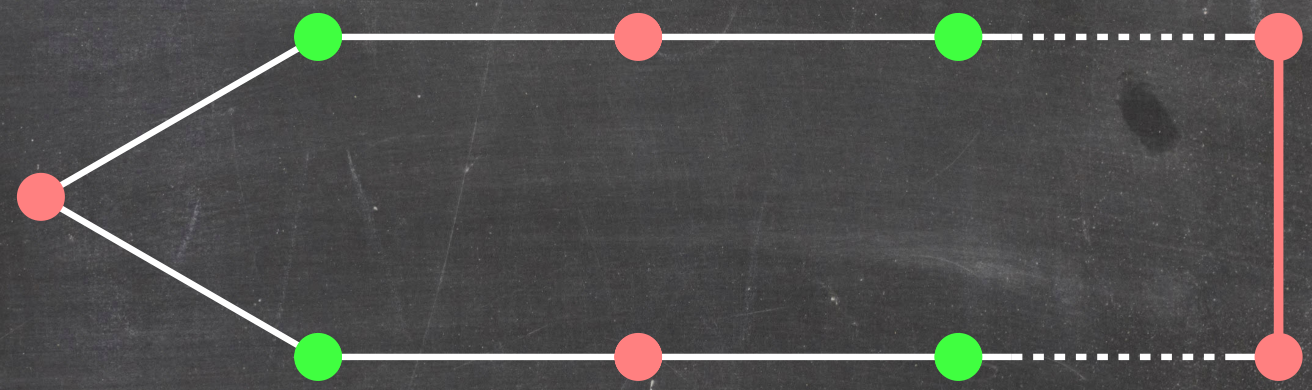
Lemma: A graph is bipartite if and only if it contains no odd cycle.

Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

Lemma: A graph is bipartite if and only if it contains no odd cycle.

Assume there exists an odd cycle in G.

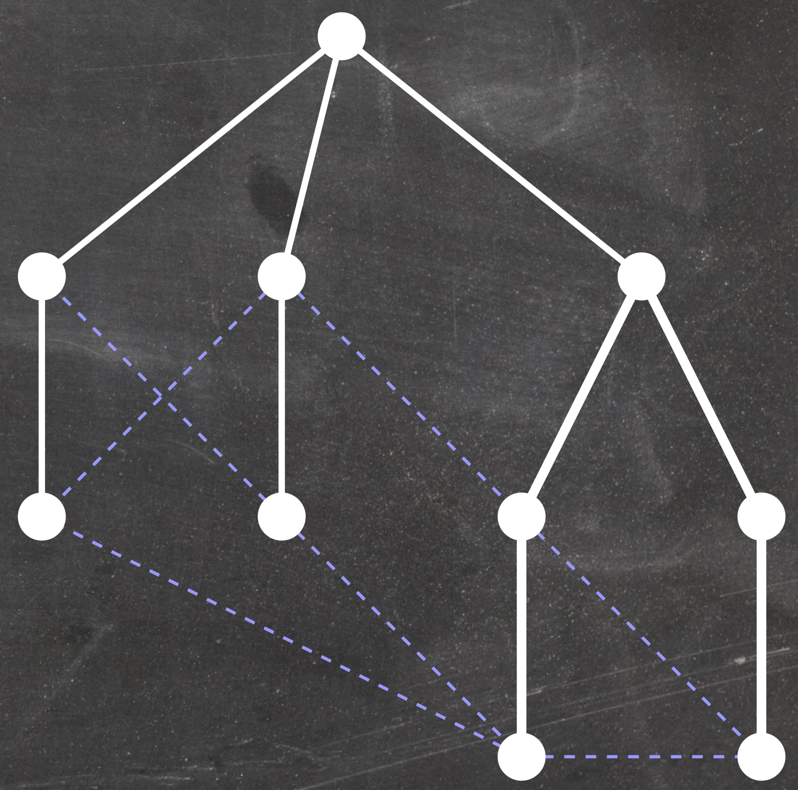


Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

Lemma: A graph is bipartite if and only if it contains no odd cycle.

Let F be a BFS forest of G.



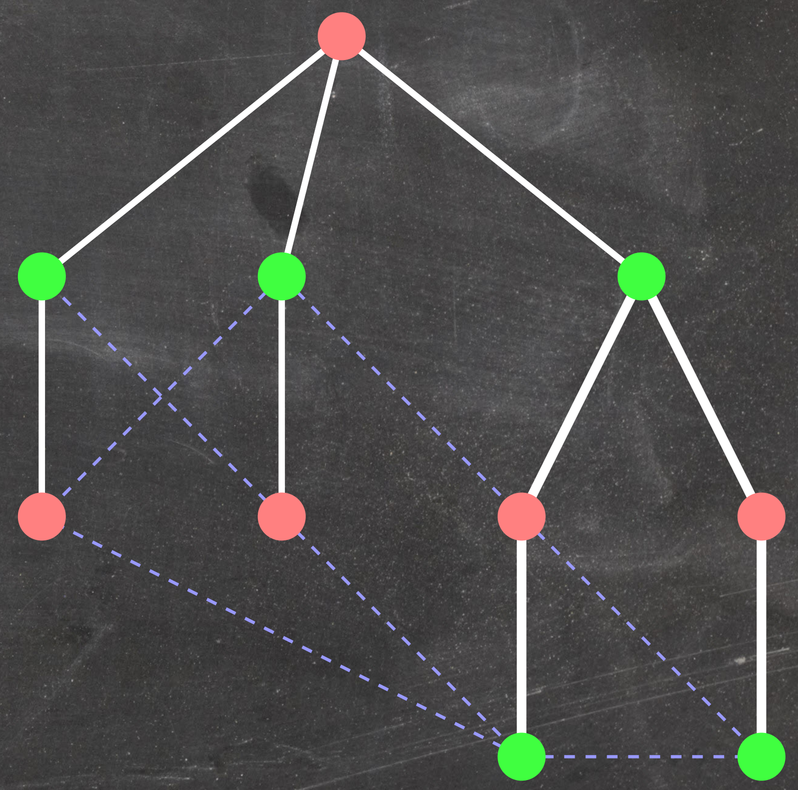
Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

Lemma: A graph is bipartite if and only if it contains no odd cycle.

Let F be a BFS forest of G.

Add vertices on odd levels to U, on even levels to W.



Bipartite Graphs

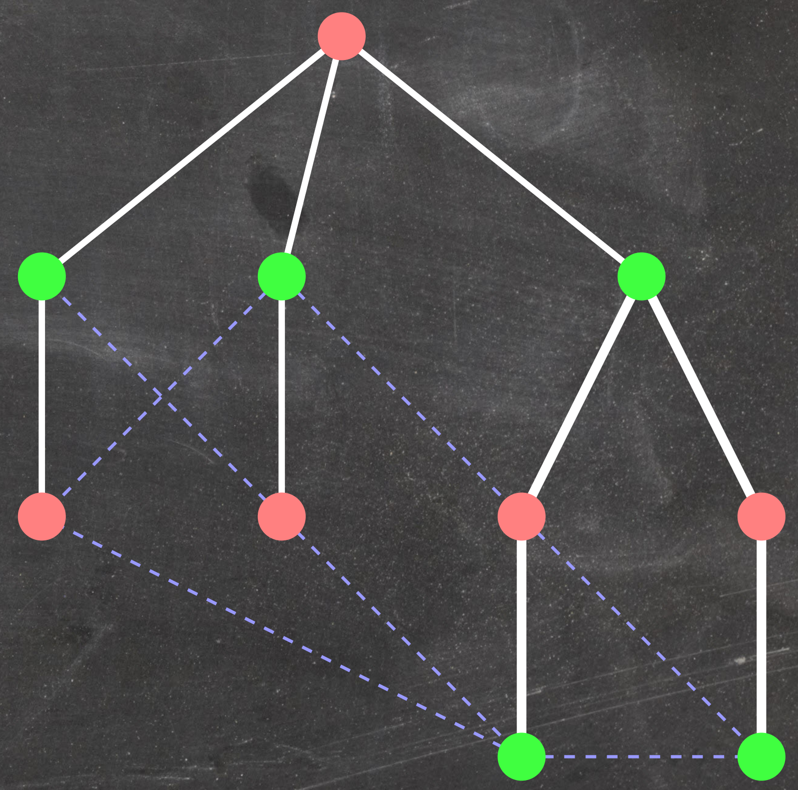
A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

Lemma: A graph is bipartite if and only if it contains no odd cycle.

Let F be a BFS forest of G.

Add vertices on odd levels to U, on even levels to W.

This is the only partition that satisfies the edges of F!



Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

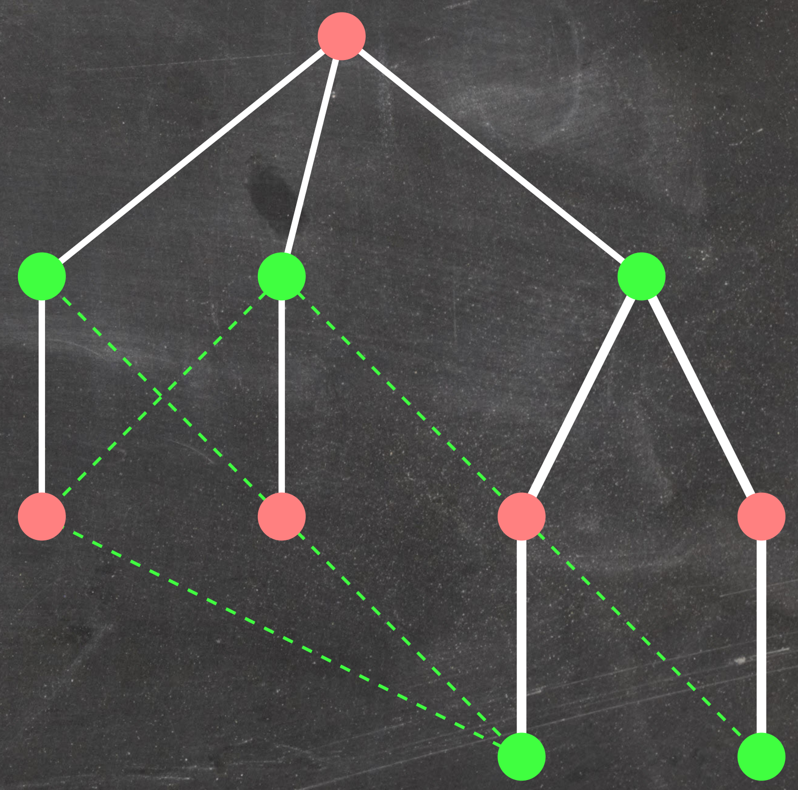
Lemma: A graph is bipartite if and only if it contains no odd cycle.

Let F be a BFS forest of G.

Add vertices on odd levels to U, on even levels to W.

This is the only partition that satisfies the edges of F!

⇒ G is bipartite if and only if there is no edge with both endpoints on the same level.



Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

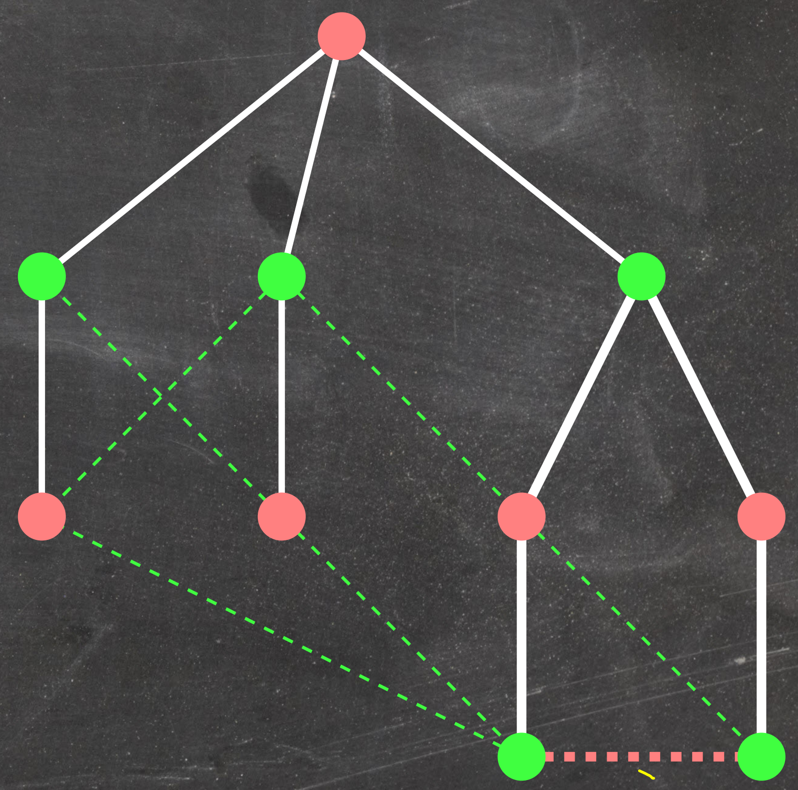
Lemma: A graph is bipartite if and only if it contains no odd cycle.

Let F be a BFS forest of G.

Add vertices on odd levels to U, on even levels to W.

This is the only partition that satisfies the edges of F!

⇒ G is bipartite if and only if there is no edge with both endpoints on the same level.



Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

Lemma: A graph is bipartite if and only if it contains no odd cycle.

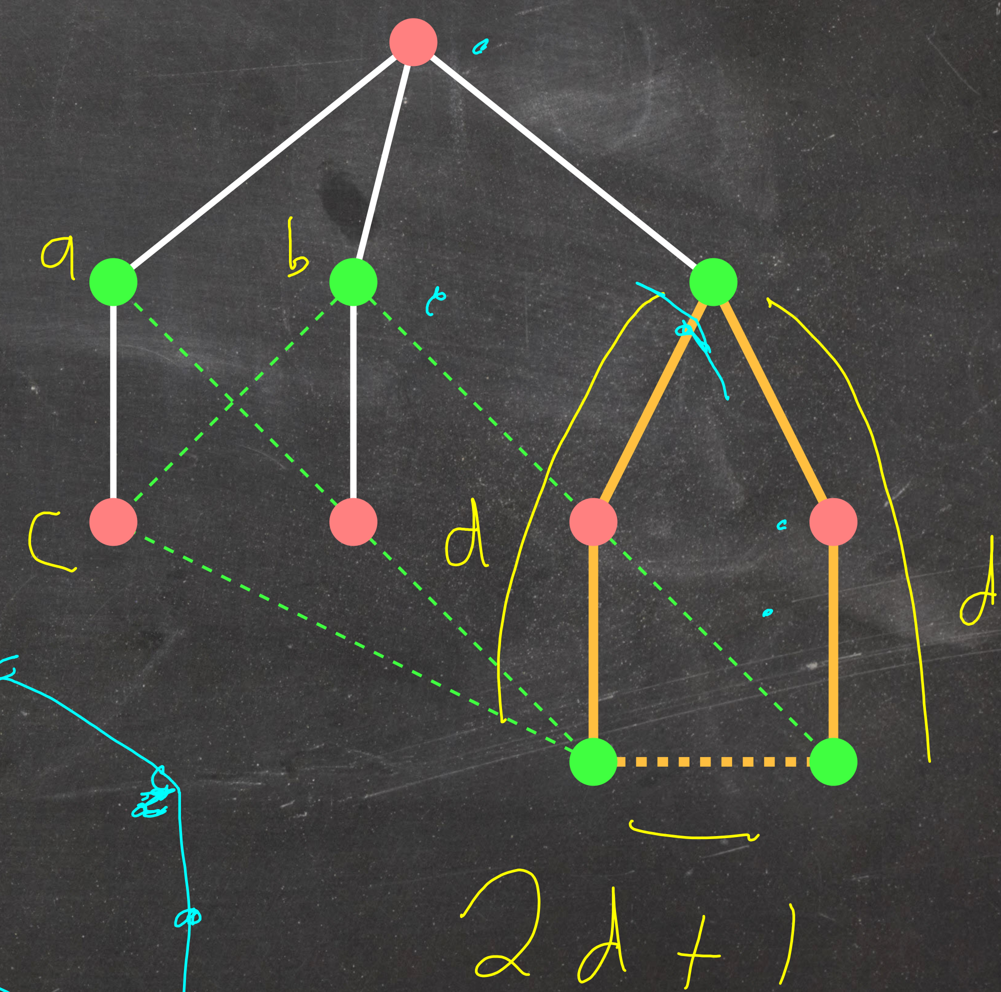
Let F be a BFS forest of G.

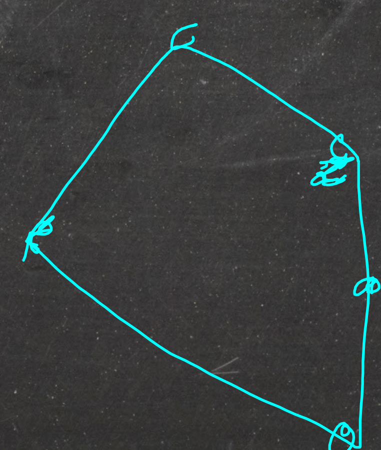
Add vertices on odd levels to U, on even levels to W.

This is the only partition that satisfies the edges of F!

$\Rightarrow$ G is bipartite if and only if there is no edge with both endpoints on the same level.

If there is such an edge, there's an odd cycle.





Bipartite Graphs

A graph is bipartite if its vertices can be partitioned into two sets (U, W) such that every edge has one endpoint in U and one endpoint in W.

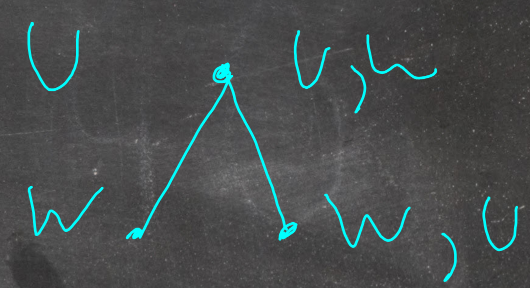
Lemma: A graph is bipartite if and only if it contains no odd cycle.

Lemma: Given a BFS forest F of G, G is bipartite if and only if there is no edge in G with both endpoints on the same level in F.

Bipartiteness Testing

- Compute BFS forest F of G.
- Collect vertices on alternating levels of F into two sets (U, W).
- Test whether any edge has both endpoints in the same set, U or W.
- If so, report the odd cycle induced by such an edge.
- Otherwise, report the bipartition (U, W).

Collecting vertices on alternating levels:



AlternatingLevels(F)

```
U = W = []
for every tree T in F
    do AlternatingLevels′(T, U, W)
return (U, W)
```

AlternatingLevels′(T, U, W)

```
U.append(T.key)
for every child T′ of T
    do AlternatingLevels′(T′, W, U)
```

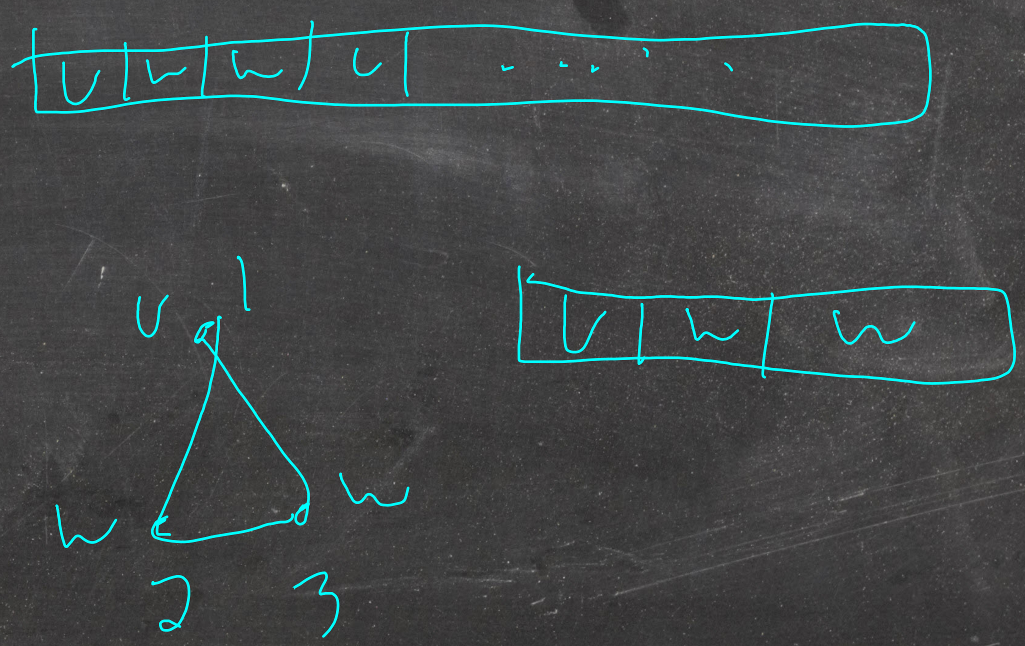
Bipartiteness Testing

- Compute BFS forest F of G.
- Collect vertices on alternating levels of F into two sets (U, W).
- Test whether any edge has both endpoints in the same set, U or W.
- If so, report the odd cycle induced by such an edge.
- Otherwise, report the bipartition (U, W).

Testing for an “odd edge”:

OddEdge(G, U, W)

```
A = an array of size n
for every vertex u ∈ U
    do A[u] = “U”
for every vertex w ∈ W
    do A[w] = “W”
for every edge (u, w) ∈ G
    do if A[u] = A[w]
          then return (u, w)
return Nothing
```



Bipartiteness Testing

- Compute BFS forest F of G.
- Collect vertices on alternating levels of F into two sets (U, W).
- Test whether any edge has both endpoints in the same set, U or W.
- If so, report the odd cycle induced by such an edge.
- Otherwise, report the bipartition (U, W).

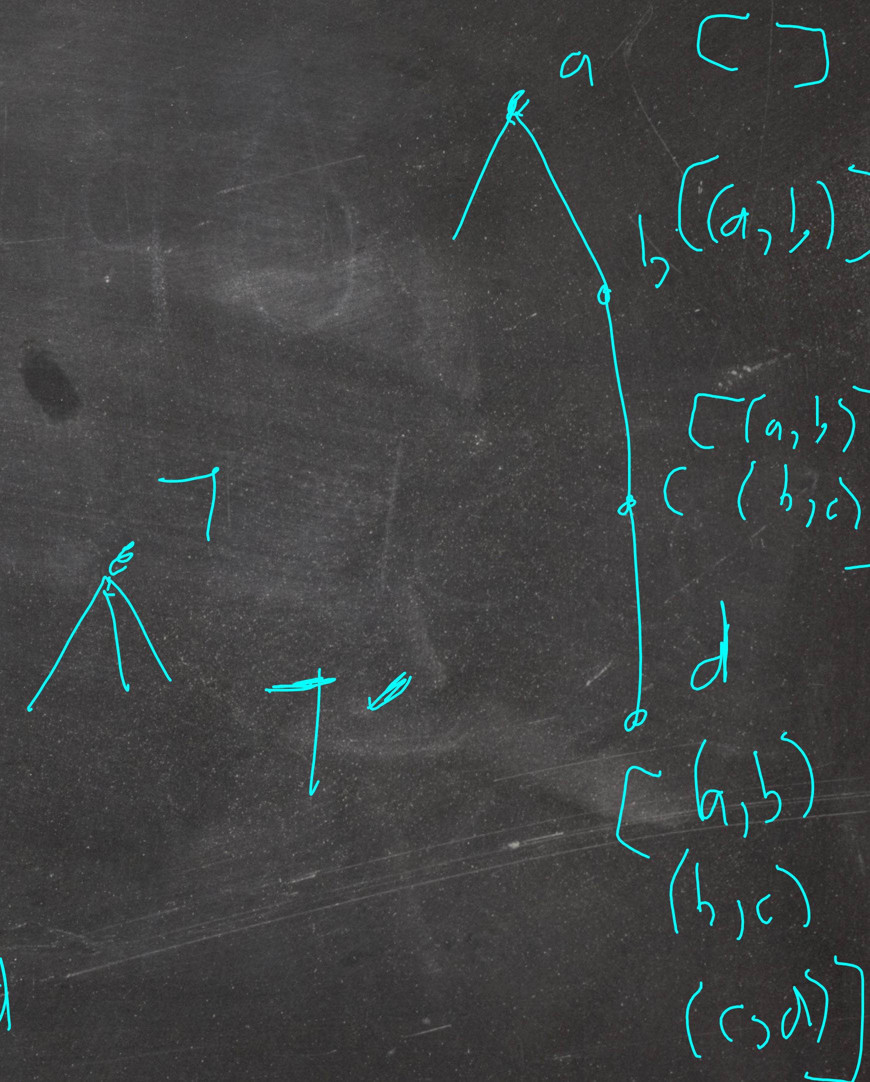
Finding the ancestor edges of all vertices:

AncestorEdges(F)

```
L = an empty list of vertex-vertex list pairs
for every tree T ∈ F
    do AncestorEdges′(T, [ ], L)
return L
```

AncestorEdges′(T, A, L)

```
L = L.append([(T.key, A)])
for every child T′ of T
    do AncestorEdges′(T′, [(T.key, T′.key)] ++ A, L)
```



Bipartiteness Testing

- Compute BFS forest F of G.
- Collect vertices on alternating levels of F into two sets (U, W).
- Test whether any edge has both endpoints in the same set, U or W.
- If so, report the odd cycle induced by such an edge.
- Otherwise, report the bipartition (U, W).

Reporting an odd cycle:

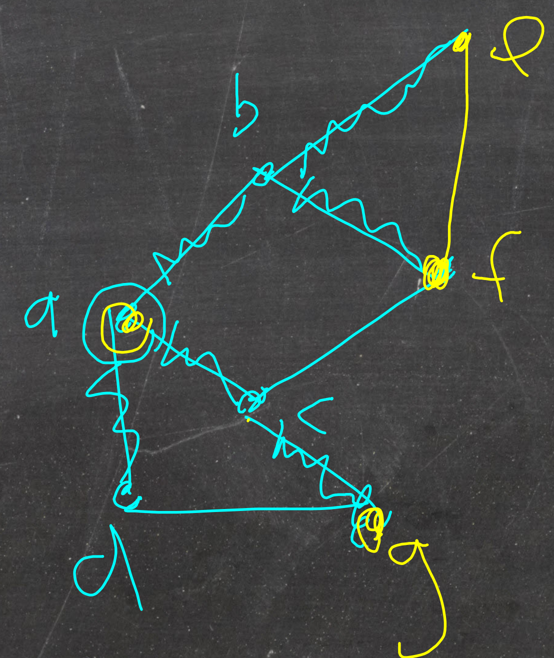
OddCycle(L, (u, w))

```
Find (u, A_u) and (w, A_w) in L
C_u = C_w = []
while A_u.head ≠ A_w.head
    do C_u.append(A_u.head)
       C_w.append(A_w.head)
       A_u = A_u.tail
       A_w = A_w.tail
C_u.reverse().concat([(u, w)]).concat(C_w)
return C_u
```

Bipartiteness Testing

- Compute BFS forest F of G.
- Collect vertices on alternating levels of F into two sets (U, W).
- Test whether any edge has both endpoints in the same set, U or W.
- If so, report the odd cycle induced by such an edge.
- Otherwise, report the bipartition (U, W).

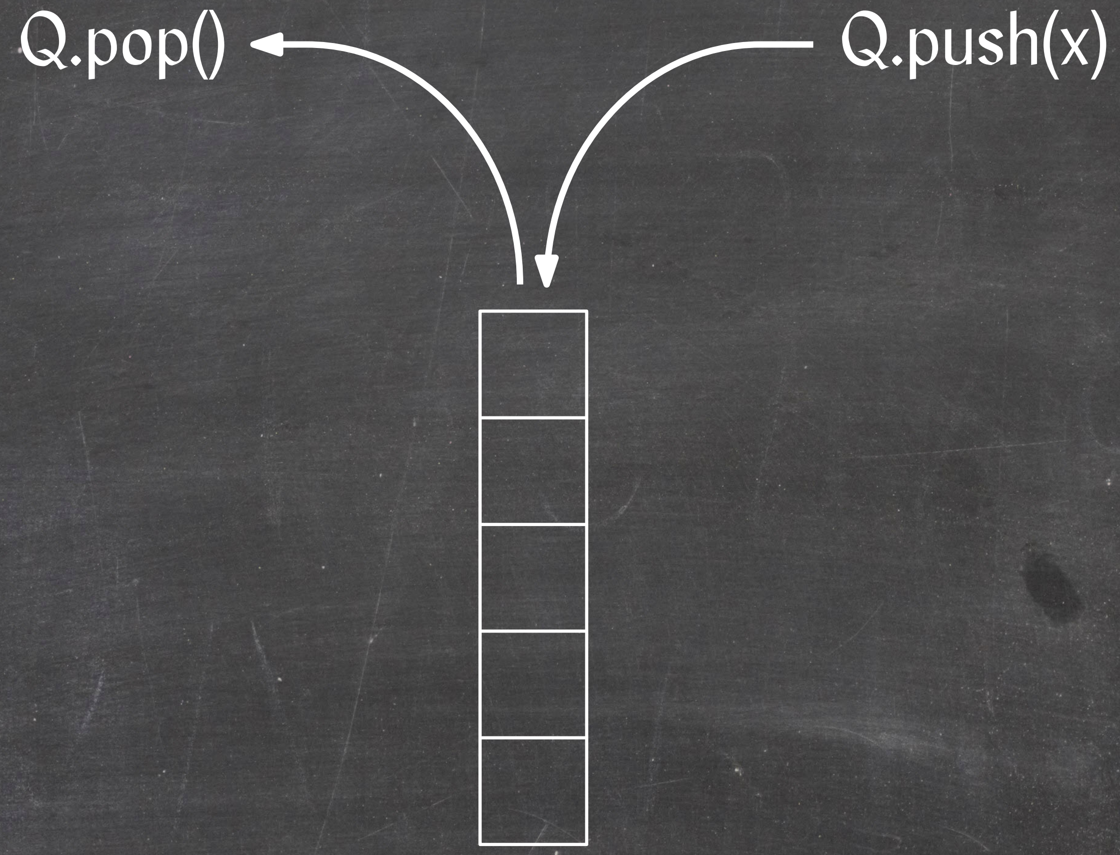
Lemma: It takes linear time to test whether a graph G is bipartite and either report a valid bipartition or an odd cycle in G.



Depth-First Search

Depth-first search (DFS) = graph traversal using a stack to implement Q.

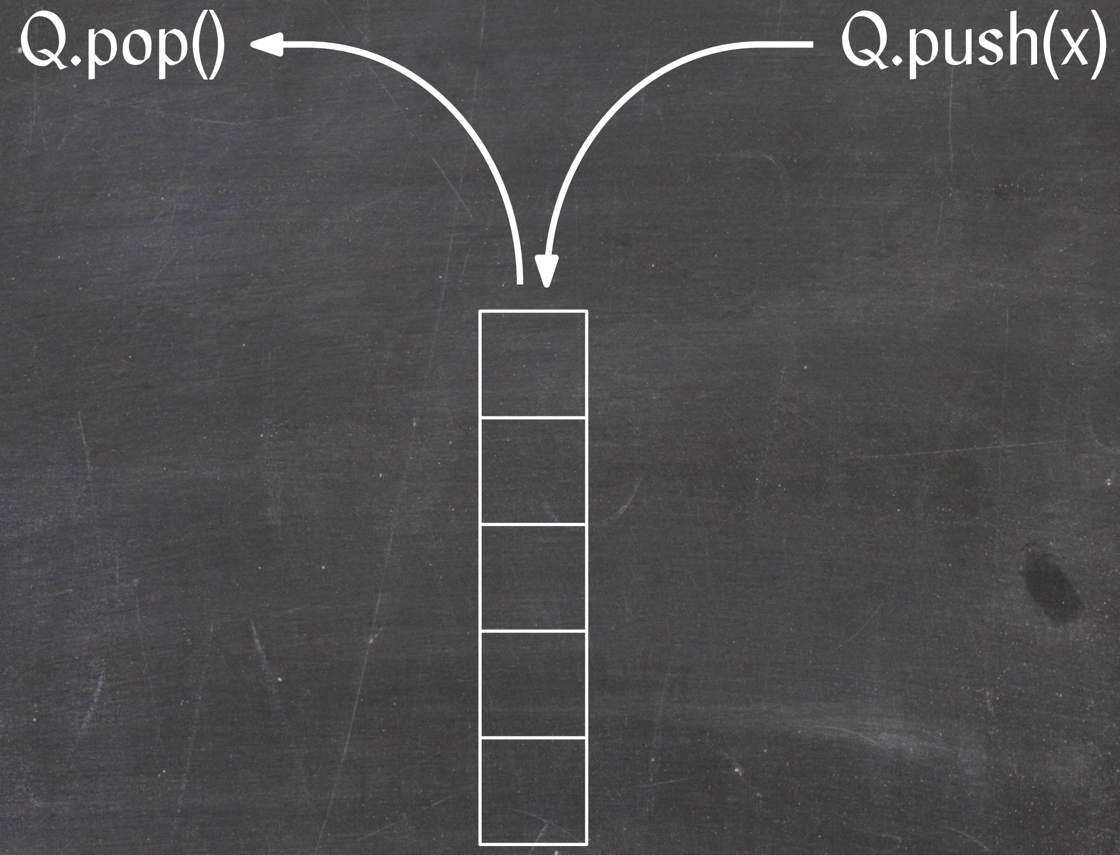
Stack:



Depth-First Search

Depth-first search (DFS) = graph traversal using a stack to implement Q.

Stack:



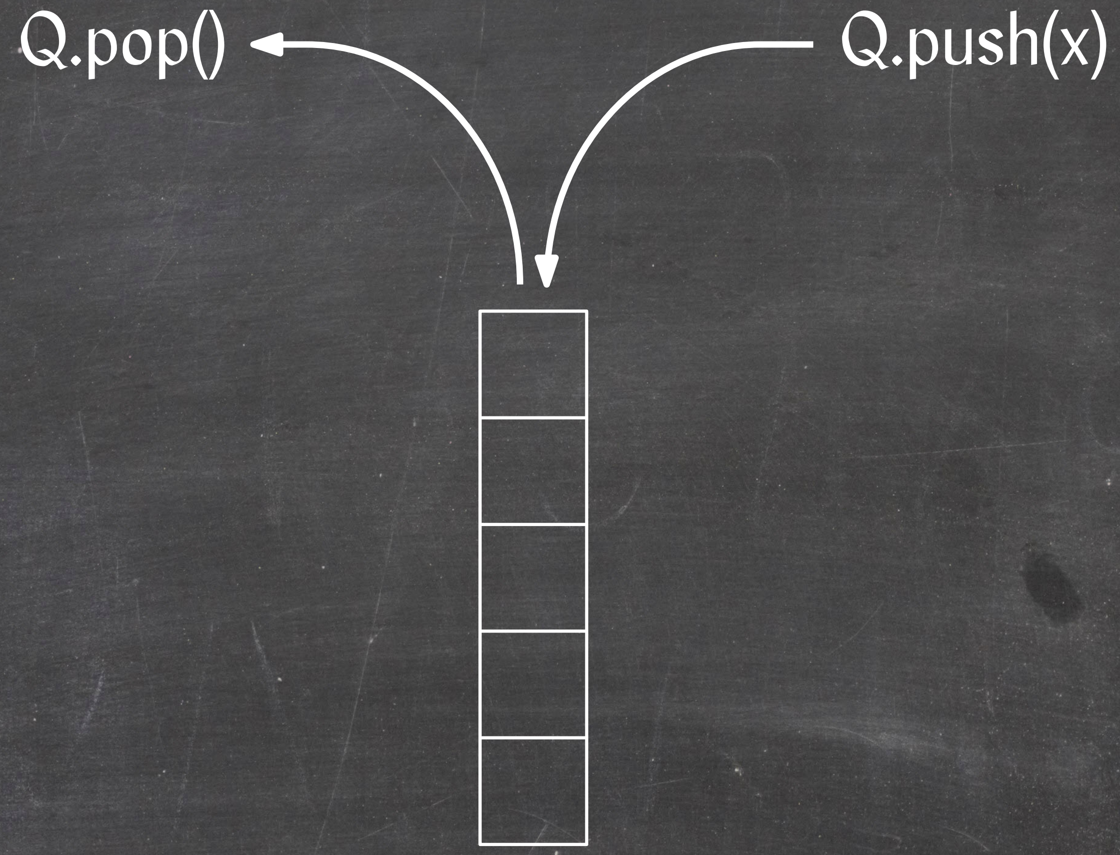
Constant-time implementations:

- Singly-linked list
- Resizeable array (amortized constant cost)

Depth-First Search

Depth-first search (DFS) = graph traversal using a stack to implement Q.

Stack:



Constant-time implementations:

- Singly-linked list
- Resizeable array (amortized constant cost)

Lemma: Depth-first search takes $O(n + m)$ time.

Depth-First Search and Preorder

Lemma: Depth-first search visits the vertices of the spanning forest it creates in preorder.

Depth-First Search and Preorder

Lemma: Depth-first search visits the vertices of the spanning forest it creates in preorder.

It visits the children of every node in left-to-right order.
(That's how we define this order.)

Depth-First Search and Preorder

Lemma: Depth-first search visits the vertices of the spanning forest it creates in preorder.

It visits the children of every node in left-to-right order.
(That's how we define this order.)

It visits every node after its parent:

- v is visited when the edge (parent(v), v) is popped.
- The edge (parent(v), v) must be pushed before this can happen.
- The edge (parent(v), v) is pushed when parent(v) is visited.

Depth-First Search and Preorder

Lemma: Depth-first search visits the vertices of the spanning forest it creates in preorder.

It visits the children of every node in left-to-right order.
(That's how we define this order.)

It visits every node after its parent:

- v is visited when the edge (parent(v), v) is popped.
- The edge (parent(v), v) must be pushed before this can happen.
- The edge (parent(v), v) is pushed when parent(v) is visited.

It visits the vertices in each subtree consecutively.

Depth-First Search and Preorder

Lemma: Depth-first search visits the vertices of the spanning forest it creates in preorder.

It visits the children of every node in left-to-right order.
(That's how we define this order.)

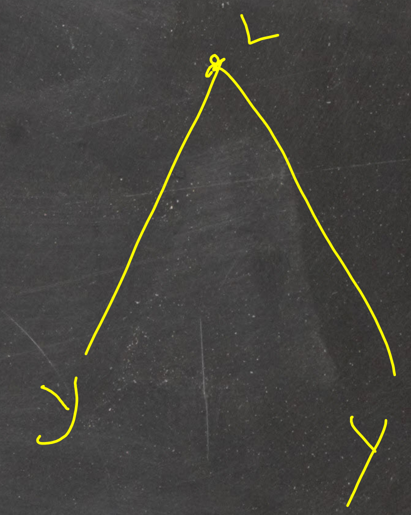
It visits every node after its parent:

- v is visited when the edge (parent(v), v) is popped.
- The edge (parent(v), v) must be pushed before this can happen.
- The edge (parent(v), v) is pushed when parent(v) is visited.

It visits the vertices in each subtree consecutively.

Observation: An edge with one explored and one unexplored endpoint is on the stack.

Depth-First Seach and Preorder



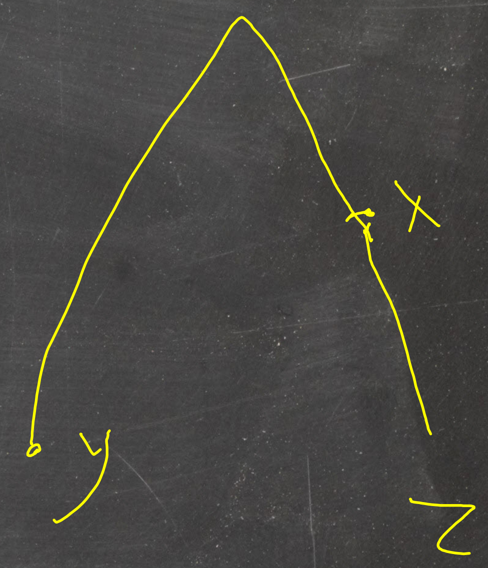
Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Depth-First Seach and Preorder



Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Case 1: y is a root.

Cannot happen because the edge (parent(z), z) is on the stack when y is visited and the stack is empty when a root is visited.

Depth-First Seach and Preorder

Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Case 2: y has a parent parent(y).

Depth-First Seach and Preorder

Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Case 2: y has a parent parent(y).

parent(y) is visited before x and thus before parent(z).

Depth-First Seach and Preorder

Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Case 2: y has a parent parent(y).

parent(y) is visited before x and thus before parent(z).

⇒ The edge (parent(y), y) is on the stack when parent(z) is visited and thus when the edge (parent(z), z) is pushed.

Depth-First Seach and Preorder

Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

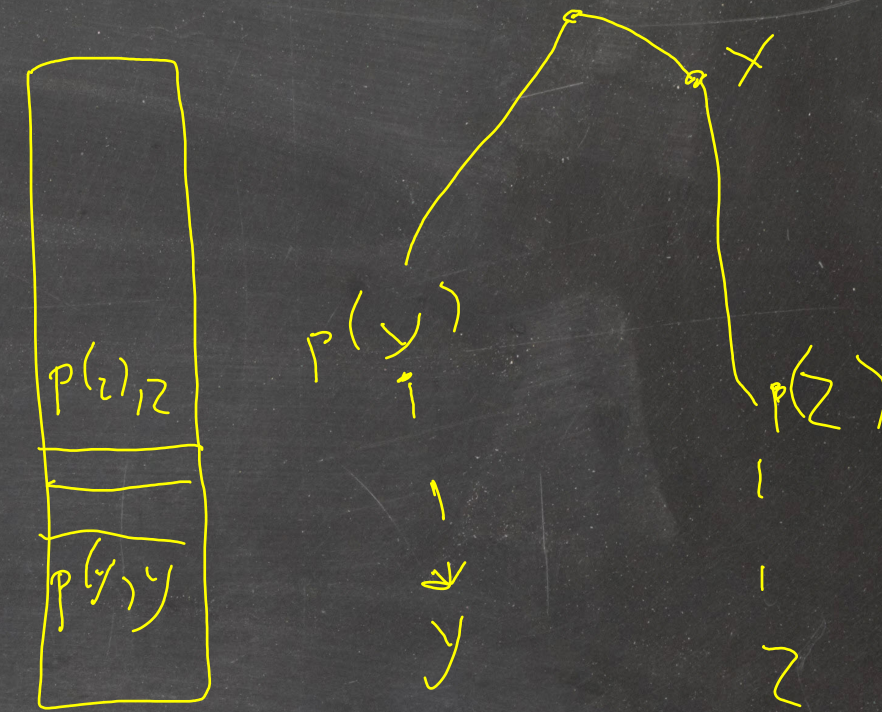
Case 2: y has a parent parent(y).

parent(y) is visited before x and thus before parent(z).

$\Rightarrow$ The edge (parent(y), y) is on the stack when parent(z) is visited and thus when the edge (parent(z), z) is pushed.

$\Rightarrow$ The edge (parent(z), z) is popped before the edge (parent(y), y).

Depth-First Seach and Preorder



Assume there exist two vertices x and y such that

- y is not a descendant of x,
- y is visited after x, and
- y is visited before some descendant z.

Choose y and z so that

- y is the first visited vertex satisfying the above conditions and
- y is visited after parent(z).

Case 2: y has a parent parent(y).

parent(y) is visited before x and thus before parent(z).

$\Rightarrow$ The edge (parent(y), y) is on the stack when parent(z) is visited and thus when the edge (parent(z), z) is pushed.

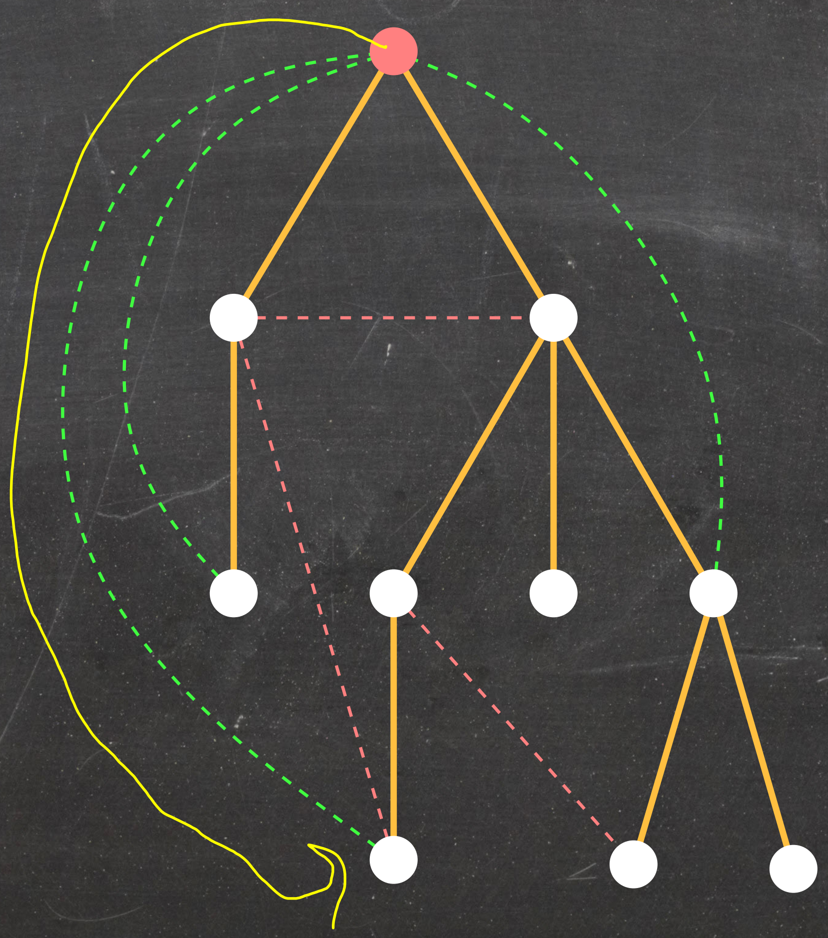
$\Rightarrow$ The edge (parent(z), z) is popped before the edge (parent(y), y).

$\Rightarrow$ z is visited before y, contradiction.

A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

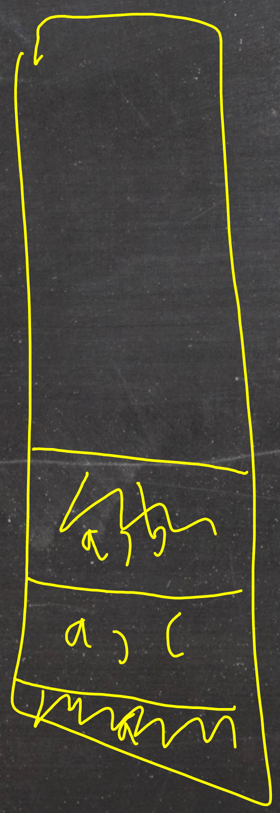


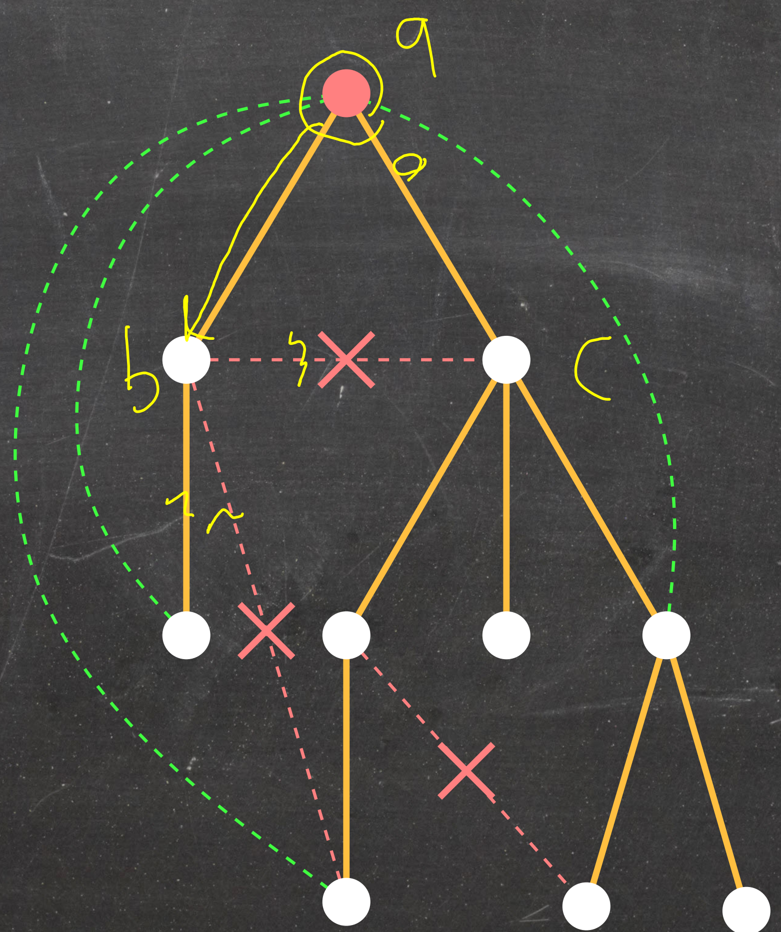
A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.





A Property of Undirected DFS Forests

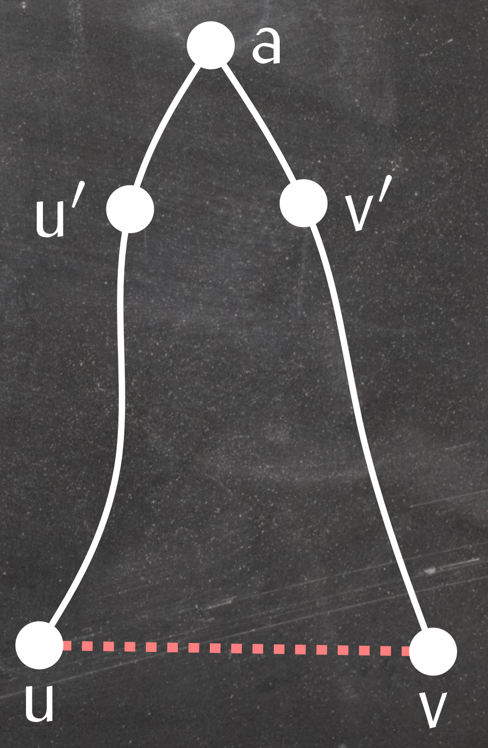
Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.

Assume $u < v$ in preorder.



A Property of Undirected DFS Forests

Three types of edges:

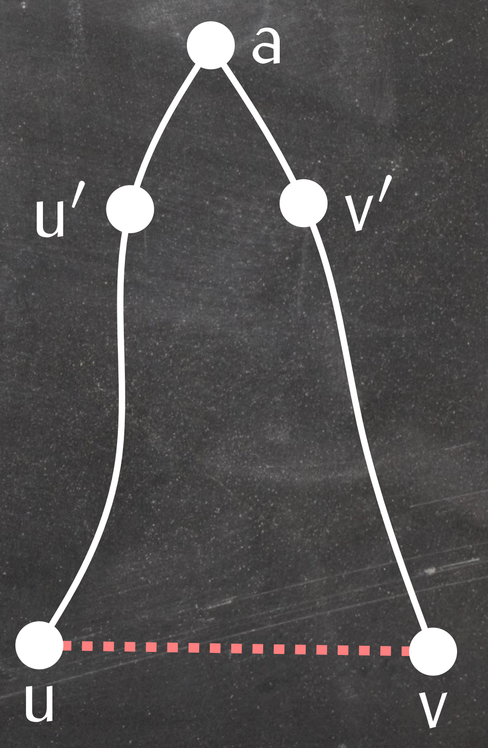
- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.

Assume $u < v$ in preorder.

$\Rightarrow$ Vertices a, u', u, v', v are visited in this order.



A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

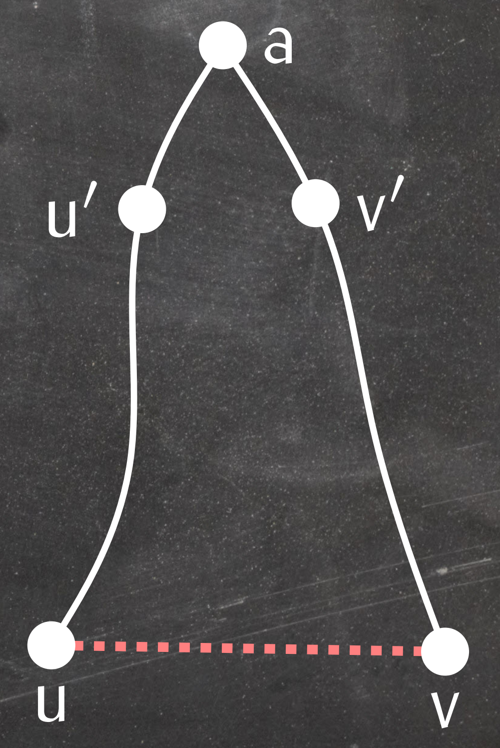
Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.

Assume $u < v$ in preorder.

$\Rightarrow$ Vertices a, u', u, v', v are visited in this order.

$\Rightarrow$ The edge (a, v') is pushed before u is visited and popped after u is visited.



A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

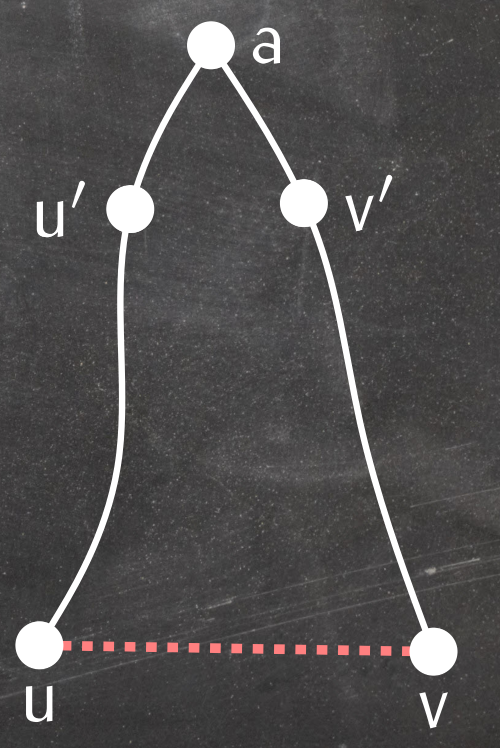
Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.

Assume $u < v$ in preorder.

$\Rightarrow$ Vertices a, u', u, v', v are visited in this order.

$\Rightarrow$ The edge (a, v') is pushed before u is visited and popped after u is visited.

$\Rightarrow$ The edge (u, v) is pushed after (a, v') is pushed and before (a, v') is popped.



A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- Cross edge (u, w): Neither u nor w is an ancestor of the other.
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.

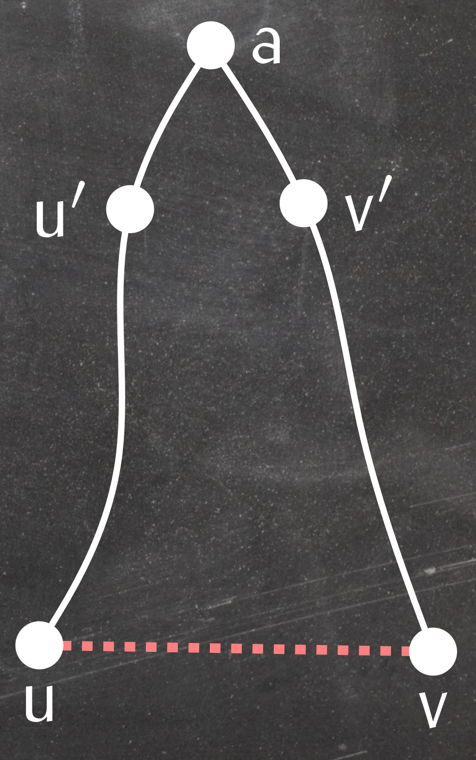
Assume $u < v$ in preorder.

$\Rightarrow$ Vertices a, u', u, v', v are visited in this order.

$\Rightarrow$ The edge (a, v') is pushed before u is visited and popped after u is visited.

$\Rightarrow$ The edge (u, v) is pushed after (a, v') is pushed and before (a, v') is popped.

$\Rightarrow$ The edge (u, v) is popped before (a, v') is popped.



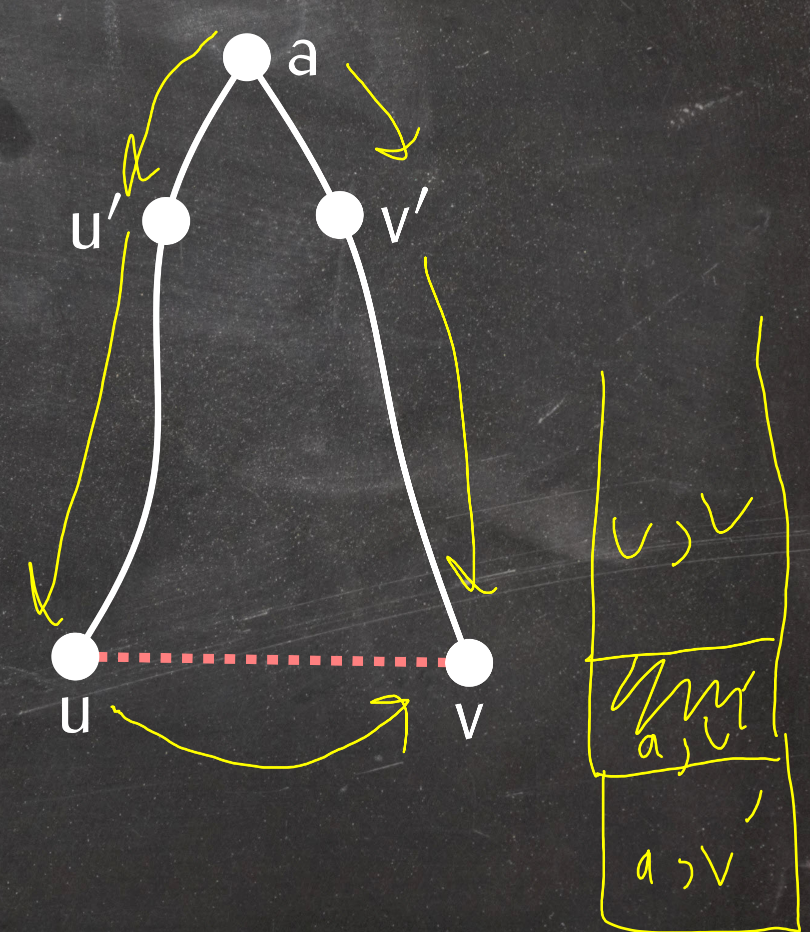
A Property of Undirected DFS Forests

Three types of edges:

- Tree edge (u, w): u is w's parent in F.
- ~~Cross edge (u, w): Neither u nor w is an ancestor of the other.~~
- Back edge (u, w): u is an ancestor of w but not its parent.

Lemma: All edges of an undirected graph G are tree or back edges with respect to a DFS forest of G.

Let a be the LCA of u and v and let u' and v' be the children of a that are ancestors of u and v.



Assume $u < v$ in preorder.

$\Rightarrow$ Vertices a, u', u, v', v are visited in this order.

$\Rightarrow$ The edge (a, v') is pushed before u is visited and popped after u is visited.

$\Rightarrow$ The edge (u, v) is pushed after (a, v') is pushed and before (a, v') is popped.

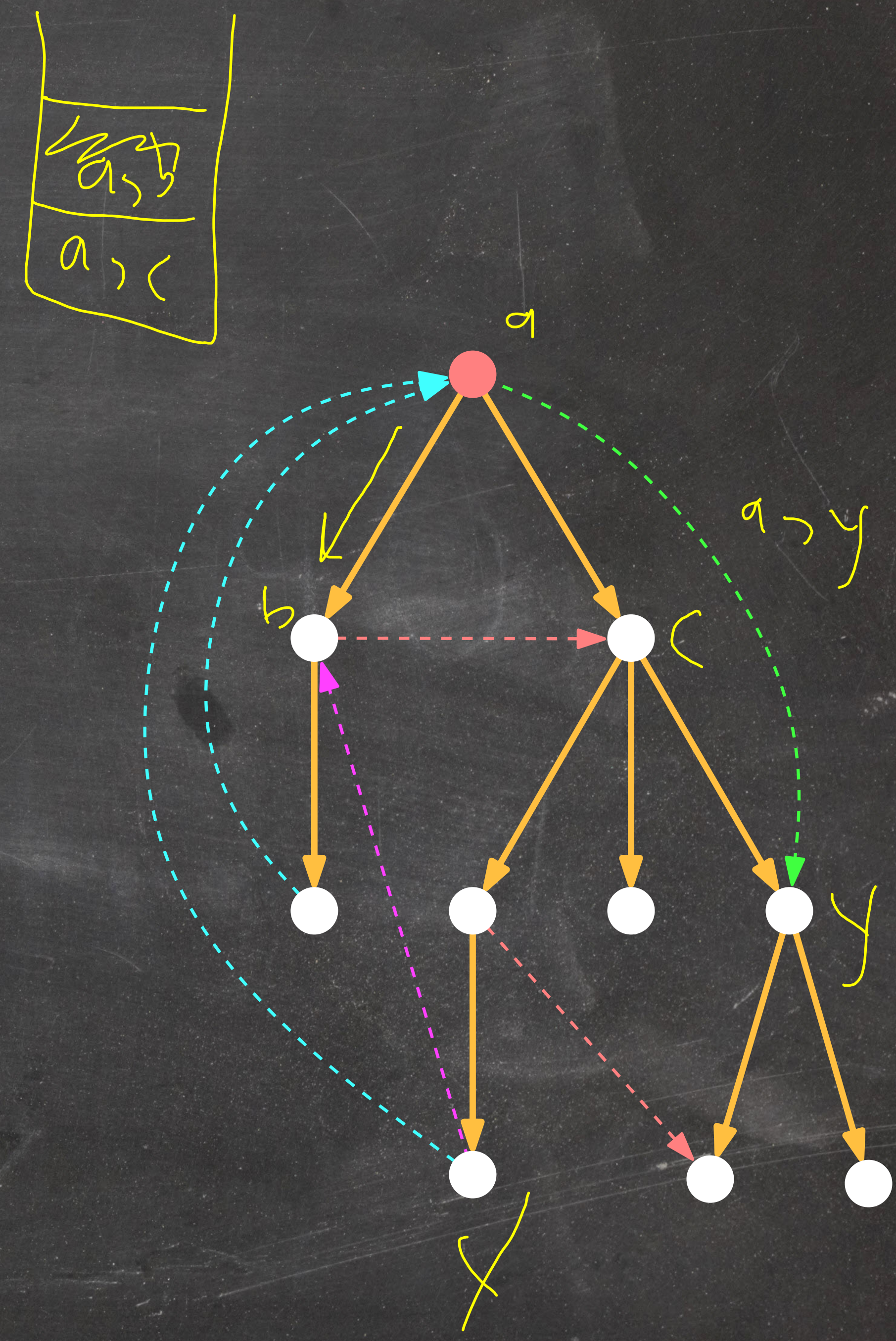
$\Rightarrow$ The edge (u, v) is popped before (a, v') is popped.

$\Rightarrow$ v is unexplored when the edge (u, v) is popped, a contradiction.

A Property of Directed DFS Forests

Five types of edges:

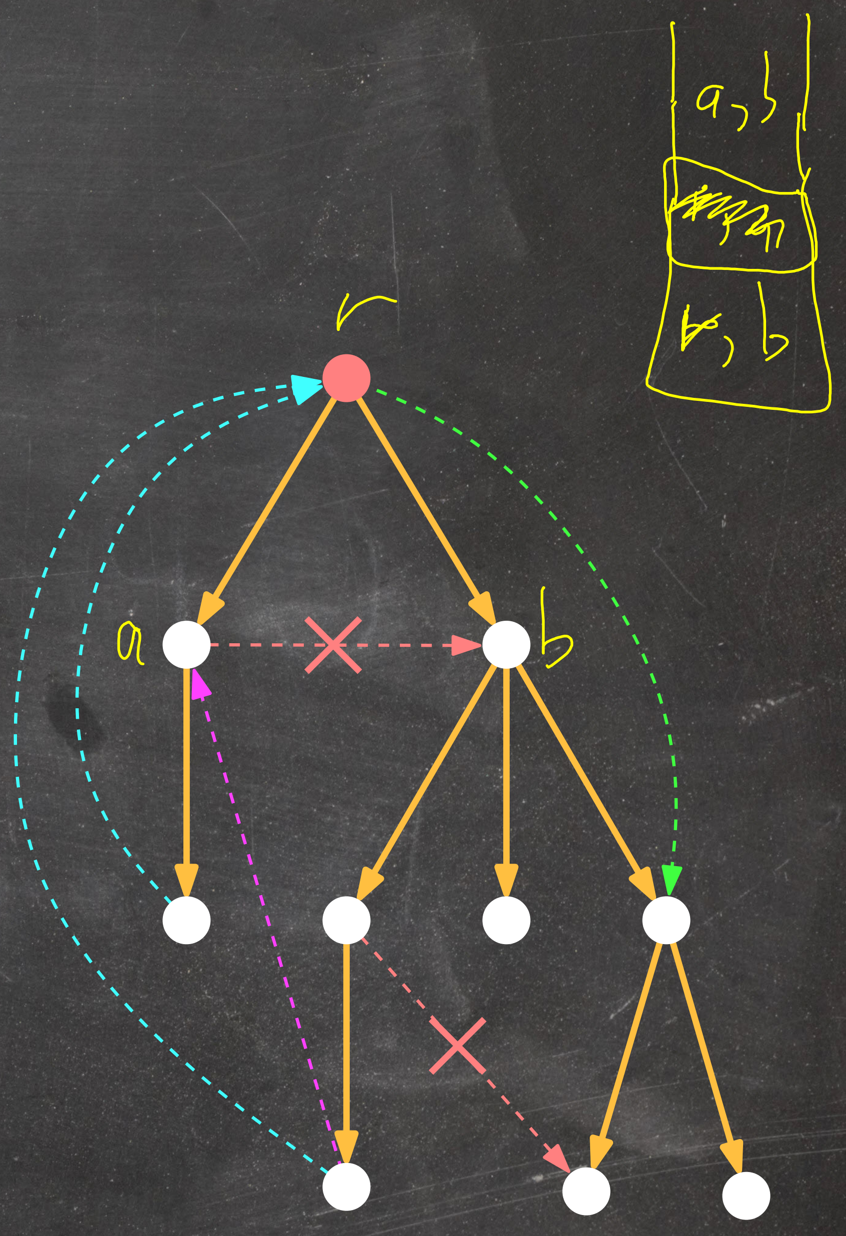
- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- Back edge (u, w): w is an ancestor of u.
- Forward cross edge (u, w): Neither u nor w is an ancestor of the other, $u < w$ in preorder/postorder.
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, $w < u$ in preorder/postorder.



A Property of Directed DFS Forests

Five types of edges:

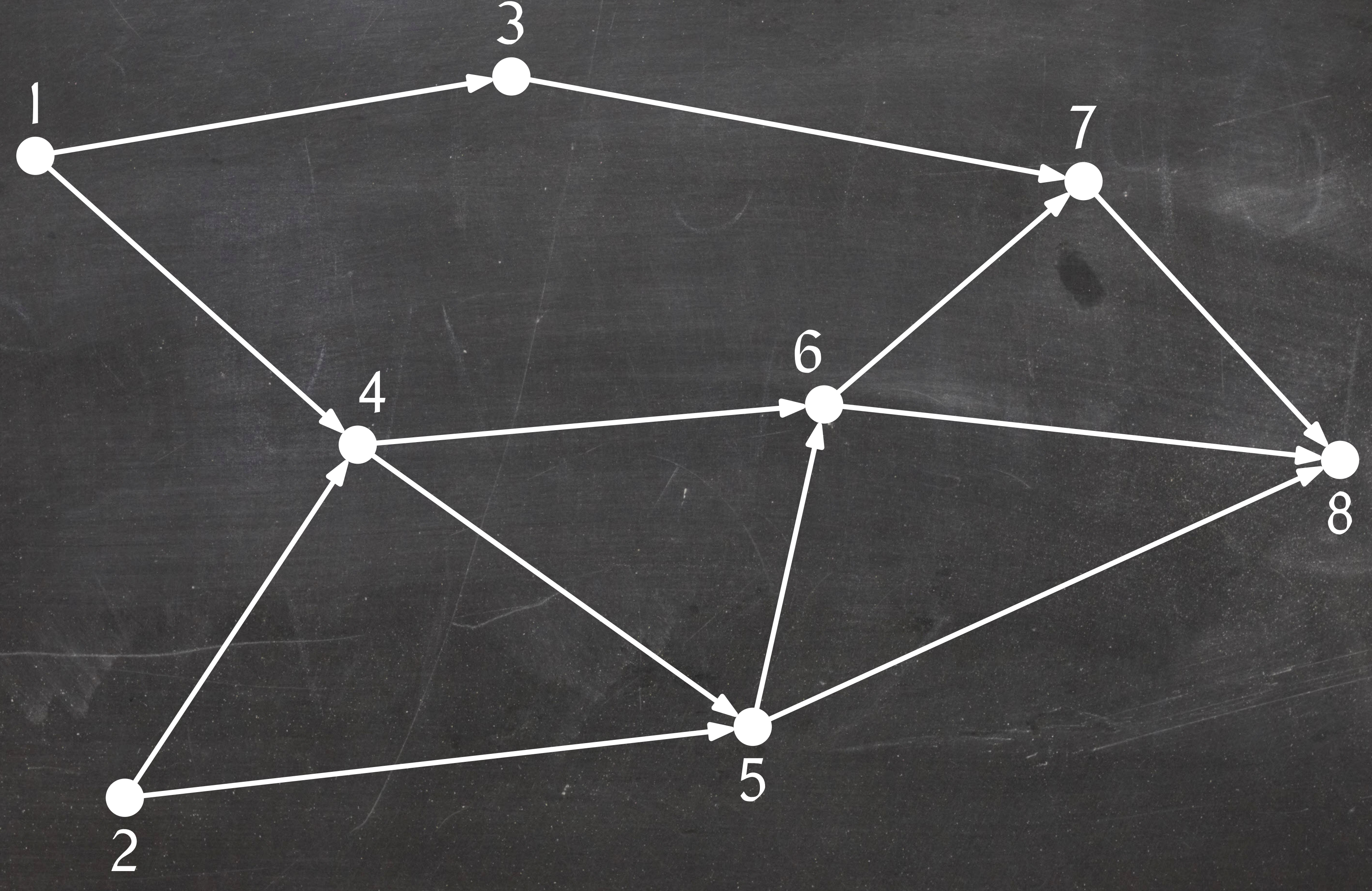
- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- Back edge (u, w): w is an ancestor of u.
- Forward cross edge (u, w): Neither u nor w is an ancestor of the other, u < w in preorder/postorder.
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, w < u in preorder/postorder.



Lemma: A directed graph G does not contain any forward cross edges with respect to a DFS forest of G.

Topological Sorting

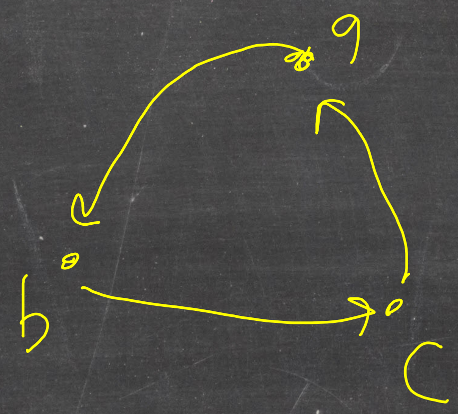
A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.



Topological Sorting

A topological ordering of a directed graph is an ordering < of the vertex set of G such that u < v for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

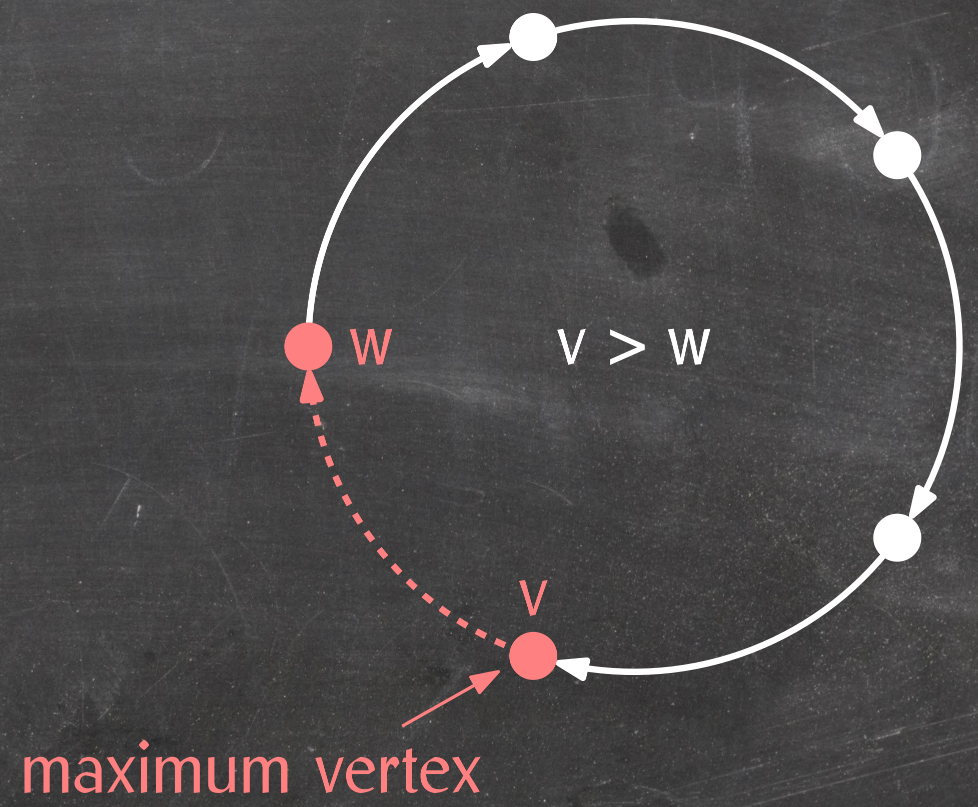


Topological Sorting

A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

If there's a cycle, there is no topological ordering.



Topological Sorting

A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

Topological Sorting

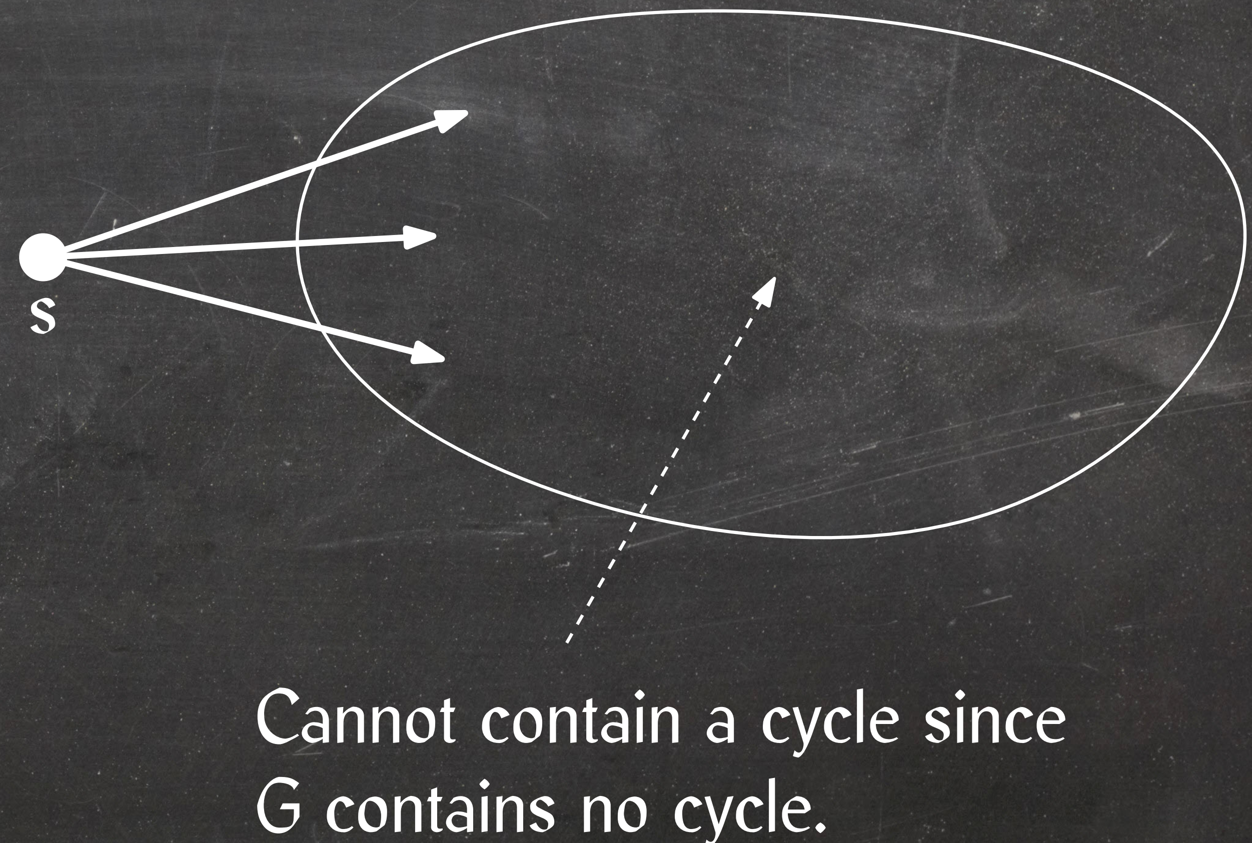
A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

$\Rightarrow$ The following algorithm produces a topological ordering:

- Give s the smallest number.
- Recursively number the rest of the vertices.



Topological Sorting

A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

Let $R(v)$ be the set of vertices reachable from v.

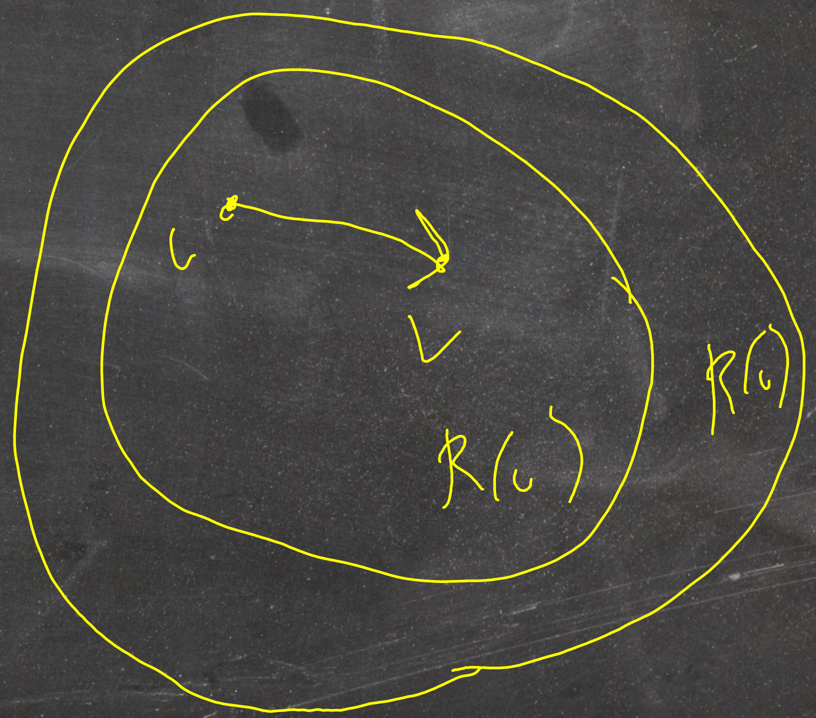
Topological Sorting

A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

Let $R(v)$ be the set of vertices reachable from v.



For an edge (u, v),

- $R(u) \supseteq R(v)$
- $u \in R(u)$
- $u \notin R(v)$ (otherwise there'd be a cycle)

$\Rightarrow$ $R(u) \supset R(v)$.

Topological Sorting

A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

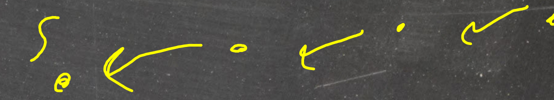
Let $R(v)$ be the set of vertices reachable from v.

For an edge (u, v),
- $R(u) \supseteq R(v)$
- $u \in R(u)$
- $u \notin R(v)$ (otherwise there'd be a cycle)

$\Rightarrow R(u) \supset R(v)$.

Pick a vertex s such that $|R(s)| \geq |R(v)|$ for all $v \in G$.

Topological Sorting



A topological ordering of a directed graph is an ordering $<$ of the vertex set of G such that $u < v$ for every edge $(u, v) \in G$.

Lemma: A graph G has a topological ordering if and only if it contains no directed cycle.

We prove that, if there is no cycle, there is always a source (vertex of in-degree 0).

Let $R(v)$ be the set of vertices reachable from v.

For an edge (u, v),

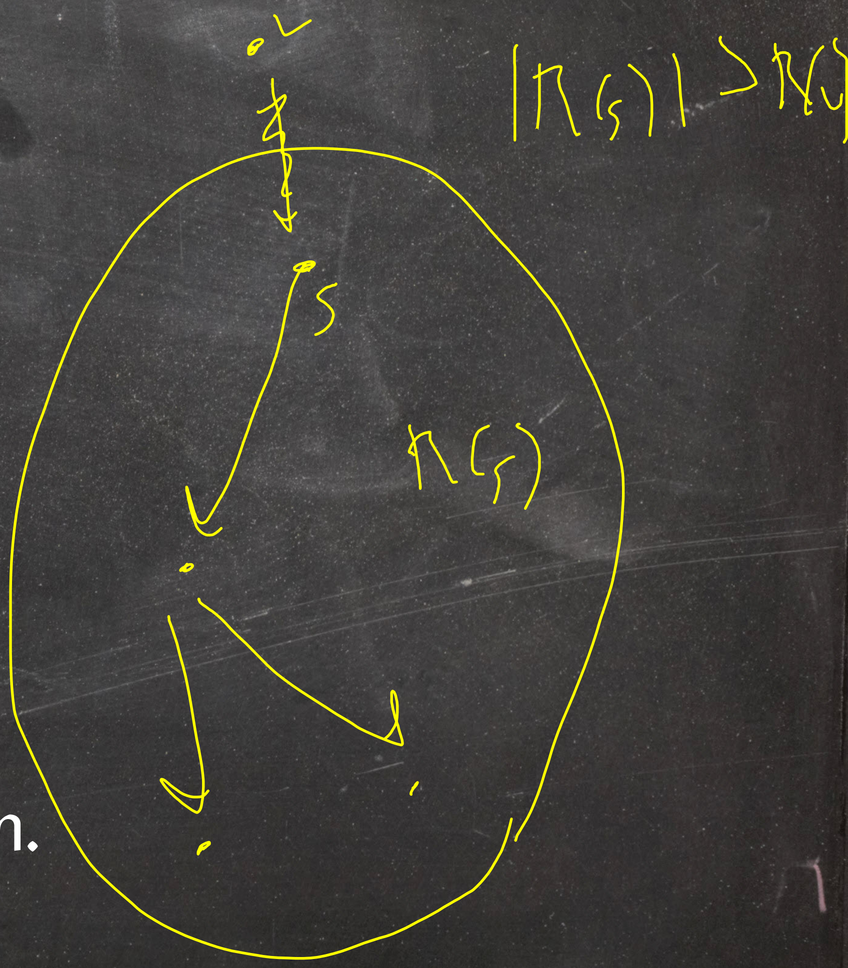
- $R(u) \supseteq R(v)$
- $u \in R(u)$
- $u \notin R(v)$ (otherwise there'd be a cycle)

$\Rightarrow$ $R(u) \supset R(v)$.

Pick a vertex s such that $|R(s)| \geq |R(v)|$ for all $v \in G$.

If s had an in-neighbour u, then $|R(u)| > |R(s)|$, a contradiction.

$\Rightarrow$ s is a source.

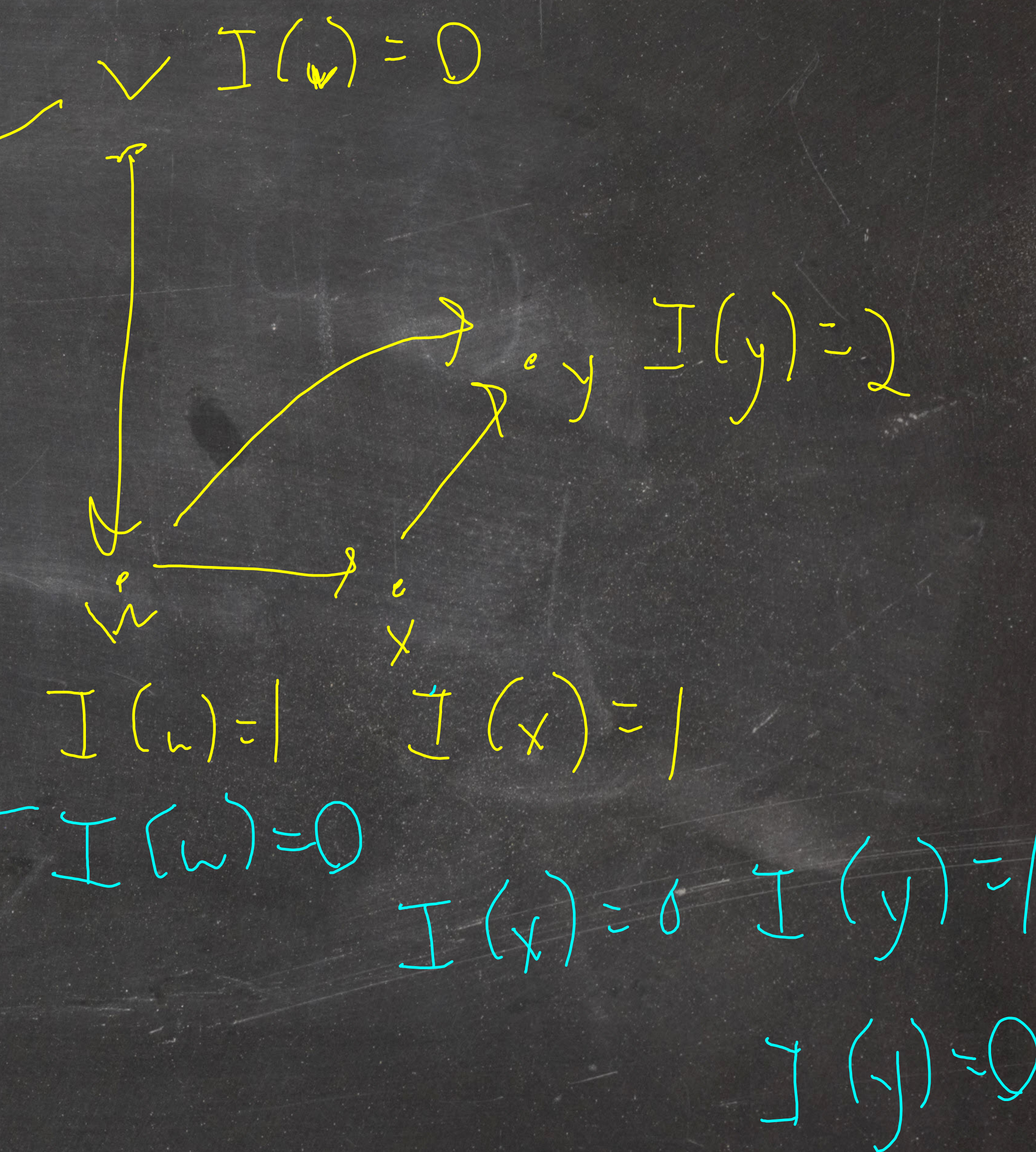


Topological Sorting

Lemma: A topological ordering of a directed acyclic graph G can be computed in $O(n + m)$ time.

SimpleTopSort(G)

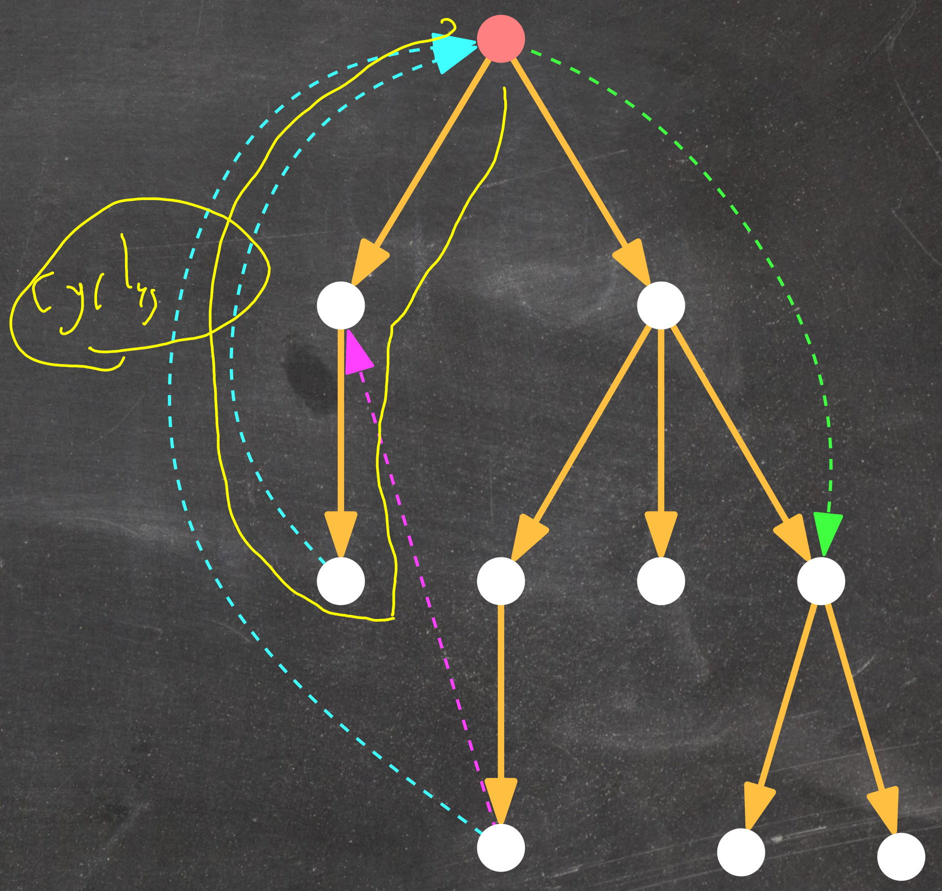
```
Q = an empty queue
for every vertex v ∈ G
    do label v with its in-degree
        if in-deg(v) = 0
            then Q.enqueue(v)
O = []
while not Q.isEmpty()
    do v = Q.dequeue()
        O.append(v)
        for every out-neighbour w of v
            do in-deg(w) = in-deg(w) − 1
                if in-deg(w) = 0
                    then Q.enqueue(w)
return O
```



Topological Sorting Using DFS

Edges in a DFS forest:

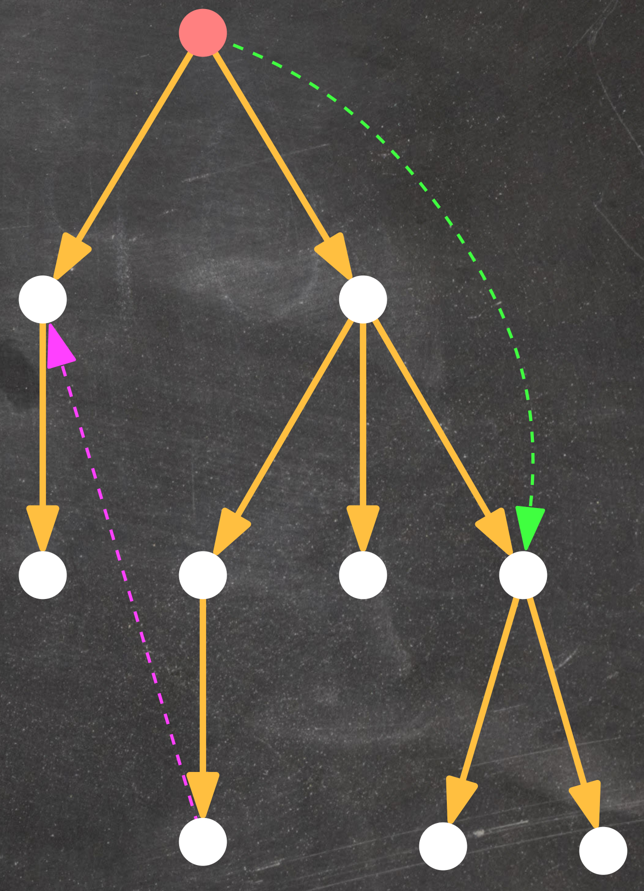
- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- Back edge (u, w): w is an ancestor of u.
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, w < u in postorder.



Topological Sorting Using DFS

Edges in a DFS forest:

- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- ~~Back edge (u, w): w is an ancestor of u.~~
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, w < u in postorder.

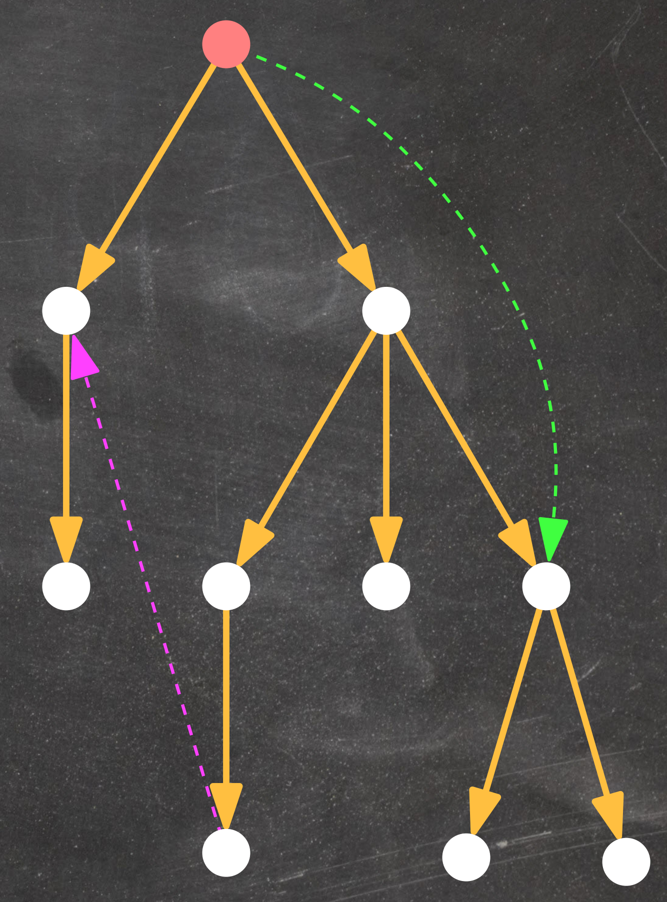


Topological Sorting Using DFS

Edges in a DFS forest:

- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- ~~Back edge (u, w): w is an ancestor of u.~~
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, w < u in postorder.

For tree, forward, and backward cross edges (u, v), u > v in postorder.



Topological Sorting Using DFS

Edges in a DFS forest:

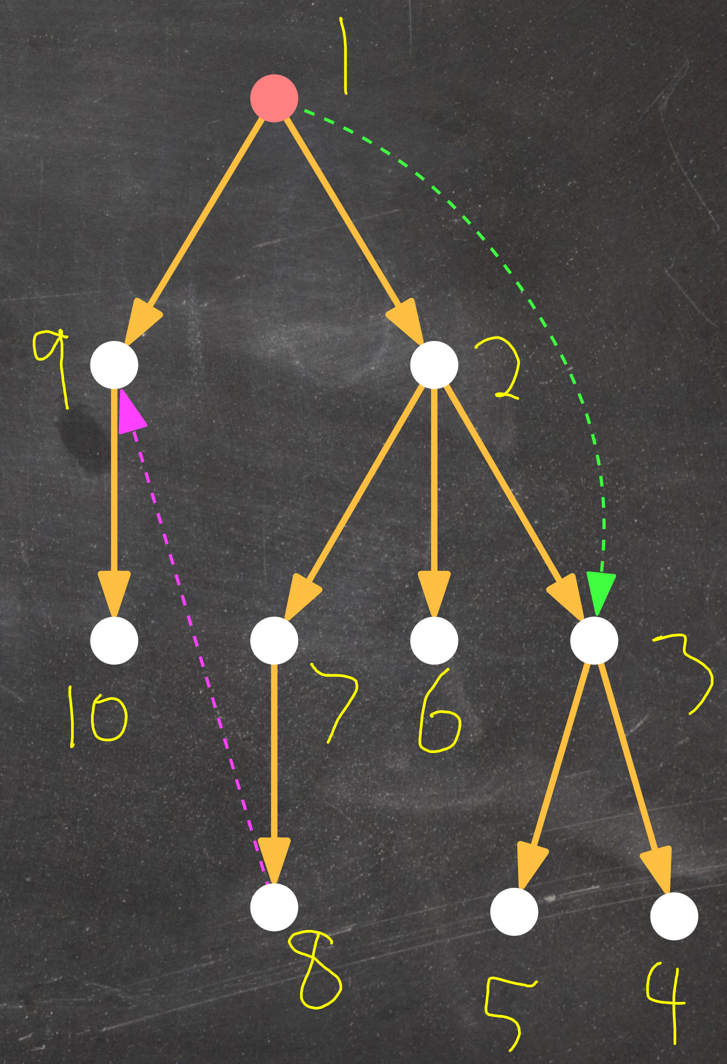
- Tree edge (u, w): u is w's parent in F.
- Forward edge (u, w): u is an ancestor of w.
- ~~Back edge (u, w): w is an ancestor of u.~~
- Backward cross edge (u, w): Neither u nor w is an ancestor of the other, w < u in postorder.

For tree, forward, and backward cross edges (u, v), u > v in postorder.

⇒ Topological sorting algorithm:

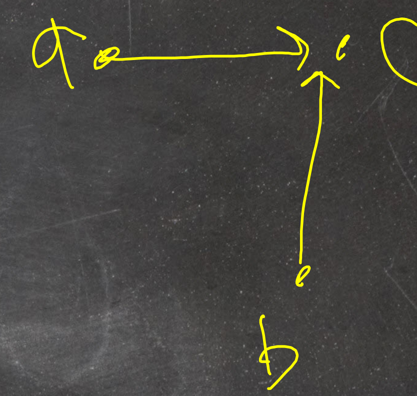
- Compute a DFS forest of G.
- Arrange the vertices in reverse postorder.

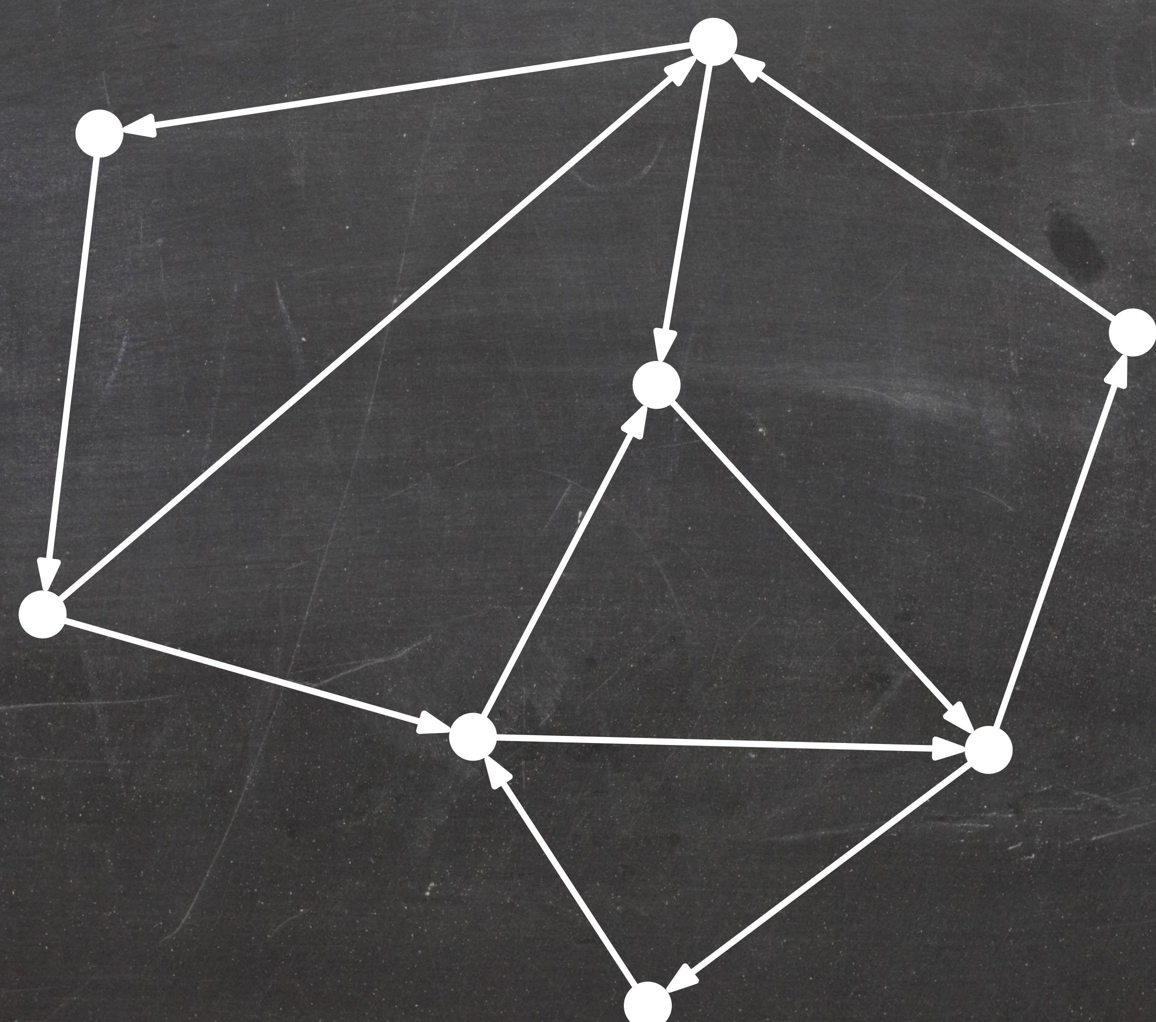
This takes $O(n + m)$ time.



Strongly Connected Components

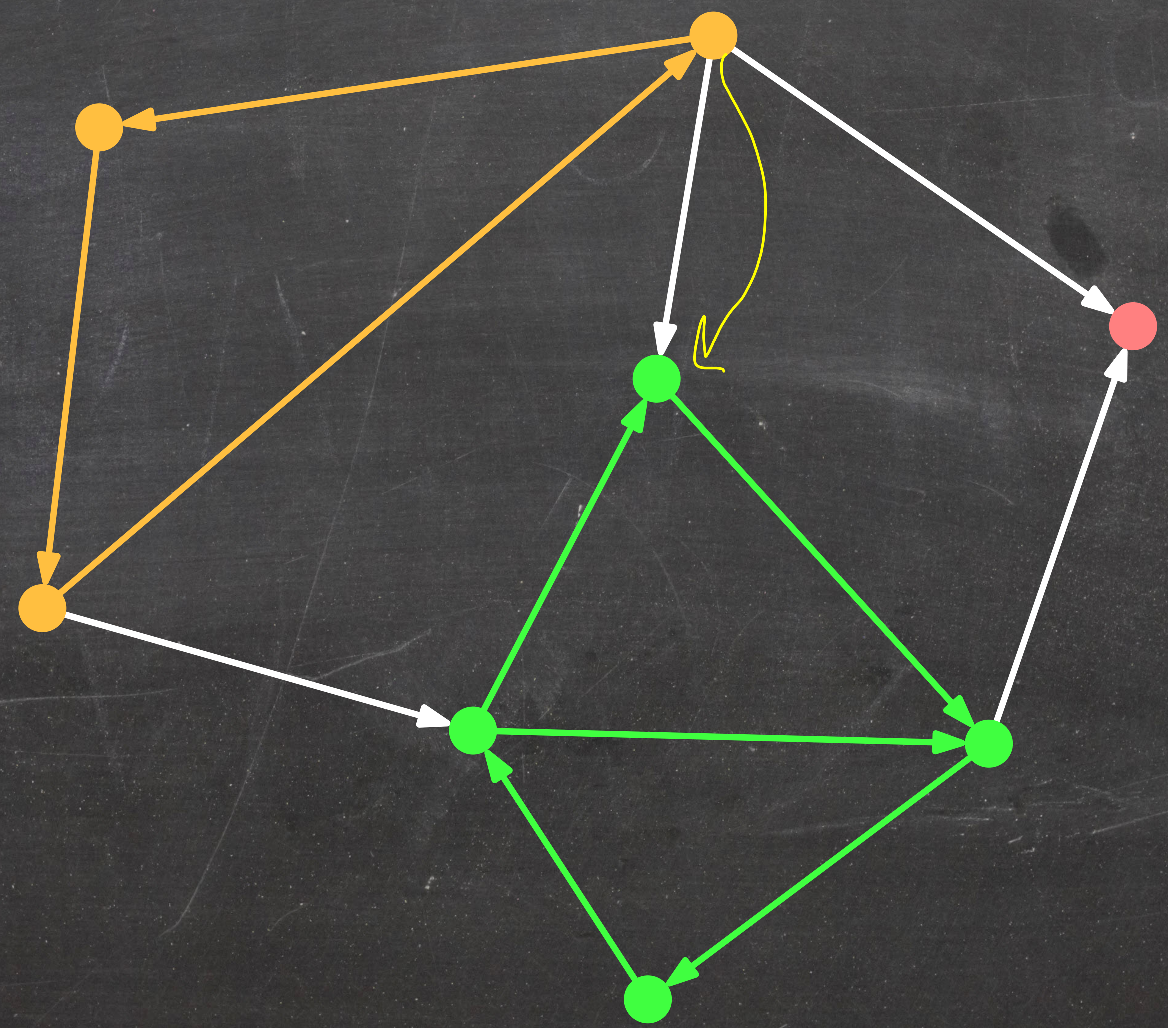
A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.





Strongly Connected Components

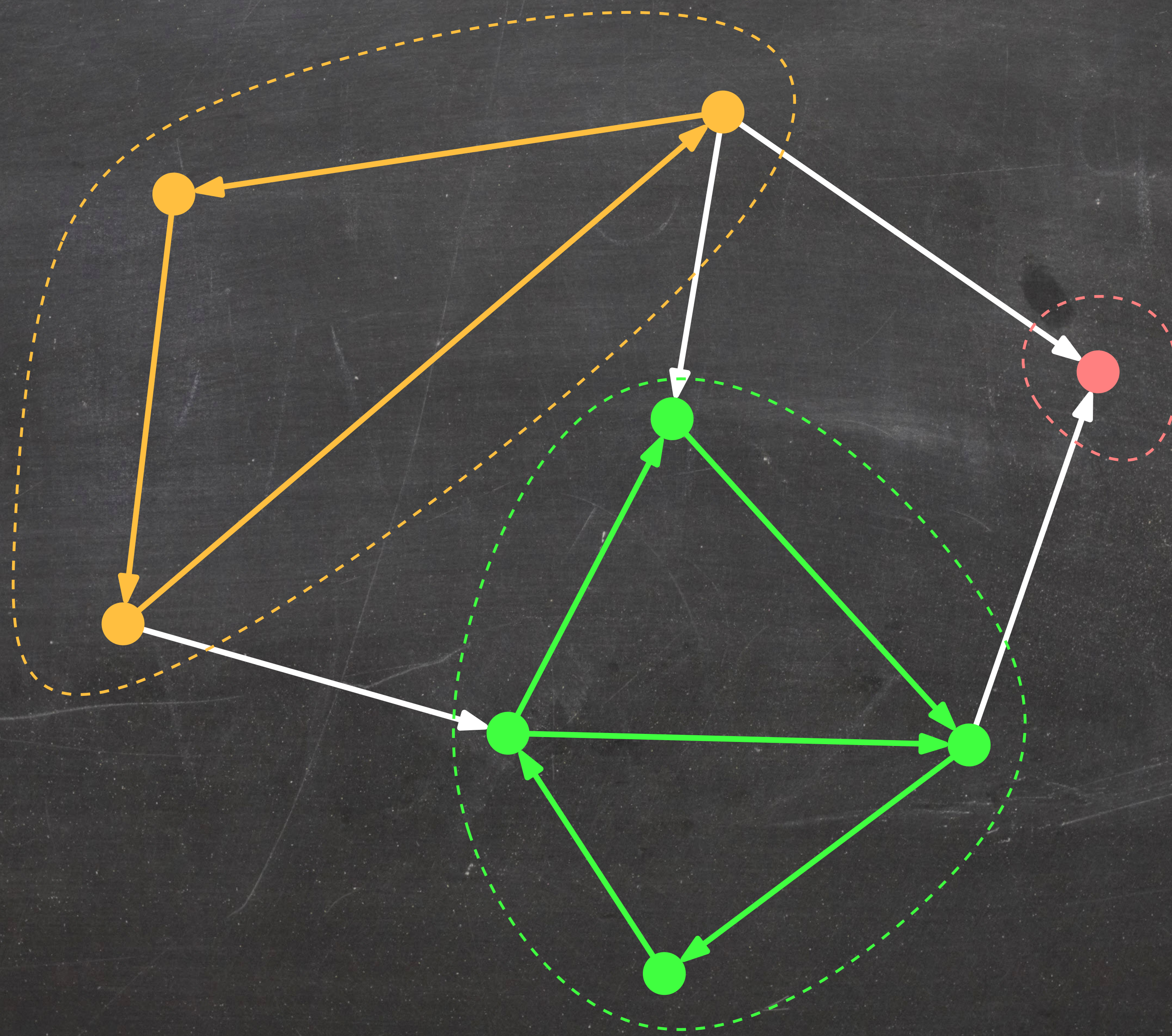
A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.



Strongly Connected Components

A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.

The strongly connected components of G are its maximal strongly connected subgraphs.



Strongly Connected Components

A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.

The strongly connected components of G are its maximal strongly connected subgraphs.

Lemma: For a DFS forest F of G and any two vertices u and w of G, $u \sim_{SCC(G)} w \Rightarrow u \sim_{CC(F)} w$. (The vertices of each strongly connected component of G belong to the same tree of any DFS forest F of G.)

Strongly Connected Components

A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.

The strongly connected components of G are its maximal strongly connected subgraphs.

Lemma: For a DFS forest F of G and any two vertices u and w of G, $u \sim_{SCC(G)} w \Rightarrow u \sim_{CC(F)} w$. (The vertices of each strongly connected component of G belong to the same tree of any DFS forest F of G.)

Let C be the strongly connected component containing u and w and let x be the first vertex in C visited during the construction of F.

Strongly Connected Components

A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.

The strongly connected components of G are its maximal strongly connected subgraphs.

Lemma: For a DFS forest F of G and any two vertices u and w of G, $u \sim_{SCC(G)} w \Rightarrow u \sim_{CC(F)} w$. (The vertices of each strongly connected component of G belong to the same tree of any DFS forest F of G.)

Let C be the strongly connected component containing u and w and let x be the first vertex in C visited during the construction of F.

It suffices to prove that $x \sim_{CC(F)} v$ for every $v \in C$.

Strongly Connected Components

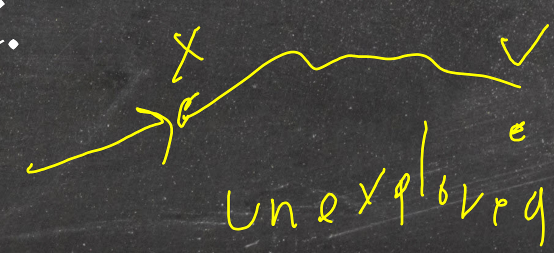
A graph is strongly connected if there exists a path from u to w and from w to u for every pair of vertices $u, w \in G$.

The strongly connected components of G are its maximal strongly connected subgraphs.

Lemma: For a DFS forest F of G and any two vertices u and w of G, $u \sim_{SCC(G)} w \Rightarrow u \sim_{CC(F)} w$. (The vertices of each strongly connected component of G belong to the same tree of any DFS forest F of G.)

Let C be the strongly connected component containing u and w and let x be the first vertex in C visited during the construction of F.

It suffices to prove that $x \sim_{CC(F)} v$ for every $v \in C$.



This follows from

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Let $P = \langle x = x_0, x_1, \ldots, x_k = v \rangle$ be such a path from x to v and assume v is not a descendant of x.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Let $P = \langle x = x_0, x_1, \ldots, x_k = v \rangle$ be such a path from x to v and assume v is not a descendant of x.

Since x is a descendant of x, there exists a maximal index $0 \leq i < k$ such that $x_0, x_1, \ldots, x_i$ are descendants of x and x_{i+1} is not.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Let $P = \langle x = x_0, x_1, \ldots, x_k = v \rangle$ be such a path from x to v and assume v is not a descendant of x.

Since x is a descendant of x, there exists a maximal index $0 \leq i < k$ such that $x_0, x_1, \ldots, x_i$ are descendants of x and x_{i+1} is not.

Since x_{i+1} is visited after x and all descendants of x have consecutive preorder numbers, we have $x_i < x_{i+1}$ in preorder.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Let $P = \langle x = x_0, x_1, \ldots, x_k = v \rangle$ be such a path from x to v and assume v is not a descendant of x.

Since x is a descendant of x, there exists a maximal index $0 \leq i < k$ such that $x_0, x_1, \ldots, x_i$ are descendants of x and x_{i+1} is not.

Since x_{i+1} is visited after x and all descendants of x have consecutive preorder numbers, we have $x_i < x_{i+1}$ in preorder.

Since x_{i+1} is no descendant of x, it is not a descendant of x_i.

Strongly Connected Components

Lemma: If there exists a path from x to v consisting of vertices that are unexplored when x is visited, then v is a descendant of x in F.

Let $P = \langle x = x_0, x_1, \ldots, x_k = v \rangle$ be such a path from x to v and assume v is not a descendant of x.

Since x is a descendant of x, there exists a maximal index $0 \leq i < k$ such that $x_0, x_1, \ldots, x_i$ are descendants of x and x_{i+1} is not.

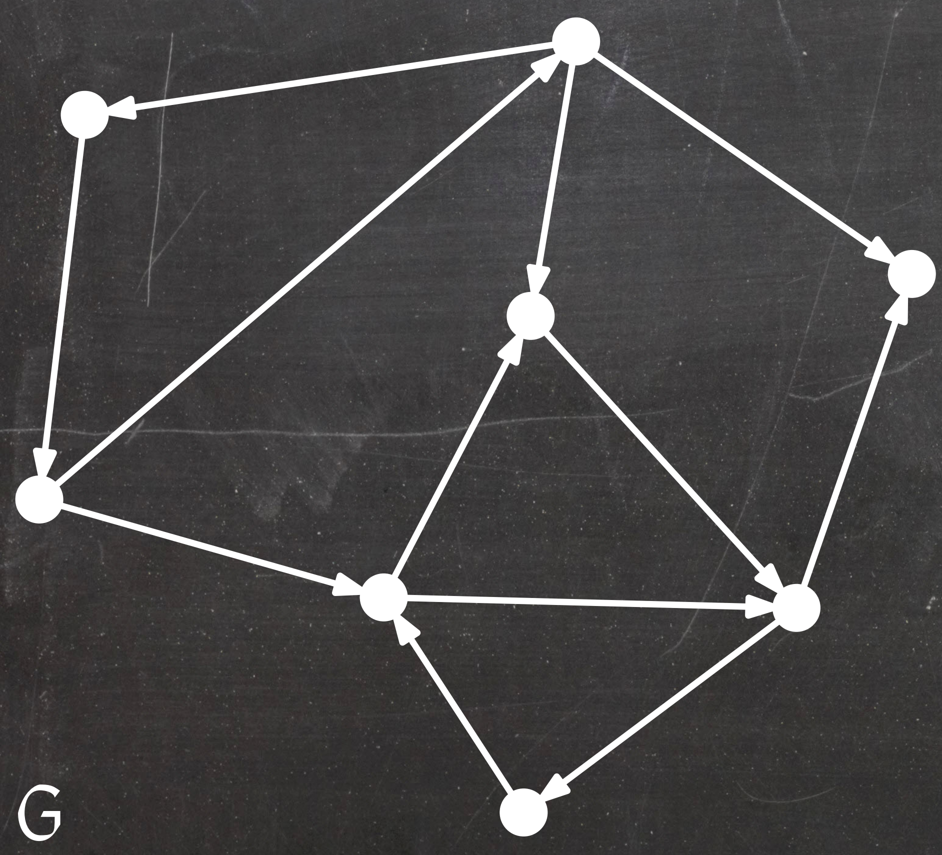
Since x_{i+1} is visited after x and all descendants of x have consecutive preorder numbers, we have $x_i < x_{i+1}$ in preorder.

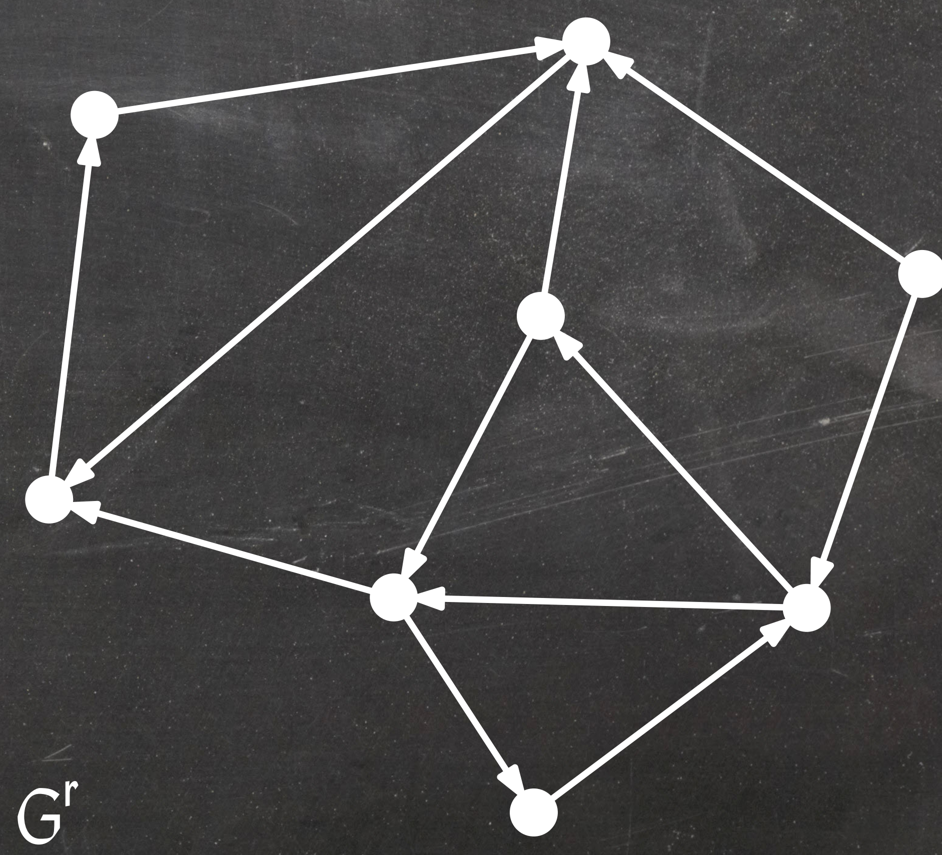
Since x_{i+1} is no descendant of x, it is not a descendant of x_i.

Since $x_i < x_{i+1}$ in preorder, this implies that (x_i, x_{i+1}) is a forward cross edge, a contradiction.

Strongly Connected Components

For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

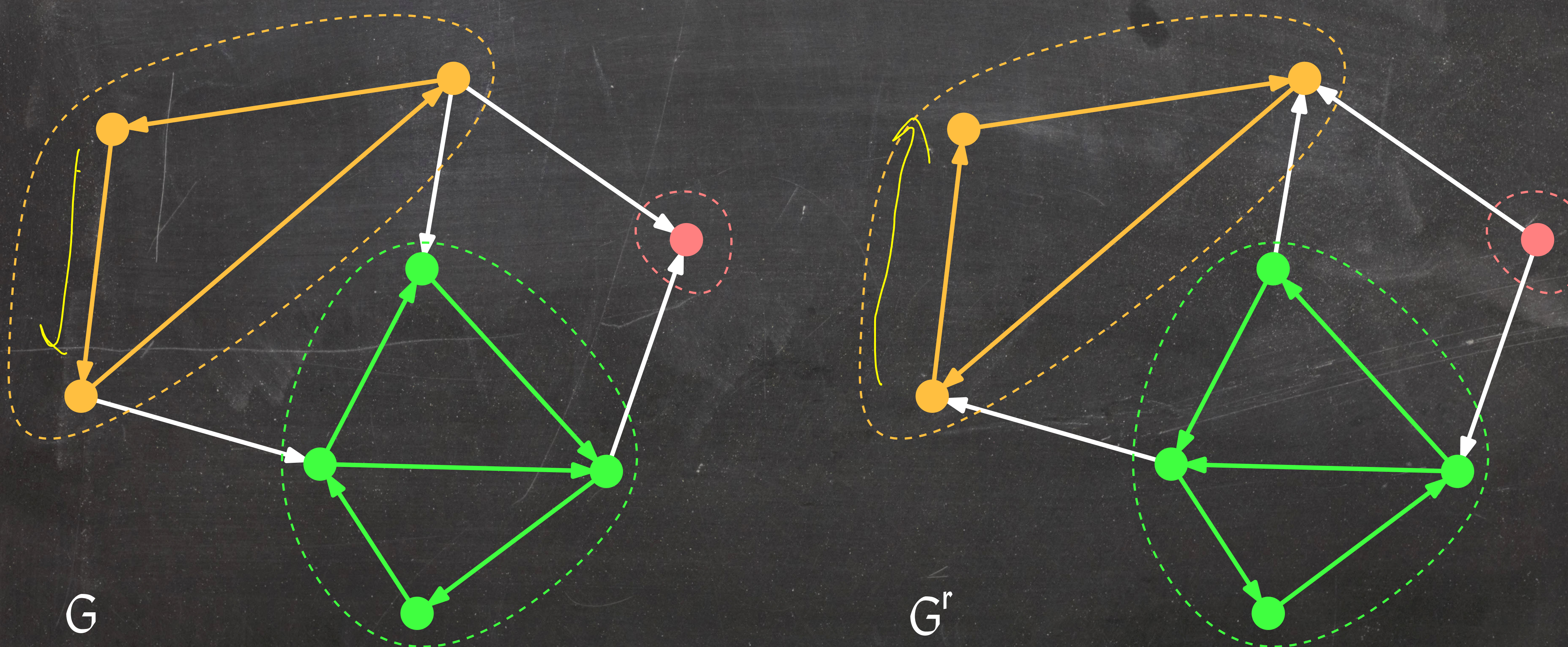




Strongly Connected Components

For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

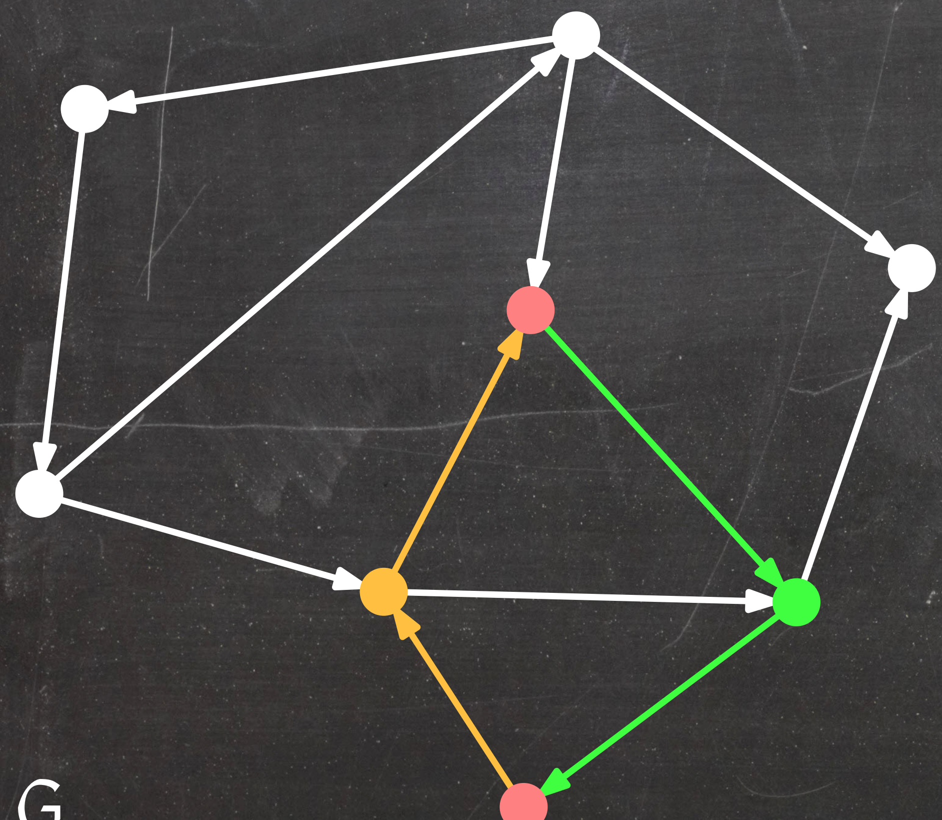


Strongly Connected Components

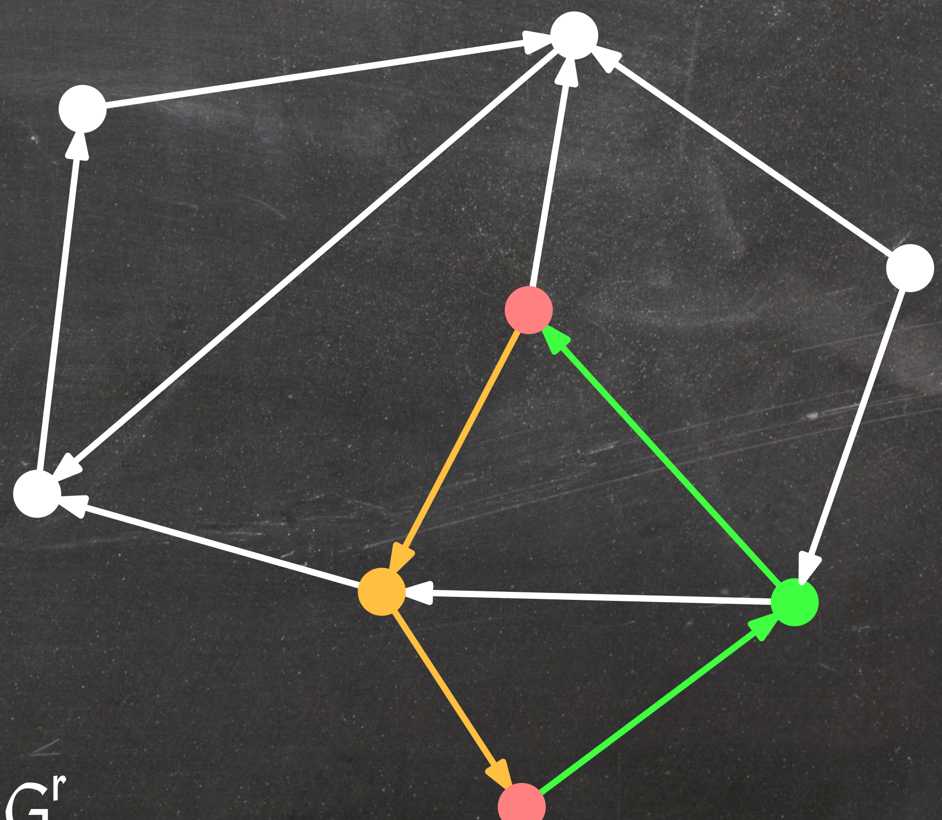
For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

Proof: We have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^r} u$.



G



G^r

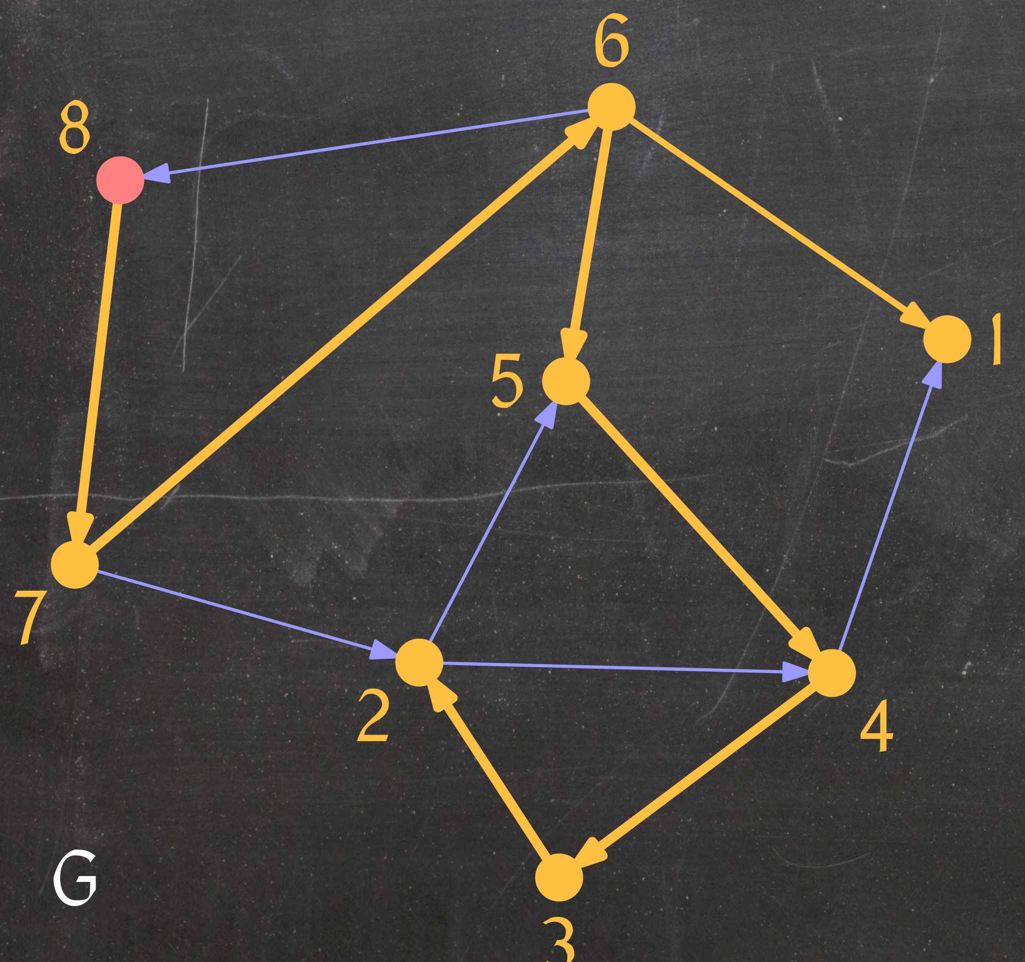
Strongly Connected Components

For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

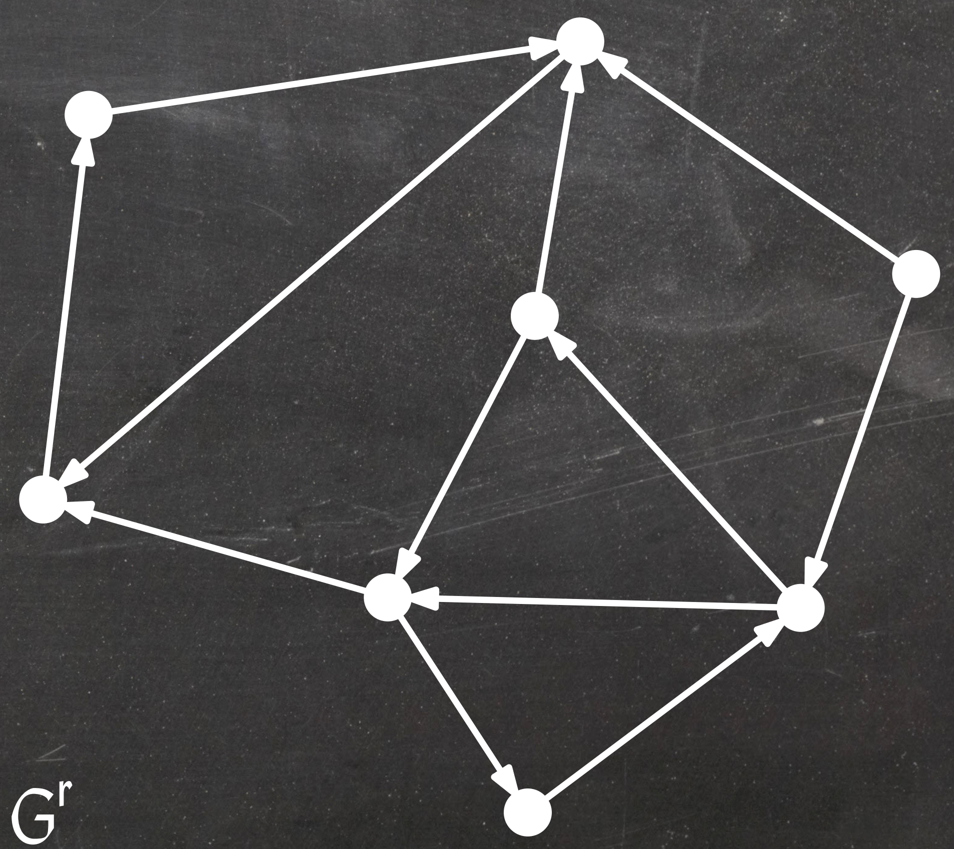
Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

Proof: We have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^r} u$.

Let F be a DFS forest of G and let $<$ be the postorder of F.



G



G^r

Strongly Connected Components

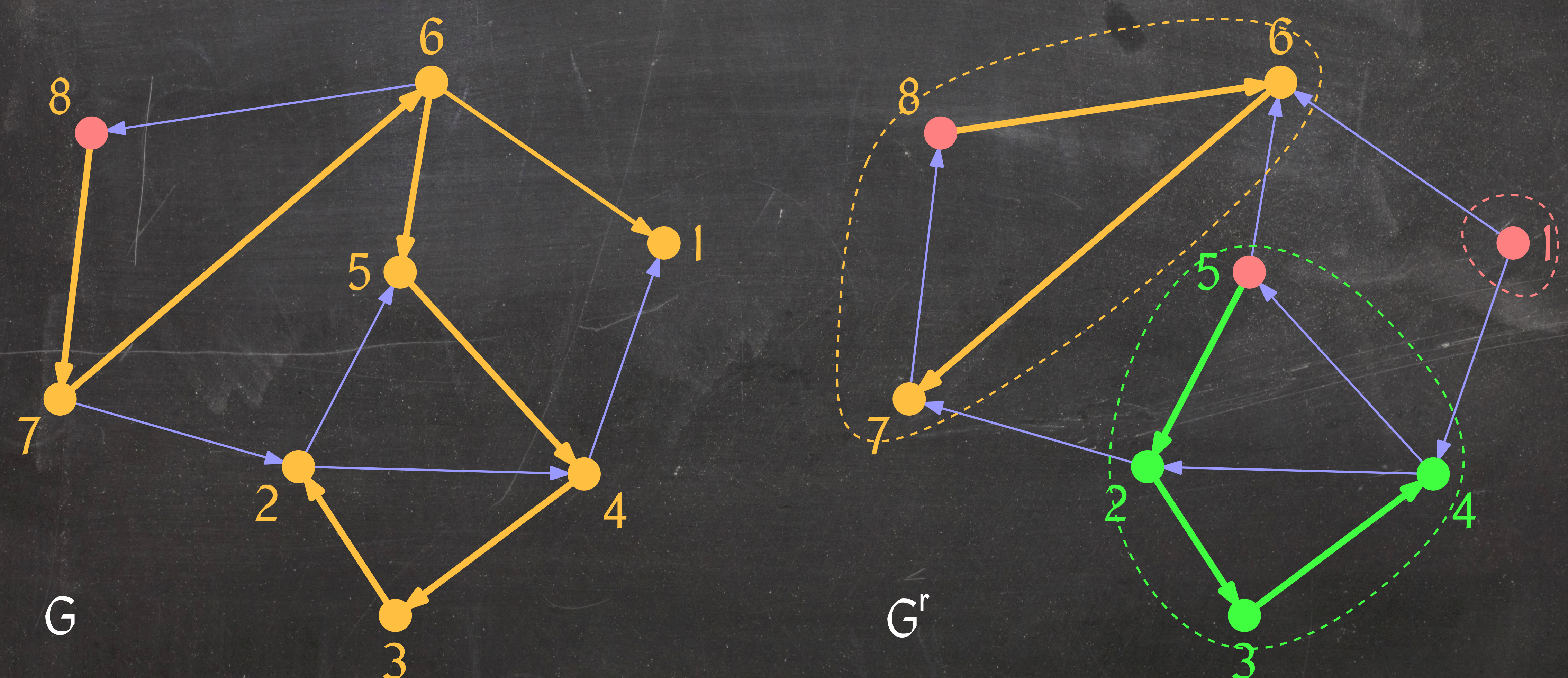
For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

Proof: We have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^r} u$.

Let F be a DFS forest of G and let $<$ be the postorder of F.

Let $F^r_>$ be the DFS forest of G^r obtained by calling TraverseFromVertex on unexplored vertices in the opposite order to $<$.



Strongly Connected Components

For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

Proof: We have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^r} u$.

Let F be a DFS forest of G and let $<$ be the postorder of F.

Let $F^r_>$ be the DFS forest of G^r obtained by calling TraverseFromVertex on unexplored vertices in the opposite order to $<$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Strongly Connected Components

For a graph $G = (V, E)$, let $G^r = (V, E^r)$, where $E^r = \{(v, u) \mid (u, v) \in E\}$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{SCC(G^r)} v$.

Proof: We have $u \rightsquigarrow_G v$ if and only if $v \rightsquigarrow_{G^r} u$.

Let F be a DFS forest of G and let $<$ be the postorder of F.

Let $F^r_>$ be the DFS forest of G^r obtained by calling TraverseFromVertex on unexplored vertices in the opposite order to $<$.

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

$\Rightarrow$ Kosaraju's strong connectivity algorithm:

- Compute a DFS forest F of G.
- Compute G^r and arrange the vertices in reverse postorder w.r.t. F.
- Compute a DFS forest F^r of G^r.
- Extract a component labelling of the vertices or the strongly connected components themselves from F^r (almost) as we did for computing connected components.

This takes $O(n + m)$ time.

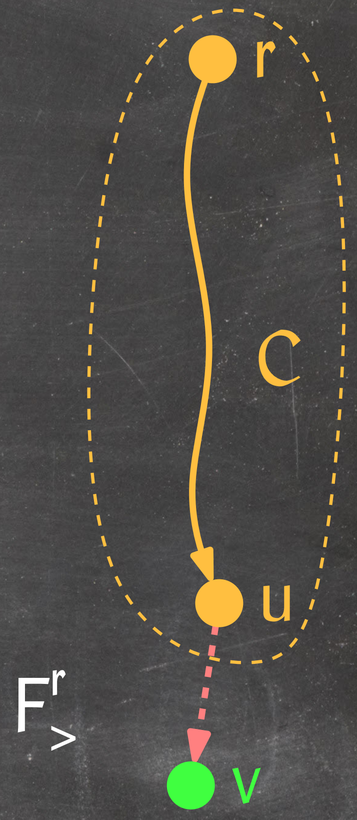
Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow (v, u) \in G$.

Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow (v, u) \in G$.

Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

Strongly Connected Components

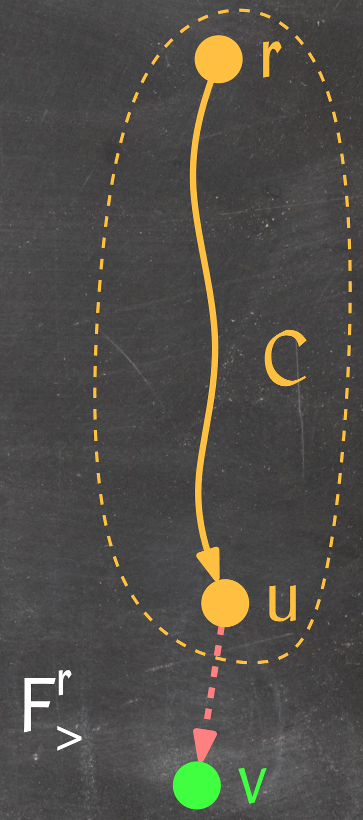
Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow$ $(v, u) \in G$.

Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

In particular, $u \sim_{SCC(G)} r$, where r is the root of the tree containing u and v.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow (v, u) \in G$.

Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

In particular, $u \sim_{SCC(G)} r$, where r is the root of the tree containing u and v.

All vertices in C are descendants of r in $F^r_>$ and $x \leq r$ for all $x \in C$.

Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow (v, u) \in G$.

Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

In particular, $u \sim_{SCC(G)} r$, where r is the root of the tree containing u and v.

All vertices in C are descendants of r in $F^r_>$ and $x \leq r$ for all $x \in C$.

Also, $v < r$ because v is a descendant of r in $F^r_>$.

Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow (v, u) \in G$.

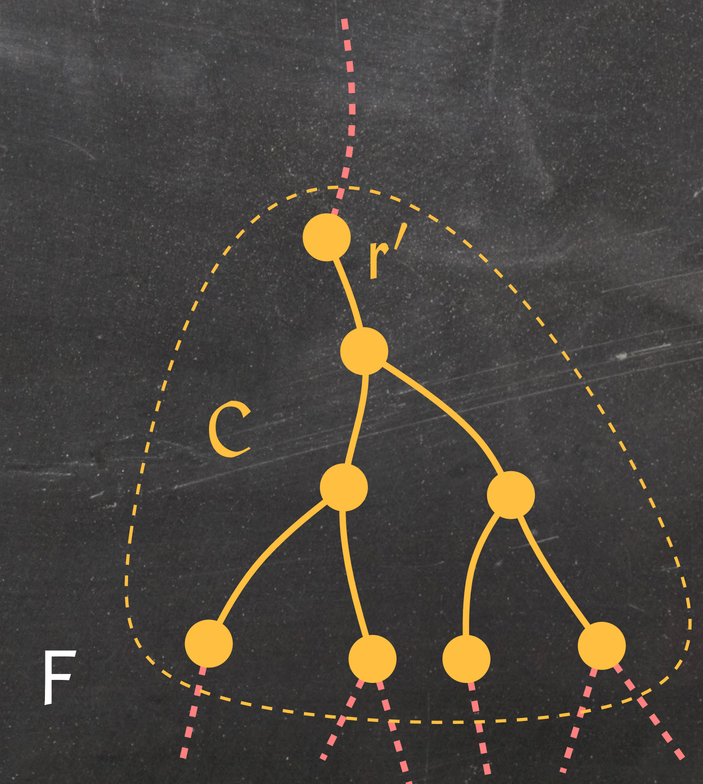
Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

In particular, $u \sim_{SCC(G)} r$, where r is the root of the tree containing u and v.

All vertices in C are descendants of r in $F^r_>$ and $x \leq r$ for all $x \in C$.

Also, $v < r$ because v is a descendant of r in $F^r_>$.

In F, all vertices in C are descendants of some vertex $r' \in C$ and $x \leq r'$ for all $x \in C$.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F^r_>)} v$.

Assume the contrary. Then there exists an edge $(u, v) \in F^r_>$ such that $u \not\sim_{SCC(G)} v$.

$\Rightarrow$ $(v, u) \in G$.

Choose this edge so that each of its ancestor edges (x, y) satisfies $x \sim_{SCC(G)} y$.

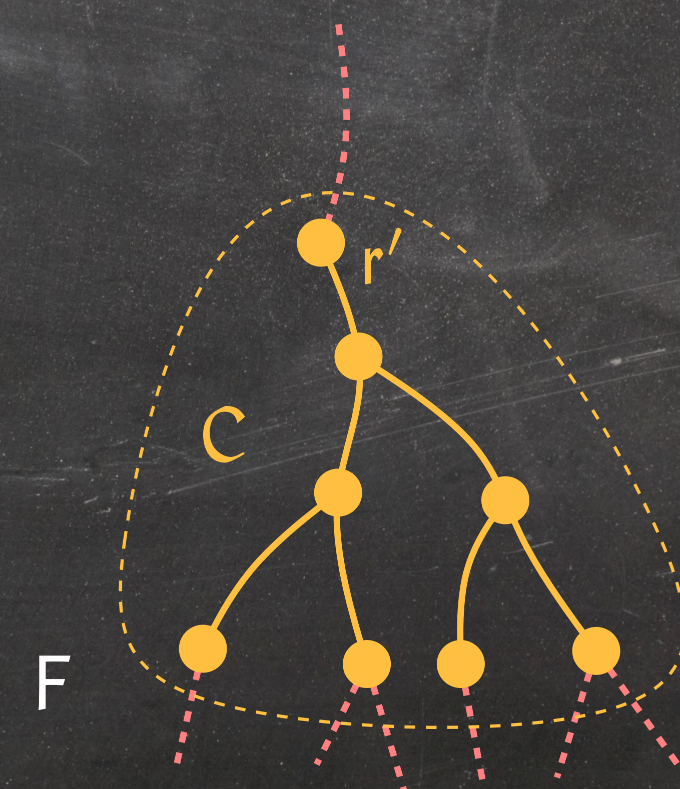
In particular, $u \sim_{SCC(G)} r$, where r is the root of the tree containing u and v.

All vertices in C are descendants of r in $F^r_>$ and $x \leq r$ for all $x \in C$.

Also, $v < r$ because v is a descendant of r in $F^r_>$.

In F, all vertices in C are descendants of some vertex $r' \in C$ and $x \leq r'$ for all $x \in C$.

$\Rightarrow$ $r = r'$ and $u \leq r$.



Strongly Connected Components

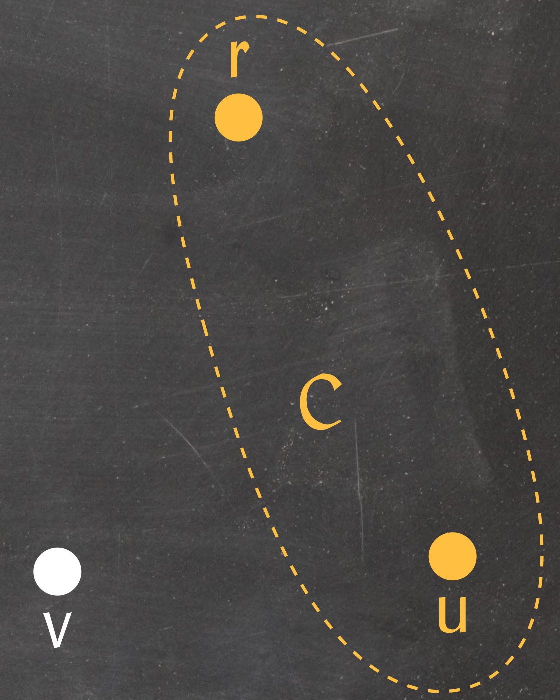
Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

If v is a descendant of r in F, then
$u \sim_{SCC(G)} v$, a contradiction.

Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

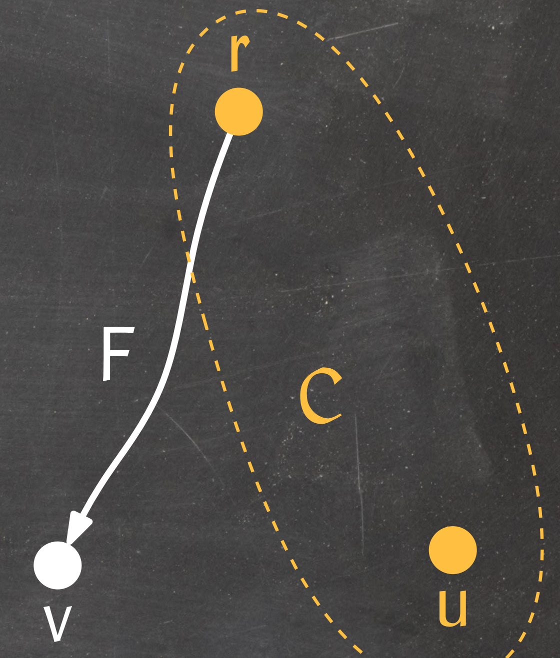
If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

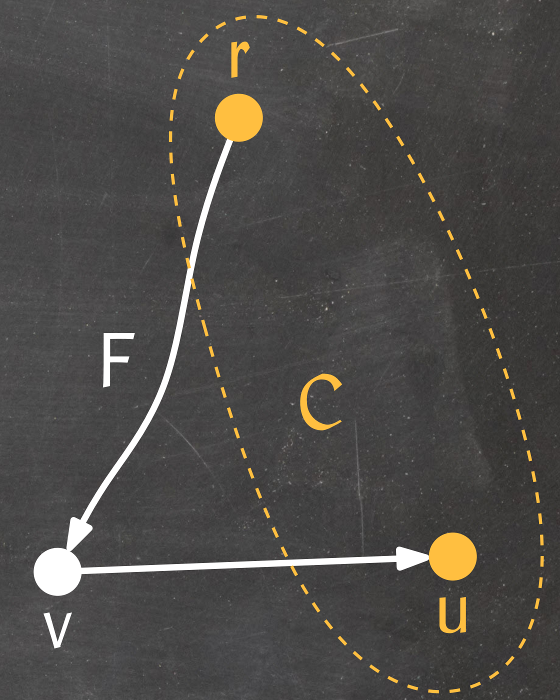
If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

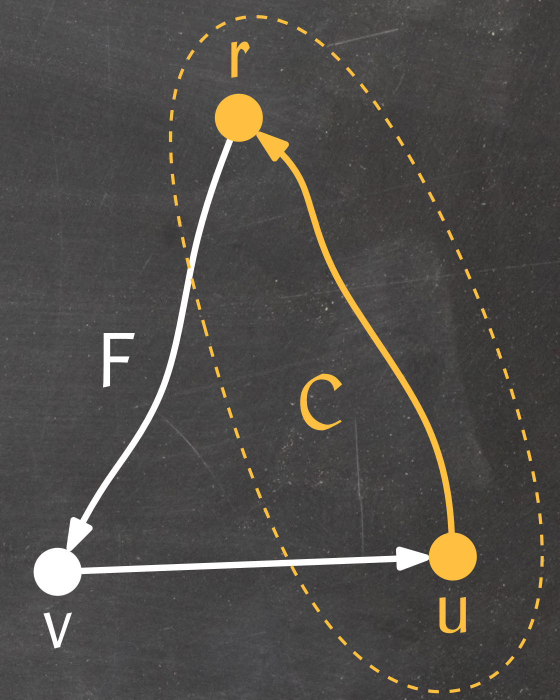
If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.

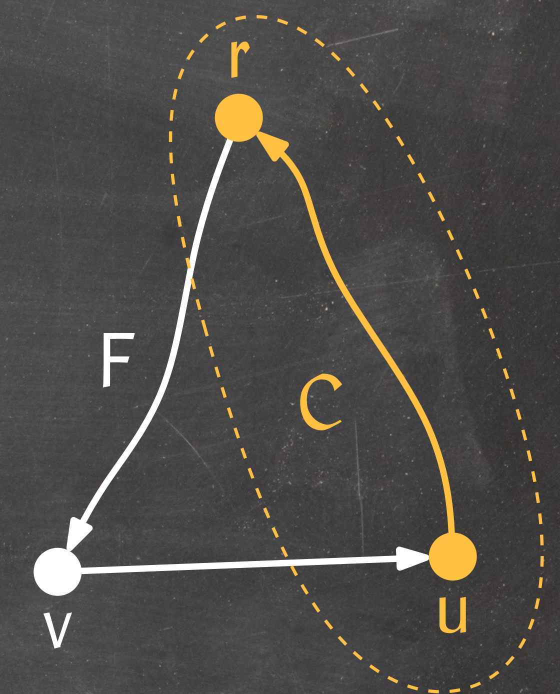


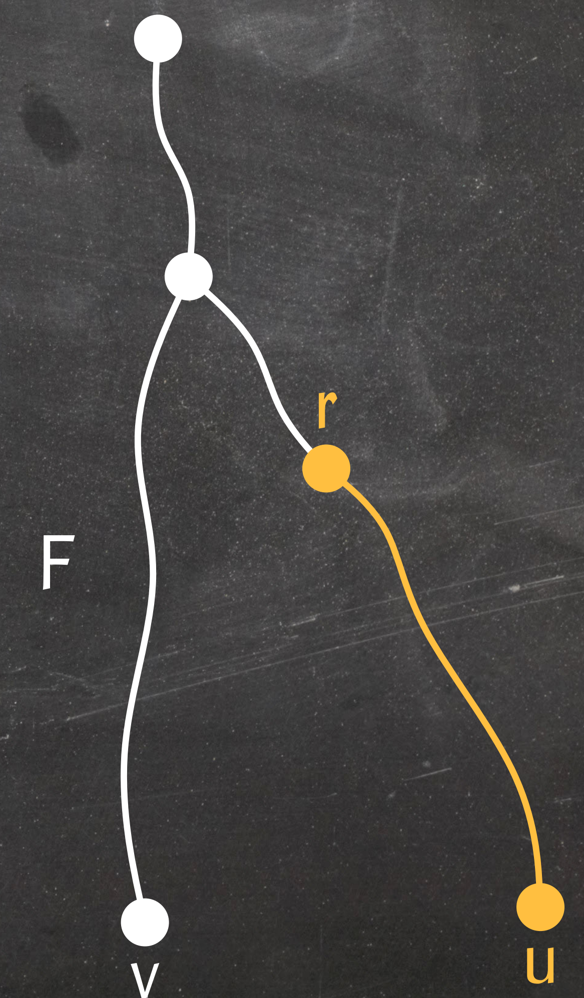
Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_{>})} v$.

If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.

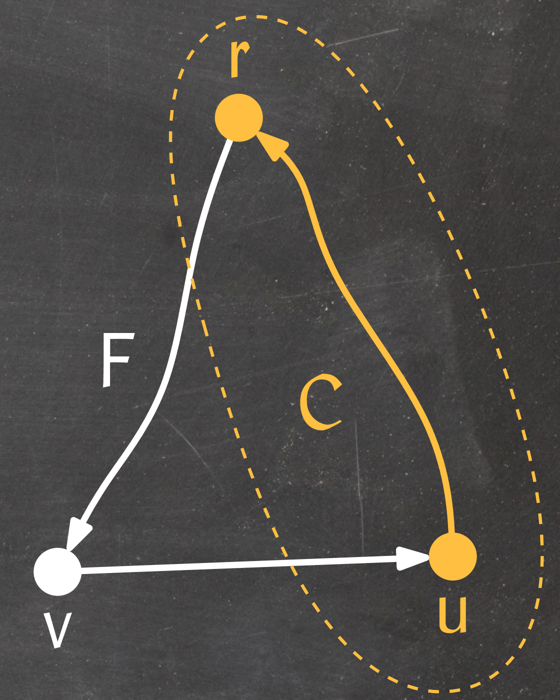
If v is not a descendant of r in F, then v is not a descendant of u because u is a descendant of r.





Strongly Connected Components

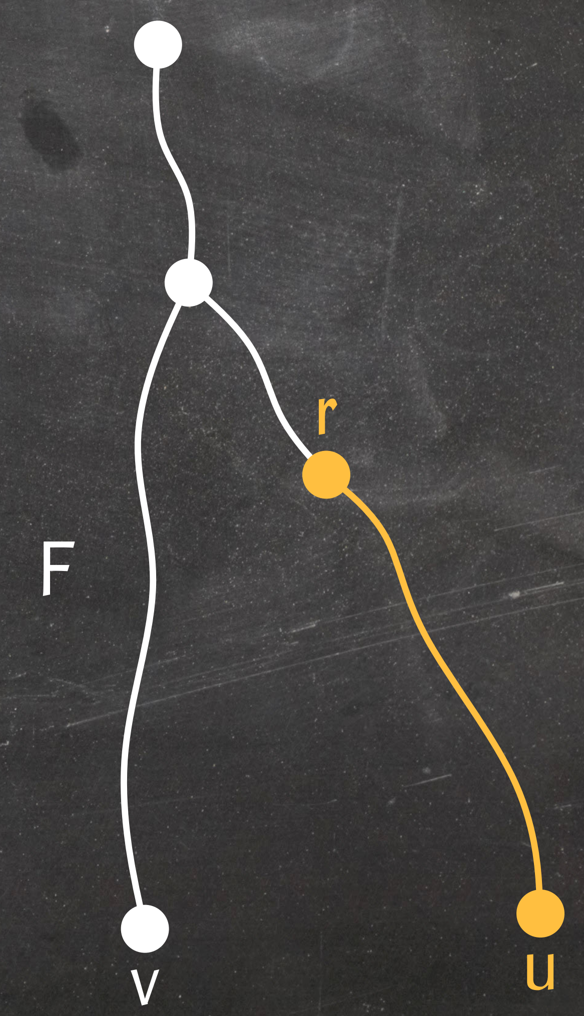
Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.



If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.

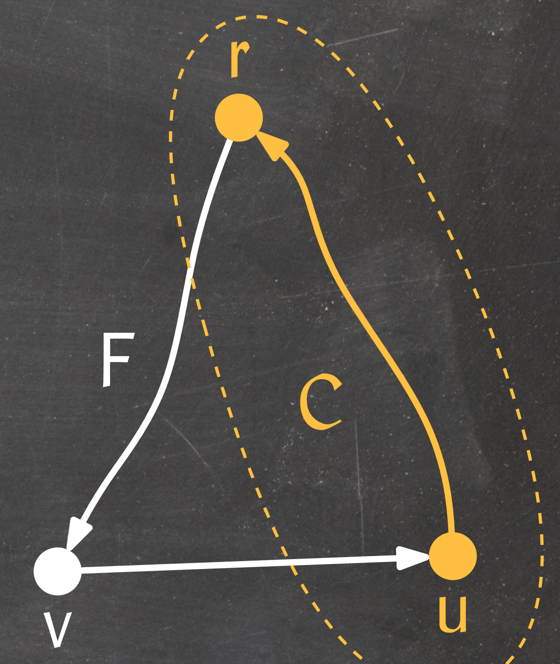
If v is not a descendant of r in F, then v is not a descendant of u because u is a descendant of r.

Since $u \leq r$, $v < r$, and the descendants of r are numbered consecutively, we have $v < u$.



Strongly Connected Components

Lemma: $u \sim_{SCC(G)} v \Leftrightarrow u \sim_{CC(F'_>)} v$.

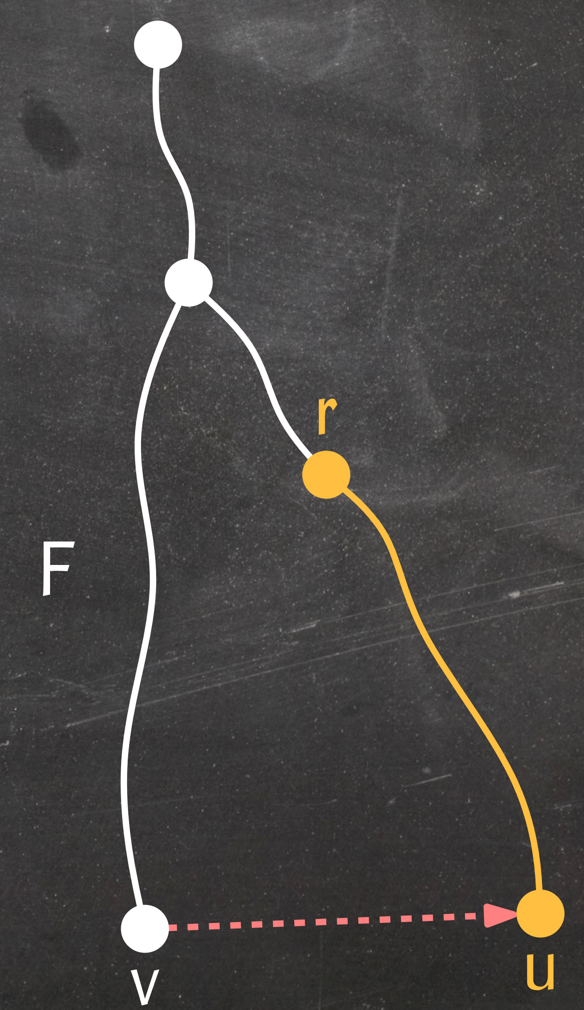


If v is a descendant of r in F, then $u \sim_{SCC(G)} v$, a contradiction.

If v is not a descendant of r in F, then v is not a descendant of u because u is a descendant of r.

Since $u \leq r$, $v < r$, and the descendants of r are numbered consecutively, we have $v < u$.

$\Rightarrow$ (v, u) is a forward cross edge w.r.t. F, a contradiction.



Summary

Graphs are fundamental in Computer Science:

Many problems are quite natural to express as graph problems:

- Matching problems
- Scheduling problems
- …

Data structures are graphs whose nodes store useful information.

Graph exploration lets us learn the structure of a graph:

- Connectivity problems
- Distances between vertices
- Planarity
- …