Elementary Graph Theory

Tao Hou

Outline

- Graphs: definitions (Review+New)
- Representations (Review)
- Topological sort
- DFS (mostly *New*)

Definitions (Review)

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***
  - an edge $e = (u, v)$ from E is a pair of vertices where $u \in V$ and $v \in V$

Definitions (Review)

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***
  - an edge $e = (u, v)$ from E is a pair of vertices where $u \in V$ and $v \in V$

- *directed* graph: an edge (u, v) is from u to v and has a direction
- *undirected* graph: no directions for the edges (so $(u, v) = (v, u)$)

Definitions (Review)

- A ***graph***

$$G = (V, E)$$

- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***
  - an edge $e = (u, v)$ from E is a pair of vertices where $u \in V$ and $v \in V$

- *directed* graph: an edge (u, v) is from u to v and has a direction
- *undirected* graph: no directions for the edges (so $(u, v) = (v, u)$)

- Sometimes given a graph G, we also let $V(G)$ denote the vertex set and $E(G)$ denote the edge set

Definitions (Review)

- A ***graph***

$$G = (V, E)$$

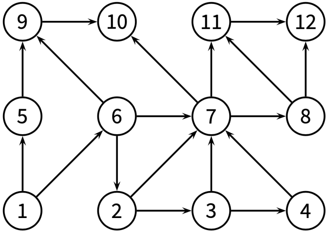
- V is the set of ***vertices*** (also called ***nodes***)
- E is the set of ***edges***
  - an edge $e = (u, v)$ from E is a pair of vertices where $u \in V$ and $v \in V$

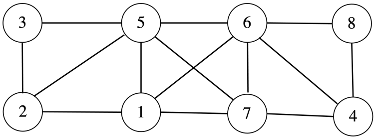
- *directed* graph: an edge (u, v) is from u to v and has a direction
- *undirected* graph: no directions for the edges (so $(u, v) = (v, u)$)

- Sometimes given a graph G, we also let $V(G)$ denote the vertex set and $E(G)$ denote the edge set

- In this course, unless otherwise noted, we assume graphs are ***simple graphs***, i.e., no *self loops* or *parallel edges*.

Examples





Given a graph $G = (V, E)$,

- We call $G' = (V', E')$ a ***subgraph*** of G if $V' \subseteq V$ and $E' \subseteq E$.

More Definitions (Review)

Given a graph $G = (V, E)$,

- We call $G' = (V', E')$ a ***subgraph*** of G if $V' \subseteq V$ and $E' \subseteq E$.
- For a subset of vertices $V' \subseteq V$, the subgraph $G' = (V', E')$ ***induced*** by V' has an edge set consisting of all edges of G whose vertices are in V', i.e.,

$$E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$$

More Definitions (Review)

Given a graph $G = (V, E)$,

- We call $G' = (V', E')$ a ***subgraph*** of G if $V' \subseteq V$ and $E' \subseteq E$.
- For a subset of vertices $V' \subseteq V$, the subgraph $G' = (V', E')$ ***induced*** by V' has an edge set consisting of all edges of G whose vertices are in V', i.e.,

$$E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$$

- A ***path*** in G is a sequence of vertices $v_1, v_2, \ldots, v_k$ s.t. each (v_i, v_{i+1}) forms an edge in G
  - This applies to both directed and undirected graphs
  - Sometime a path also refers to the ***sequence of edges*** on the path

More Definitions (Review)

Given a graph $G = (V, E)$,

- We call $G' = (V', E')$ a ***subgraph*** of G if $V' \subseteq V$ and $E' \subseteq E$.
- For a subset of vertices $V' \subseteq V$, the subgraph $G' = (V', E')$ ***induced*** by V' has an edge set consisting of all edges of G whose vertices are in V', i.e.,

$$E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$$

- A ***path*** in G is a sequence of vertices $v_1, v_2, \ldots, v_k$ s.t. each (v_i, v_{i+1}) forms an edge in G
  - This applies to both directed and undirected graphs
  - Sometime a path also refers to the ***sequence of edges*** on the path
  - A path is called ***simple*** if there are no duplicate vertices on the path

More Definitions (Review)

Given a graph $G = (V, E)$,

- We call $G' = (V', E')$ a ***subgraph*** of G if $V' \subseteq V$ and $E' \subseteq E$.
- For a subset of vertices $V' \subseteq V$, the subgraph $G' = (V', E')$ ***induced*** by V' has an edge set consisting of all edges of G whose vertices are in V', i.e.,

$$E' = \{(u, v) \in E \mid u \in V' \text{ and } v \in V'\}$$

- A ***path*** in G is a sequence of vertices $v_1, v_2, \ldots, v_k$ s.t. each (v_i, v_{i+1}) forms an edge in G
  - This applies to both directed and undirected graphs
  - Sometime a path also refers to the ***sequence of edges*** on the path
  - A path is called ***simple*** if there are no duplicate vertices on the path
- A ***cycle*** is a path starting and ending at the same vertex
  - A cycle is called ***simple*** if there are no duplicate vertices on the cycle other than the starting and ending vertices

For a *directed* graph $G = (V, E)$,

- The ***out-degree*** of a vertex $x \in V$ is the number of edges starting with x, i.e,

$$\text{out-deg}(x) = \left|\{(u, v) \in E \mid u = x\}\right|$$

- The ***in-degree*** of a vertex $x \in V$ is the number of edges ending with x, i.e,

$$\text{in-deg}(x) = \left|\{(u, v) \in E \mid v = x\}\right|$$

More Definitions (Review)

For a *directed* graph $G = (V, E)$,

- The ***out-degree*** of a vertex $x \in V$ is the number of edges starting with x, i.e,

$$\text{out-deg}(x) = \left|\{(u, v) \in E \mid u = x\}\right|$$

- The ***in-degree*** of a vertex $x \in V$ is the number of edges ending with x, i.e,

$$\text{in-deg}(x) = \left|\{(u, v) \in E \mid v = x\}\right|$$

- We have

$$\sum_{v \in V} \text{out-deg}(v) = \sum_{v \in V} \text{in-deg}(v) = |E|$$

More Definitions (Review)

For a *directed* graph $G = (V, E)$,

- The ***out-degree*** of a vertex $x \in V$ is the number of edges starting with x, i.e,

$$\text{out-deg}(x) = \left|\{(u, v) \in E \mid u = x\}\right|$$

- The ***in-degree*** of a vertex $x \in V$ is the number of edges ending with x, i.e,

$$\text{in-deg}(x) = \left|\{(u, v) \in E \mid v = x\}\right|$$

- We have

$$\sum_{v \in V} \text{out-deg}(v) = \sum_{v \in V} \text{in-deg}(v) = |E|$$

For an *undirected* graph $G = (V, E)$,

- The ***degree*** of a vertex $x \in V$ is the number of edges having x as a vertex, i.e,

$$\text{deg}(x) = \left|\{(u, v) \in E \mid u = x \text{ or } v = x\}\right|$$

More Definitions (Review)

For a *directed* graph $G = (V, E)$,

- The ***out-degree*** of a vertex $x \in V$ is the number of edges starting with x, i.e,

$$\text{out-deg}(x) = \left|\{(u, v) \in E \mid u = x\}\right|$$

- The ***in-degree*** of a vertex $x \in V$ is the number of edges ending with x, i.e,

$$\text{in-deg}(x) = \left|\{(u, v) \in E \mid v = x\}\right|$$

- We have

$$\sum_{v \in V} \text{out-deg}(v) = \sum_{v \in V} \text{in-deg}(v) = |E|$$

For an *undirected* graph $G = (V, E)$,

- The ***degree*** of a vertex $x \in V$ is the number of edges having x as a vertex, i.e,

$$\text{deg}(x) = \left|\{(u, v) \in E \mid u = x \text{ or } v = x\}\right|$$

- We have

$$\sum_{v \in V} \text{deg}(v) = 2|E|$$

Connectivity (Review)

Given an ***undirected*** graph $G = (V, E)$,

- Two vertices u, v are ***connected*** in G if there is a path from u to v in G

Given an ***undirected*** graph $G = (V, E)$,

- Two vertices u, v are ***connected*** in G if there is a path from u to v in G
- A ***connected component*** $U \subseteq V$ of G is a ***maximal*** set of vertices where each pair are connected by a path in G (***maximal*** means you cannot add more vertices to U anymore)

Connectivity (Review)

Given an ***undirected*** graph $G = (V, E)$,

- Two vertices u, v are ***connected*** in G if there is a path from u to v in G
- A ***connected component*** $U \subseteq V$ of G is a ***maximal*** set of vertices where each pair are connected by a path in G (***maximal*** means you cannot add more vertices to U anymore)
  - Sometimes, a connected component also refers to the *subgraph induced by* U.

Connectivity (Review)

Given an ***undirected*** graph $G = (V, E)$,

- Two vertices u, v are ***connected*** in G if there is a path from u to v in G
- A ***connected component*** $U \subseteq V$ of G is a ***maximal*** set of vertices where each pair are connected by a path in G (***maximal*** means you cannot add more vertices to U anymore)
  - Sometimes, a connected component also refers to the *subgraph induced by* U.
- G is called ***connected*** if it contains a single connected component (i.e., every two vertices are connected by a path)

Given an ***undirected*** graph $G = (V, E)$,

- Two vertices u, v are ***connected*** in G if there is a path from u to v in G
- A ***connected component*** $U \subseteq V$ of G is a ***maximal*** set of vertices where each pair are connected by a path in G (***maximal*** means you cannot add more vertices to U anymore)
    - Sometimes, a connected component also refers to the *subgraph induced by* U.
- G is called ***connected*** if it contains a single connected component (i.e., every two vertices are connected by a path)

The version of 'connected components' for ***directed*** graphs are called ***strongly connected components***, which we do not touch

Tree

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here ‘acyclic’ means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here ‘acyclic’ means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph

Tree

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here 'acyclic' means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph
  - Each connected component is a tree (so a forest nothing but a disjoint-union of trees)

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here 'acyclic' means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph
  - Each connected component is a tree (so a forest nothing but a disjoint-union of trees)
- A ***rooted*** tree is a ***directed*** graph derived from a tree (which is undirected) by choosing a ***root*** vertex first, and then directing edges s.t. each edge points from a ***parent*** to its ***child***.

Tree

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here ‘acyclic’ means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph
  - Each connected component is a tree (so a forest nothing but a disjoint-union of trees)
- A ***rooted*** tree is a ***directed*** graph derived from a tree (which is undirected) by choosing a ***root*** vertex first, and then directing edges s.t. each edge points from a ***parent*** to its ***child***.
  - One way to understand the ‘directing’ process: perform a DFS on the tree starting from the root. The directed edges always point from a vertex visited *earlier* to a vertex visited *later*
  - Specifically, the root vertex is visited the earliest, so edges are always pointing from the root to other vertices

Tree

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here 'acyclic' means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph
  - Each connected component is a tree (so a forest nothing but a disjoint-union of trees)
- A ***rooted*** tree is a ***directed*** graph derived from a tree (which is undirected) by choosing a ***root*** vertex first, and then directing edges s.t. each edge points from a ***parent*** to its ***child***.
  - One way to understand the 'directing' process: perform a DFS on the tree starting from the root. The directed edges always point from a vertex visited *earlier* to a vertex visited *later*
  - Specifically, the root vertex is visited the earliest, so edges are always pointing from the root to other vertices
  - Most 'tree data structures' are indeed rooted trees, e.g., binary trees, heaps, B-tree

Tree

- A ***tree*** is an ***acyclic***, ***undirected***, ***connected*** graph.
  - Here ‘acyclic’ means having no ***edge-disjoint*** cycles, i.e., there is not a cycle containing distinct edges
- A ***forest*** is an ***acyclic***, ***undirected*** graph
  - Each connected component is a tree (so a forest nothing but a disjoint-union of trees)
- A ***rooted*** tree is a ***directed*** graph derived from a tree (which is undirected) by choosing a ***root*** vertex first, and then directing edges s.t. each edge points from a ***parent*** to its ***child***.
  - One way to understand the ‘directing’ process: perform a DFS on the tree starting from the root. The directed edges always point from a vertex visited *earlier* to a vertex visited *later*
  - Specifically, the root vertex is visited the earliest, so edges are always pointing from the root to other vertices
  - Most ‘tree data structures’ are indeed rooted trees, e.g., binary trees, heaps, B-tree
- More on rooted tree:
  - Each vertex has exactly one in-coming edge from its ***parent*** except the root, which has no in-coming edges.
  - If there is a path from u to v, then u is an ***ancestor*** of v and v is a ***descendant*** of u

Observation

A tree with n vertices has $n - 1$ edges.

Observation

A tree with n vertices has $n-1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n-1$ edges one by one.

Observation

A tree with n vertices has $n - 1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n - 1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).

Observation

A tree with n vertices has $n - 1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n - 1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n - 1$:

Observation

A tree with n vertices has $n - 1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n - 1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n - 1$:
  - If the number of edges is $< n - 1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)

Observation

A tree with n vertices has $n-1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n-1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n-1$:
  - If the number of edges is $< n-1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)
- Number of edges cannot be $> n-1$:

Observation

A tree with n vertices has $n-1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n-1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n-1$:
  - If the number of edges is $< n-1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)
- Number of edges cannot be $> n-1$:
  - If the number of edges is $> n-1$, consider adding the first $n-1$ edges.

Observation

A tree with n vertices has $n-1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n-1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n-1$:
  - If the number of edges is $< n-1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)
- Number of edges cannot be $> n-1$:
  - If the number of edges is $> n-1$, consider adding the first $n-1$ edges.
  - Since the tree has no cycle, only situation (1) can happen.

Observation

A tree with n vertices has $n-1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n-1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n-1$:
  - If the number of edges is $< n-1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)
- Number of edges cannot be $> n-1$:
  - If the number of edges is $> n-1$, consider adding the first $n-1$ edges.
  - Since the tree has no cycle, only situation (1) can happen.
  - So after adding the $n-1$ edges, there is only one connected component.

Observation

A tree with n vertices has $n - 1$ edges.

Proof:

- Consider that initially we only have the n vertices of the tree, and we add each of the $n - 1$ edges one by one.
- When we add each edge, exactly one of the following two things can happen:
  (1) Two connected components in the graph merge into one connected component (number of connected components decrease by 1);
  (2) A cycle is created (number of connected components stays the same).
- Number of edges cannot be $< n - 1$:
  - If the number of edges is $< n - 1$, then the number of connected components cannot decrease to one (adding each edge decreases the number of connected component by at most one)
- Number of edges cannot be $> n - 1$:
  - If the number of edges is $> n - 1$, consider adding the first $n - 1$ edges.
  - Since the tree has no cycle, only situation (1) can happen.
  - So after adding the $n - 1$ edges, there is only one connected component.
  - This means that when we add the n-th edge, it must create a cycle.

Fact

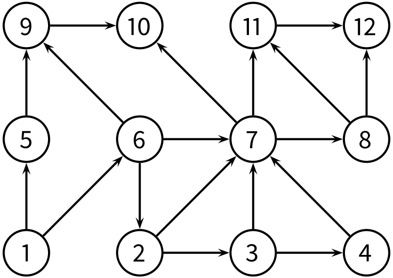
A connected, undirected graph with n vertices and $n - 1$ edges is a tree

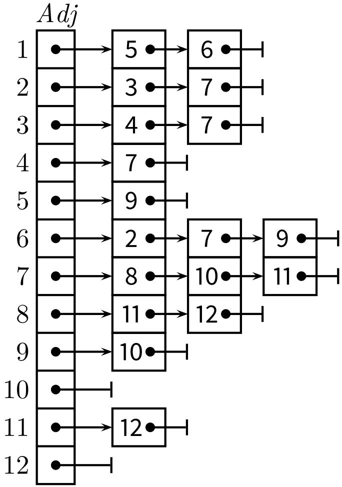
- How do we represent a graph $G = (V, E)$ in a computer?

Adjacency-list representation:

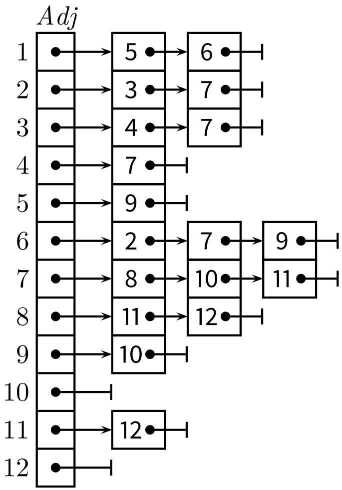
- $V = \{1, 2, \ldots, |V|\}$
- G consists of an array Adj
- A vertex $u \in V$ is represented by an element in the array Adj
- $Adj[u]$ is the ***adjacency list*** of vertex u
  - the list of the vertices that are adjacent to u
  - i.e., the list of all v such that $(u, v) \in E$
  - Notice the difference between *directed* and *undirected* graphs

Example


9
10
11
12
5
6
7
8
1
2
3
4


Adj
1 5 6
2 3 7
3 4 7
4 7
5 9
6 2 7 9
7 8 10 11
8 11 12
9 10
10
11 12
12

- Iteration through E? $O(|V| + |E|)$
  - okay (not optimal)
- Checking $(u, v) \in E$? $O(|V|)$
  - looks bad, but it depends



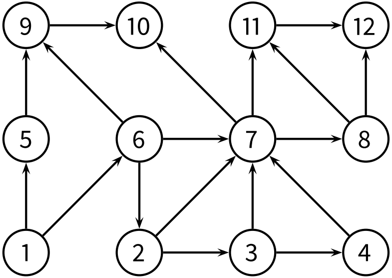
Adjacency-Matrix Representation (Review)

Adjacency-matrix representation:

- $V = \{1, 2, \ldots |V|\}$
- G consists of a $|V| \times |V|$ matrix A
- $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Example



	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Using the Adjacency Matrix (Review)

- Iteration through E? $O(|V|^2)$
  - possibly very bad
- Checking $(u, v) \in E$? $O(1)$
  - optimal

	1	2	3	4	5	6	7	8	9	10	11	12
1					1	1						
2			1				1					
3				1			1					
4							1					
5									1			
6		1					1		1			
7								1		1	1	
8											1	1
9										1		
10												
11												1
12												

Space Complexity (Review)

- Adjacency-list representation

$$\boxed{O(|V| + |E|)}$$

  optimal

- Adjacency-matrix representation

$$\boxed{O(|V|^2)}$$

  possibly very bad

Choosing a Graph Representation (Review)

- Adjacency-list representation
  - generally good, especially for its optimal space complexity
  - bad for ***dense*** graphs and algorithms that require random access to edges
  - preferable for ***sparse*** graphs or graphs with ***low degree***
- Adjacency-matrix representation
  - suffers from a bad space complexity
  - good for algorithms that require random access to edges
  - preferable for ***dense*** graphs
- Sparse vs. dense graph
  - ***Sparse*** graph: $|E| = O(|V|)$
  - ***Dense*** graph: $|E| = \Theta(|V|^2)$

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward edges*
  - in another word, if there is an edge (u, v), then u appears before v in the ordering (that's also the reason why we can do this *only* on DAG instead of general graphs)

- **Problem:** (topological sort)

  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward edges*
  - in another word, if there is an edge (u, v), then u appears before v in the ordering (that's also the reason why we can do this *only* on DAG instead of general graphs)
  - ***Note***: The 'acyclic' here is for directed graphs and therefore means only 'no cycles' (we don't need to say 'no edge-disjoint cycles' here)

- **Problem:** (topological sort)

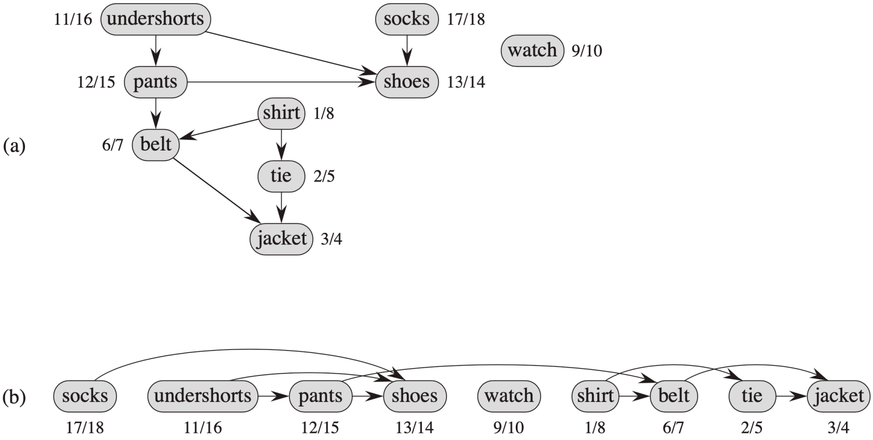
  Given a *directed acyclic graph* (DAG)

  - find an ordering of vertices such that you only end up with *forward edges*
  - in another word, if there is an edge (u, v), then u appears before v in the ordering (that's also the reason why we can do this *only* on DAG instead of general graphs)
  - ***Note***: The 'acyclic' here is for directed graphs and therefore means only 'no cycles' (we don't need to say 'no edge-disjoint cycles' here)

- **Example:** dependencies in software packages

  - find an installation order for a set of software packages
  - such that every package is installed only after all the packages it depends on

Example



(Example from CLRS)

Topological Sort Algorithm

Topological-Sort(G)

while $\exists v \in V$ s.t. $in\text{-}deg(v) = 0$

    output v

    remove v and all its out-going edges from G

Topological Sort Algorithm

TOPOLOGICAL-SORT(G)

```
while ∃v ∈ V s.t. in-deg(v) = 0
      output v
      remove v and all its out-going edges from G
```

Argument of correctness:

- We remove an edge only when its starting vertex has been output in the order
- Thus, when a vertex *v* has in-degree 0, this means that all vertices pointing to *v* (if any) have been output, so that we can also safely output v

Topological Sort Algorithm

TOPOLOGICAL-SORT(G)

while $\exists v \in V$ s.t. $in\text{-}deg(v) = 0$
  output v
  remove v and all its out-going edges from G

Argument of correctness:

- We remove an edge only when its starting vertex has been output in the order
- Thus, when a vertex v has in-degree 0, this means that all vertices pointing to v (if any) have been output, so that we can also safely output v

Question:

- Why should there always be a vertex with 0 in-degree?

Topological Sort: Alternative Algorithm

```
TOPOLOGICAL-SORT(G)
    DFS(G)
    output V sorted in reverse order of f[·]
```

Topological Sort: Alternative Algorithm

```
TOPOLOGICAL-SORT(G)
DFS(G)
output V sorted in reverse order of f[·]
```

We will see why this algorithm works later on.
Some comments:

- The first algorithm is mainly of theoretical value (helps you to understand the whole procedure)
- In practice, you should utilize DFS to compute topological sorting for DAGs because it's much simpler (you don't need to bother to delete the edges)
- So topological sort can be done in $O(|V| + |E|)$ time

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s
  - We also enumerate *all possible* seeds and traverse the entire graph eventually

Depth-First Search

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s
  - We also enumerate *all possible* seeds and traverse the entire graph eventually
- Visiting of vertices is done in *recursive* fashion:
  - When we visit a vertex u, we immediately visit an adjacent vertex v of u without finishing the visiting of u
  - We finish visiting u when all adjacent vertices has been visited (hence the finishing of the visiting is defined *recursively*)
  - We backtrack when we finish visiting a vertex (done automatically by recursion)

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s
  - We also enumerate *all possible* seeds and traverse the entire graph eventually
- Visiting of vertices is done in *recursive* fashion:
  - When we visit a vertex u, we immediately visit an adjacent vertex v of u without finishing the visiting of u
  - We finish visiting u when all adjacent vertices has been visited (hence the finishing of the visiting is defined *recursively*)
  - We backtrack when we finish visiting a vertex (done automatically by recursion)
- Produces a ***DFS forest***, consisting of all the ***DFS trees*** rooted at the seeds

Depth-First Search

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s
  - We also enumerate *all possible* seeds and traverse the entire graph eventually
- Visiting of vertices is done in *recursive* fashion:
  - When we visit a vertex u, we immediately visit an adjacent vertex v of u without finishing the visiting of u
  - We finish visiting u when all adjacent vertices has been visited (hence the finishing of the visiting is defined *recursively*)
  - We backtrack when we finish visiting a vertex (done automatically by recursion)
- Produces a ***DFS forest***, consisting of all the ***DFS trees*** rooted at the seeds
- Coloring for vertices:
  - ***white***: not yet visited
  - ***grey***: being visited, but haven't finished visiting
  - ***black***: finished visiting

- *Input:* $G = (V, E)$, which can be *directed* or *undirected*
- Explores the graph starting from s, touching all vertices that are reachable from s
  - We also enumerate *all possible* seeds and traverse the entire graph eventually
- Visiting of vertices is done in *recursive* fashion:
  - When we visit a vertex u, we immediately visit an adjacent vertex v of u without finishing the visiting of u
  - We finish visiting u when all adjacent vertices has been visited (hence the finishing of the visiting is defined *recursively*)
  - We backtrack when we finish visiting a vertex (done automatically by recursion)
- Produces a ***DFS forest***, consisting of all the ***DFS trees*** rooted at the seeds
- Coloring for vertices:
  - ***white***: not yet visited
  - ***grey***: being visited, but haven't finished visiting
  - ***black***: finished visiting
- Associates ***two time-stamps*** to each vertex
  - $d[u]$ records when DFS starts visiting u (turns *grey*)
  - $f[u]$ records when DFS finishes visiting u and therefore backtracks from u (turns *black*)

DFS Algorithm

```
DFS(G)
for each vertex u ∈ V(G)
        color[u] = WHITE
        π[u] = NIL
time = 0 // "global" variable
for each vertex u ∈ V(G)
        if color[u] == WHITE
                DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] = GREY
time = time + 1
d[u] = time
for each v ∈ Adj[u]
        if color[v] == WHITE
                π[v] = u
                DFS-VISIT(v)
color[u] = BLACK
time = time + 1
f[u] = time
```

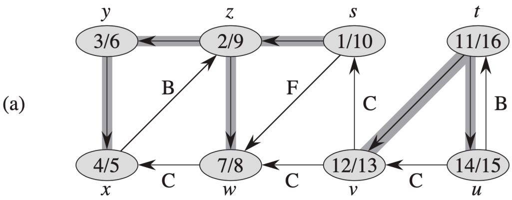
DFS Algorithm

```
DFS(G)
for each vertex u ∈ V(G)
    color[u] = WHITE
    π[u] = NIL
time = 0 // "global" variable
for each vertex u ∈ V(G)
    if color[u] == WHITE
        DFS-VISIT(u)
```

```
DFS-VISIT(u)
color[u] = GREY
time = time + 1
d[u] = time
for each v ∈ Adj[u]
    if color[v] == WHITE
        π[v] = u
        DFS-VISIT(v)
color[u] = BLACK
time = time + 1
f[u] = time
```

- A first very silly question: Can DFS ever end?

DFS: Example



(Example from CLRS)

Complexity of DFS

Complexity of DFS

- The loop in **DFS-VISIT**(u) (lines 4–7) executes for $O(\text{out-deg}(u))$ times
- We call **DFS-VISIT**(u) *once* for each vertex u
  - either in **DFS**, or recursively in **DFS-VISIT**
  - because we call it only if $color[u] = $ WHITE, but then we immediately set $color[u] = $ GREY
- So, the overall complexity is $\Theta(|V| + |E|)$

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $\left[d[u], f[u]\right]$ and $\left[d[v], f[v]\right]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $\left[d[v], f[v]\right] \subseteq \left[d[u], f[u]\right]$ and v is a descendant of u)

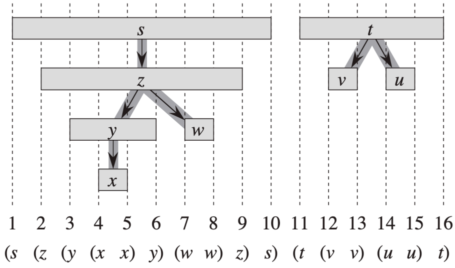
Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $[d[v], f[v]] \subseteq [d[u], f[u]]$ and v is a descendant of u)

Example (from CLRS):

(b)



Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $\left[d[u], f[u]\right]$ and $\left[d[v], f[v]\right]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $\left[d[v], f[v]\right] \subseteq \left[d[u], f[u]\right]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $[d[v], f[v]] \subseteq [d[u], f[u]]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$
- Then, by comparing $d[v]$ with $f[u]$, we have two case: (1) $d[v] < f[u]$; (2) $d[v] > f[u]$

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $[d[u], f[u]]$ and $[d[v], f[v]]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $[d[v], f[v]] \subseteq [d[u], f[u]]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$
- Then, by comparing $d[v]$ with $f[u]$, we have two case: (1) $d[v] < f[u]$; (2) $d[v] > f[u]$
- First consider $d[v] < f[u]$ (aka. $d[u] < d[v] < f[u]$)

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $\left[d[u], f[u]\right]$ and $\left[d[v], f[v]\right]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $\left[d[v], f[v]\right] \subseteq \left[d[u], f[u]\right]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$
- Then, by comparing $d[v]$ with $f[u]$, we have two case: (1) $d[v] < f[u]$; (2) $d[v] > f[u]$
- First consider $d[v] < f[u]$ (aka. $d[u] < d[v] < f[u]$)
- Observe: *the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations*

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $\left[d[u], f[u]\right]$ and $\left[d[v], f[v]\right]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $\left[d[v], f[v]\right] \subseteq \left[d[u], f[u]\right]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$
- Then, by comparing $d[v]$ with $f[u]$, we have two case: (1) $d[v] < f[u]$; (2) $d[v] > f[u]$
- First consider $d[v] < f[u]$ (aka. $d[u] < d[v] < f[u]$)
- Observe: *the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations*
- This means that v is a descendant of u in the DFS forest

Parenthesis Theorem

In a DFS on a (directed or undirected) graph G, for any two vertices u and v, exactly one of the following two holds:

1. The intervals $\left[d[u], f[u]\right]$ and $\left[d[v], f[v]\right]$ are entirely disjoint, and neither one is a descendant of the other in the DFS forest
2. One interval is entirely contained in the other interval, and the vertex is a descendant of another (e.g., $\left[d[v], f[v]\right] \subseteq \left[d[u], f[u]\right]$ and v is a descendant of u)

proof:

- Without loss of generality, assume $d[u] < d[v]$
- Then, by comparing $d[v]$ with $f[u]$, we have two case: (1) $d[v] < f[u]$; (2) $d[v] > f[u]$
- First consider $d[v] < f[u]$ (aka. $d[u] < d[v] < f[u]$)
- Observe: *the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations*
- This means that v is a descendant of u in the DFS forest
- Also, the visiting of u cannot finish before we finish visiting u (this is how recursive calls work), so $f[v] < f[u]$ (aka. $d[u] < d[v] < f[v] < f[u]$)

- Now consider $d[v] > f[u]$
- Obviously, $d[u] < f[u] < d[v] < f[v]$, so the two intervals are disjoint

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v on G consisting of *only* white vertices

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v on G consisting of *only* white vertices

proof:

- "$\Rightarrow$": let w be any descendant of u in the DFS tree
- By the previous Parenthesis Theorem, we have that $d[u] < d[w]$, so we u is discovered, w is still white
- Notice that on the path from u to v in the DFS tree, all vertices are descendants of v, so all of them are white at time $d[u]$

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree
- Let x be the first vertex on the path that is not a descendant of u (why such an x exists?)

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree
- Let x be the first vertex on the path that is not a descendant of u (why such an x exists?)
- Let w be the predecessor of x on the path (so that w is a descendant of u; notice that w could be u itself)

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree
- Let x be the first vertex on the path that is not a descendant of u (why such an x exists?)
- Let w be the predecessor of x on the path (so that w is a descendant of u; notice that w could be u itself)
- Since $d[u] < d[x]$, by the Parenthesis Theorem, we must have $d[u] < f[u] < d[x]$ (because x is not descendant of u)

White-Path Theorem

In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree
- Let x be the first vertex on the path that is not a descendant of u (why such an x exists?)
- Let w be the predecessor of x on the path (so that w is a descendant of u; notice that w could be u itself)
- Since $d[u] < d[x]$, by the Parenthesis Theorem, we must have $d[u] < f[u] < d[x]$ (because x is not descendant of u)
- Consider the time the search visits w, we must have that x is white during the whole process (because if you haven't finish visiting w, you definitely haven't finished visiting u)

White-Path Theorem

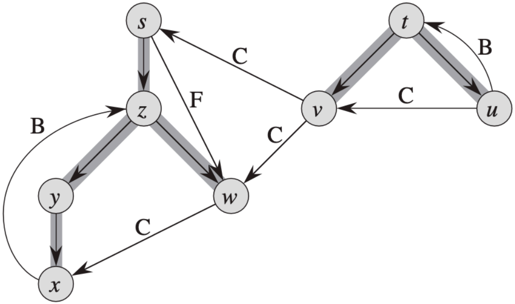
In a DFS forest of a (directed or undirected) graph G, a vertex v is a descendant of a vertex u if and only if at time $d[u]$, there is a path from u to v consisting of *only* white vertices

proof (continue):

- "$\Leftarrow$": Use proof by contradiction, suppose that there is a "white path" from u to v at time $d[u]$, but v is not a descendant of u in the DFS tree
- Let x be the first vertex on the path that is not a descendant of u (why such an x exists?)
- Let w be the predecessor of x on the path (so that w is a descendant of u; notice that w could be u itself)
- Since $d[u] < d[x]$, by the Parenthesis Theorem, we must have $d[u] < f[u] < d[x]$ (because x is not descendant of u)
- Consider the time the search visits w, we must have that x is white during the whole process (because if you haven't finish visiting w, you definitely haven't finished visiting u)
- But if this is true, then x must be a descendant of w and in turn a descendant of u (a contradiction)

Four Types of Edges in DFS on *Undirected* Graphs

- ***Tree edge***: Edges on the DFS forest
- ***Back edge***: Connecting a vertex to its *ancestor* in the DFS forest
- ***Forward edge***: Non-tree edges connecting a vertex to its *descendant* in the DFS forest
- ***Cross edge***: all other edges



(Example from CLRS)

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c
- Without loss of generality, assume c is a simple cycle

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c
- Without loss of generality, assume c is a simple cycle
- Let v be the first vertex on c discovered by DFS , and let u be the vertex pointing to v on c

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c
- Without loss of generality, assume c is a simple cycle
- Let v be the first vertex on c discovered by DFS , and let u be the vertex pointing to v on c
- When v is discovered, we have that all vertices on path from v to u (on c) are white (undiscovered)

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c
- Without loss of generality, assume c is a simple cycle
- Let v be the first vertex on c discovered by DFS , and let u be the vertex pointing to v on c
- When v is discovered, we have that all vertices on path from v to u (on c) are white (undiscovered)
- By the White-Path Theorem, u must be a descendant of v in the depth-first forest

Detecting cycles in an undirected graph using DFS

Lemma

A directed graph G has a cycle if and only if a depth-first search on G yields a back edge

proof:

- "$\Leftarrow$": easy
- Now we try to show the forward direction "$\Rightarrow$"
- Suppose that G contains a cycle c
- Without loss of generality, assume c is a simple cycle
- Let v be the first vertex on c discovered by DFS , and let u be the vertex pointing to v on c
- When v is discovered, we have that all vertices on path from v to u (on c) are white (undiscovered)
- By the White-Path Theorem, u must be a descendant of v in the depth-first forest
- Therefore, (u, v) is a back edge

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1   DFS(G)
2   output V sorted in reverse order of f[·]
```

Proof of correctness:

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

1 **DFS**(G)
2 output V sorted in reverse order of $f[\cdot]$

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1  DFS(G)
2  output V sorted in reverse order of f[·]
```

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$
- First observe: the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1  DFS(G)
2  output V sorted in reverse order of f[·]
```

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$
- First observe: the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations
- When we are visiting u and exploring the edge (u, v) in DFS, we have that v cannot be grey because otherwise (u, v) would be a back edge (notice that u must be at the stack top when we are exploring (u, v)), contradicting the previous Lemma saying that DFS on DAG yields no back edges

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1  DFS(G)
2  output V sorted in reverse order of f[·]
```

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$
- First observe: the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations
- When we are visiting u and exploring the edge (u, v) in DFS, we have that v cannot be grey because otherwise (u, v) would be a back edge (notice that u must be at the stack top when we are exploring (u, v)), contradicting the previous Lemma saying that DFS on DAG yields no back edges
- Then v must be white or black

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1   DFS(G)
2   output V sorted in reverse order of f[·]
```

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$
- First observe: the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations
- When we are visiting u and exploring the edge (u, v) in DFS, we have that v cannot be grey because otherwise (u, v) would be a back edge (notice that u must be at the stack top when we are exploring (u, v)), contradicting the previous Lemma saying that DFS on DAG yields no back edges
- Then v must be white or black
- If v is white, then we shall visit v as a result of exploring the edge (u, v). By DFS, we cannot finish visiting u before finishing visiting v. So $f[v] < f[u]$.

Topological Sort: Alternative Algorithm

TOPOLOGICAL-SORT(G)

```
1  DFS(G)
2  output V sorted in reverse order of f[·]
```

Proof of correctness:

- It suffices to show that for any edge $(u, v) \in G$, $f[v] < f[u]$
- First observe: the grey vertices in DFS always form a linear chain of descendants (in the DFS tree) corresponding to the stack of active DFS-VISIT invocations
- When we are visiting u and exploring the edge (u, v) in DFS, we have that v cannot be grey because otherwise (u, v) would be a back edge (notice that u must be at the stack top when we are exploring (u, v)), contradicting the previous Lemma saying that DFS on DAG yields no back edges
- Then v must be white or black
- If v is white, then we shall visit v as a result of exploring the edge (u, v). By DFS, we cannot finish visiting u before finishing visiting v. So $f[v] < f[u]$.
- If v is black, we have already finished visiting v. But the visiting of u is not finished. So we obviously have $f[v] < f[u]$.

Observation

If there is a path from a vertex u to a vertex v in an ***undirected*** graph G (aka. u, v are in the same connected component), then u, v must be in the same DFS tree after performing a depth-first search on G.

Observation

If there is a path from a vertex u to a vertex v in an ***undirected*** graph G (aka. u, v are in the same connected component), then u, v must be in the same DFS tree after performing a depth-first search on G.

Comment: The opposite is also true. Think about what these observations implies

Observation

If there is a path from a vertex u to a vertex v in an ***undirected*** graph G (aka. u, v are in the same connected component), then u, v must be in the same DFS tree after performing a depth-first search on G.

Comment: The opposite is also true. Think about what these observations implies

Proof:

- Consider a path P connecting u, v in G
- Let x be the first vertex on P visited by DFS. Apparently, we can reach u and v from x
- By the description of DFS, the DFS visit on x will touch all vertices that are reachable from x. So we will reach u and v from visiting x.
- Therefore, u, v, x are all in the same DFS tree.