Directed Graphs & Applications

Algorithm Design & Analysis

Spring 2019
Outline

Directed Graphs

Connectivity and Traversals in Directed Graphs

Applications

Deciding Strong Connectivity
DAGs and Topological Sorting
Directed Graphs

Definition

A directed graph $G = (V, E)$ consists of two sets...
Definition

A directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
Directed Graphs

Definition

A directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$
Directed Graphs

Definition

A directed graph $G = (V, E)$ consists of two sets

- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?
Directed Graphs

Definition
A directed graph $G = (V, E)$ consists of two sets
- A set $V$ called the vertices of $G$
- A set $E$ of ordered pairs of distinct vertices of $V$ called the edges of $G$

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.
Definition

A *directed graph* $G = (V, E)$ consists of two sets

- A set $V$ called the *vertices* of $G$
- A set $E$ of *ordered pairs* of distinct vertices of $V$ called the *edges* of $G$

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.

**Example:** A *directed walk* in $G$ is a sequence $P = u = v_0, v_1, \ldots, v_n = v$ in which each $e_i = (v_{i-1}, v_i)$
**Directed Graphs**

**Definition**

A directed graph \( G = (V, E) \) consists of two sets

- A set \( V \) called the **vertices** of \( G \)
- A set \( E \) of **ordered pairs** of **distinct** vertices of \( V \) called the **edges** of \( G \)

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.

**Example:** A directed walk in \( G \) is a sequence \( P = u = v_0, v_1, \ldots, v_n = v \) in which each \( e_i = (v_{i-1}, v_i) \)

Also: directed path, simple path, closed walk, circuit, cycle
Directed Graphs

Definition

A directed graph \( G = (V, E) \) consists of two sets

- A set \( V \) called the vertices of \( G \)
- A set \( E \) of ordered pairs of distinct vertices of \( V \) called the edges of \( G \)

Note: No loops or multiple edges. Why?

Properties of undirected graphs have counterparts in directed graphs, with some differences.

Example: A directed walk in \( G \) is a sequence \( P = u = v_0, v_1, \ldots, v_n = v \) in which each \( e_i = (v_{i-1}, v_i) \)

Also: directed path, simple path, closed walk, circuit, cycle

Now \( v \) is reachable from \( u \) if there is a directed walk from \( u \) to \( v \)
Reachability in Directed Graphs: An Example
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex
Reachability in Directed Graphs: An Example

BFS and DFS both work on directed graphs
Both visit exactly the nodes reachable from the start vertex
Reachability: An Equivalence Relation
Reachability: An Equivalence Relation

In *undirected* graphs, reachability is an *equivalence relation* between pairs of vertices.
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
Reachability: An Equivalence Relation

In *undirected* graphs, reachability is an *equivalence relation* between pairs of vertices

- $u$ is reachable from $u$ (*reflexive*)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (*symmetric*)
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (symmetric)
- If $v$ is reachable from $u$ and $w$ is reachable from $v$, then $w$ is reachable from $u$ (transitive)
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (symmetric)
- If $v$ is reachable from $u$ and $w$ is reachable from $v$, then $w$ is reachable from $u$ (transitive)

Definition

A binary relation $\sim$ on a set $X$ is an equivalence relation on $X$ if $\sim$ has the following properties
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (symmetric)
- If $v$ is reachable from $u$ and $w$ is reachable from $v$, then $w$ is reachable from $u$ (transitive)

Definition

A binary relation $\simeq$ on a set $X$ is an equivalence relation on $X$ if $\simeq$ has the following properties

- Reflexive \ For all $x \in X$, $x \simeq x$
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (symmetric)
- If $v$ is reachable from $u$ and $w$ is reachable from $v$, then $w$ is reachable from $u$ (transitive)

**Definition**

A binary relation $\simeq$ on a set $X$ is an equivalence relation on $X$ if $\simeq$ has the following properties

**Reflexive** For all $x \in X$, $x \simeq x$

**Symmetric** For all $x, y \in X$, $x \simeq y \iff y \simeq x$
Reachability: An Equivalence Relation

In undirected graphs, reachability is an equivalence relation between pairs of vertices

- $u$ is reachable from $u$ (reflexive)
- If $v$ is reachable from $u$, then $u$ is reachable from $v$ (symmetric)
- If $v$ is reachable from $u$ and $w$ is reachable from $v$, then $w$ is reachable from $u$ (transitive)

**Definition**

A binary relation $\sim$ on a set $X$ is an equivalence relation on $X$ if $\sim$ has the following properties

**Reflexive** For all $x \in X$, $x \sim x$

**Symmetric** For all $x, y \in X$, $x \sim y \iff y \sim x$

**Transitive** For all $x, y, z \in X$, $x \sim y$ and $y \sim z \implies x \sim z$
Equivalence Relations ⇔ Equivalence Classes
**Equivalence Relations ⇐ Equivalence Classes**

An equivalence relation on a set $S$ gives rise to *equivalence classes* $S_x = \{y : y \text{ is equivalent to } x\}$. These equivalence classes have the following properties
Equivalence Relations $\iff$ Equivalence Classes

An equivalence relation on a set $S$ gives rise to equivalence classes $S_x = \{y : y \text{ is equivalent to } x\}$. These equivalence classes have the following properties

- For every $x \in S$, $x \in S_x$
Equivalence Relations $\Leftrightarrow$ Equivalence Classes

An equivalence relation on a set $S$ gives rise to *equivalence classes* $S_x = \{y : y \text{ is equivalent to } x\}$. These equivalence classes have the following properties:

- For every $x \in S$, $x \in S_x$
- For every $x, y \in S$, either $S_x = S_y$ or $S_x \cap S_y = \emptyset$. That is, the equivalence classes *partition* $S$.


An equivalence relation on a set \( S \) gives rise to *equivalence classes* \( S_x = \{ y : y \text{ is equivalent to } x \} \). These equivalence classes have the following properties:

- For every \( x \in S \), \( x \in S_x \)
- For every \( x, y \in S \), either \( S_x = S_y \) or \( S_x \cap S_y = \emptyset \). That is, the equivalence classes *partition* \( S \)
- Alternate notation for \( S_x \): \([x]\)
Equivalence Relations ⇔ Equivalence Classes

An equivalence relation on a set $S$ gives rise to *equivalence classes* $S_x = \{y : y$ is equivalent to $x\}$. These equivalence classes have the following properties

- For every $x \in S$, $x \in S_x$
- For every $x, y \in S$, either $S_x = S_y$ or $S_x \cap S_y = \emptyset$. That is, the equivalence classes *partition* $S$
- Alternate notation for $S_x$: $[x]$

For an *undirected* graph $G = (V, E)$, reachability is an equivalence relation on $V$


**Equivalence Relations ⇔ Equivalence Classes**

An equivalence relation on a set $S$ gives rise to *equivalence classes* $S_x = \{y : y \text{ is equivalent to } x\}$. These equivalence classes have the following properties:

- For every $x \in S$, $x \in S_x$
- For every $x, y \in S$, either $S_x = S_y$ or $S_x \cap S_y = \emptyset$. That is, the equivalence classes partition $S$
- Alternate notation for $S_x$: $[x]$

For an *undirected* graph $G = (V, E)$, reachability is an equivalence relation on $V$

- For each $v \in V$, $[v]$ is the set of vertices in the connected component of $G$ containing $v$. 
Connectivity in Directed Graphs

In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric
Connectivity in Directed Graphs

In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric.

We can define a related equivalence relation on the vertices of a directed graph.
Connectivity in Directed Graphs

In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric.

We can define a related equivalence relation on the vertices of a directed graph.

**Definition**

Two vertices $u, v$ in a directed graph $G$ are *mutually reachable* if there is a directed path from $u$ to $v$ and one from $v$ to $u$. 
Connectivity in Directed Graphs

In directed graphs, reachability is reflexive and transitive, but not guaranteed to be symmetric.

We can define a related equivalence relation on the vertices of a directed graph.

**Definition**
Two vertices $u, v$ in a directed graph $G$ are *mutually reachable* if there is a directed path from $u$ to $v$ and one from $v$ to $u$.

That is, $u, v$ are mutually reachable if $v$ is reachable from $u$ and $u$ is reachable from $v$. 
Mutual Reachability: An Example
Mutual Reachability: An Example
Mutal Reachability : An Equivalence Relation

Claim: Mutual Reachability is an equivalence relation
**Mutual Reachability : An Equivalence Relation**

Claim: Mutual Reachability is an equivalence relation

- Reflexive: Are \( u \) and \( u \) mutually reachable? Yes
Mutual Reachability: An Equivalence Relation

Claim: Mutual Reachability is an equivalence relation

- Reflexive: Are $u$ and $u$ mutually reachable? Yes
- Symmetric: If $u$ and $v$ are mutually reachable, are $v$ and $u$ mutually reachable? Yes
**Mutual Reachability: An Equivalence Relation**

Claim: Mutual Reachability is an equivalence relation

- Reflexive: Are $u$ and $u$ mutually reachable? Yes
- Symmetric: If $u$ and $v$ are mutually reachable, are $v$ and $u$ mutually reachable? Yes
- Transitive: If $u$ and $v$ are mutually reachable and $v$ and $w$ are mutually reachable, are $u$ and $w$ mutually reachable? Yes
Strong Connectivity

Definition

A graph $G$ is *strongly connected* if every pair of vertices are mutually reachable.
Strong Connectivity

Definition

A graph $G$ is *strongly connected* if every pair of vertices are mutually reachable.

The Mutual Reachability relation decomposes $G$ into *strongly connected components*. 
Strong Components: An Example

A graph and its strongly connected components
Strong Components: An Example

A graph and its strongly connected components
Outline

Directed Graphs

Connectivity and Traversals in Directed Graphs

Applications

Deciding Strong Connectivity

DAGs and Topological Sorting
Application: Deciding Strong Connectivity

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.
Application: Deciding Strong Connectivity

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- Observe: $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths
**Application: Deciding Strong Connectivity**

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- **Observe:** $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths
- **Pick a vertex $v$.** Check to see whether every other vertex is reachable from $v$
Application: Deciding Strong Connectivity

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- Observe: $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths
- Pick a vertex $v$. Check to see whether every other vertex is reachable from $v$
- Now see whether $v$ is reachable from every other vertex

Can we do better?
**Application: Deciding Strong Connectivity**

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- Observe: $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths
- Pick a vertex $v$. Check to see whether every other vertex is reachable from $v$
- Now see whether $v$ is reachable from every other vertex

**Analysis**
**Application: Deciding Strong Connectivity**

BFS can be used to determine whether a graph \( G = (V, E) \) is strongly connected.

- Observe: \( BFS(G, v) \) on a directed graph \( G \) will identify all vertices reachable from \( v \) by directed paths.
- Pick a vertex \( v \). Check to see whether every other vertex is reachable from \( v \).
- Now see whether \( v \) is reachable from every other vertex.

**Analysis**

- First step: one call to BFS: \( O(n + m) \) time.
Application: Deciding Strong Connectivity

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- Observe: $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths
- Pick a vertex $v$. Check to see whether every other vertex is reachable from $v$
- Now see whether $v$ is reachable from every other vertex

Analysis

- First step: one call to BFS: $O(n + m)$ time
- Second step: $n - 1$ calls to BFS: $O(n \times (n + m))$ time
**Application: Deciding Strong Connectivity**

BFS can be used to determine whether a graph $G = (V, E)$ is strongly connected.

- Observe: $BFS(G, v)$ on a directed graph $G$ will identify all vertices reachable from $v$ by directed paths.
- Pick a vertex $v$. Check to see whether every other vertex is reachable from $v$.
- Now see whether $v$ is reachable from every other vertex.

**Analysis**

- First step: one call to BFS: $O(n + m)$ time.
- Second step: $n - 1$ calls to BFS: $O(n \times (n + m))$ time.

Can we do better?
Application: Deciding Strong Connectivity

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.

Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.

So call BFS($G_{rev}, v$): Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis

• BFS($G, v$): $O(n + m)$ time

• Build $G_{rev}$: $O(n + m)$ time. [Do you believe this?]

• Depends on the data structure representing $G$!

• BFS($G_{rev}, v$): $O(n + m)$ time
Application: Deciding Strong Connectivity

Idea: Flip all the edges of \( G \) and call BFS on \( v \) again. Precisely

- Let \( G_{\text{rev}} = (V, E_{\text{rev}}) \), where \((u, v) \in E_{\text{rev}} \) if \((v, u) \in E\).
Application: Deciding Strong Connectivity

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$. 
Application: Deciding Strong Connectivity

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).
Application: Deciding Strong Connectivity

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis
**Application: Deciding Strong Connectivity**

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis

- $BFS(G, v)$: $O(n + m)$ time
**Application: Deciding Strong Connectivity**

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

**Analysis**

- $BFS(G, v)$: $O(n + m)$ time
- Build $G_{rev}$: $O(n + m)$ time. [Do you believe this?]
**Application: Deciding Strong Connectivity**

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

**Analysis**

- $BFS(G, v)$: $O(n + m)$ time
- Build $G_{rev}$: $O(n + m)$ time. [Do you believe this?]
  - Depends on the data structure representing $G$!
Application: Deciding Strong Connectivity

Idea: Flip all the edges of $G$ and call BFS on $v$ again. Precisely

- Let $G_{rev} = (V, E_{rev})$, where $(u, v) \in E_{rev}$ if $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{rev}$ iff there is a directed path from $u$ to $v$ in $G$.
- So call $BFS(G_{rev}, v)$: Every vertex is reachable from $v$ (in $G_{rev}$) if and only if $v$ is reachable from every vertex (in $G$).

Analysis

- $BFS(G, v)$: $O(n + m)$ time
- Build $G_{rev}$: $O(n + m)$ time. [Do you believe this?]
  - Depends on the data structure representing $G$!
- $BFS(G_{rev}, v)$: $O(n + m)$ time
Outline

Directed Graphs

Connectivity and Traversals in Directed Graphs

Applications

Deciding Strong Connectivity

DAGs and Topological Sorting
Application: Topological Sorting

Definition

A directed graph is acyclic (or a DAG) if it contains no directed cycles.
**Application: Topological Sorting**

**Definition**

A directed graph is *acyclic* (or a *DAG*) if it contains no directed cycles.

**Definition**

An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a directed graph $G = (V, E)$ is a *topological ordering* if every edge $(v_i, v_j) \in E$ satisfies $i < j$. 

Clearly, only a DAG can have a topological ordering! Do they always? Can we find one?
Application: Topological Sorting

**Definition**

A directed graph is **acyclic** (or a **DAG**) if it contains no directed cycles.

**Definition**

An ordering \( v_1, v_2, \ldots v_n \) of the vertices of a directed graph \( G = (V, E) \) is a **topological ordering** if every edge \((v_i, v_j) \in E\) satisfies \( i < j \).

Clearly, only a DAG can have a topological ordering!
Application: Topological Sorting

Definition
A directed graph is *acyclic* (or a *DAG*) if it contains no directed cycles.

Definition
An ordering $v_1, v_2, \ldots v_n$ of the vertices of a directed graph $G = (V, E)$ is a *topological ordering* if every edge $(v_i, v_j) \in E$ satisfies $i < j$.

Clearly, only a DAG can have a topological ordering!

Do they always?
Application: Topological Sorting

Definition
A directed graph is *acyclic* (or a *DAG*) if it contains no directed cycles.

Definition
An ordering $v_1, v_2, \ldots, v_n$ of the vertices of a directed graph $G = (V, E)$ is a *topological ordering* if every edge $(v_i, v_j) \in E$ satisfies $i < j$.

Clearly, only a DAG can have a topological ordering!

Do they always?

Can we find one?
DAG and Topological Order: An Example
DAG and Topological Order: An Example
Finding a Topological Order for a DAG

Claim
Every DAG $G$ has a vertex with in-degree (out-degree) 0
Finding a Topological Order for a DAG

**Claim**
Every DAG $G$ has a vertex with in-degree (out-degree) 0

**Proof.**
Consider a simple path of maximum length (# of edges)
$P = u = v_0, v_1, \ldots, v_n = v$. 

Finding a Topological Order for a DAG

Claim
Every DAG $G$ has a vertex with in-degree (out-degree) 0

Proof.
Consider a simple path of maximum length (# of edges)
$P = u = v_0, v_1, \ldots, v_n = v$.
Suppose there’s an edge $e = (w, u)$. Either $w$ is on $P$ or it isn’t
Finding a Topological Order for a DAG

Claim
Every DAG $G$ has a vertex with in-degree (out-degree) $0$

Proof.
Consider a simple path of maximum length ($\#$ of edges)
$P = u = v_0, v_1, \ldots, v_n = v$.
Suppose there’s an edge $e = (w, u)$. Either $w$ is on $P$ or it isn’t
- $w$ on $P$: Then $G$ contains a directed cycle. $\rightarrow\leftarrow$
Finding a Topological Order for a DAG

**Claim**
Every DAG $G$ has a vertex with in-degree (out-degree) 0

**Proof.**
Consider a simple path of maximum length ($\#$ of edges) $P = u = v_0, v_1, \ldots, v_n = v$.

Suppose there’s an edge $e = (w, u)$. Either $w$ is on $P$ or it isn’t
- $w$ on $P$: Then $G$ contains a directed cycle. $\rightarrow \leftarrow$
- $w$ not on $P$: Then I can make a longer simple path $\rightarrow \leftarrow$
Finding a Topological Order for a DAG

Claim
Every DAG $G$ has a vertex with in-degree (out-degree) 0

Proof.
Consider a simple path of maximum length (# of edges) $P = u = v_0, v_1, \ldots, v_n = v$.
Suppose there’s an edge $e = (w, u)$. Either $w$ is on $P$ or it isn’t
- $w$ on $P$: Then $G$ contains a directed cycle. →←
- $w$ not on $P$: Then I can make a longer simple path →←
So $u$ has in-degree 0. Same idea works for out-degree.....
Finding a Topological Order for a DAG

Claim
Every DAG $G$ has a vertex with in-degree (out-degree) 0

Proof.
Consider a simple path of maximum length (# of edges) $P = u = v_0, v_1, \ldots, v_n = v$.
Suppose there’s an edge $e = (w, u)$. Either $w$ is on $P$ or it isn’t
- $w$ on $P$: Then $G$ contains a directed cycle. $\rightarrow \leftarrow$
- $w$ not on $P$: Then I can make a longer simple path $\rightarrow \leftarrow$
So $u$ has in-degree 0. Same idea works for out-degree.....

Idea
Build order by repeatedly removing a vertex of in-degree 0 from $G$.  \[\square\]
Topological Sorting Algorithm

Algorithm 1 Topological Sorting

procedure TS(G) ▷ G = (V, E) is a DAG
    T[1..n] ← 0; i ← 0
    while V is not empty do
        i ← i + 1
        Find a vertex v ∈ V with indeg(v) = 0
        T[i] ← v
        Delete v (and its edges) from G
    end while
end procedure

Prove correctness by induction on n: If G is a DAG, so is G − v.
Finding v quickly

1. Compute in-degrees $ID[1..n]$ of all vertices (How would you do this?)
2. Scan $ID[]$ to produce a set $S$ of all vertices of in-degree 0: $O(n)$ time
3. Update $S$: When $v$ is deleted, decrement $ID[u]$ for each neighbor $u$; if $ID[u] = 0$, add $u$ to $S$: $O(outdeg(v))$ time
4. Total time for previous step over all vertices: $\sum_{v \in V} c \ast outdeg(v) = c \sum_{v \in V} outdeg(v) = c \ast m$: $O(m)$ time

Result: Topological Sorting takes $O(n + m)$ time and space!
Finding v quickly

1. Compute in-degrees $ID[1..n]$ of all vertices (How would you do this?)
Finding v quickly

1. Compute in-degrees $ID[1..n]$ of all vertices (How would you do this?)
2. Scan ID[] to produce a set $S$ of all vertices of in-degree 0: $O(n)$ time
Finding $v$ quickly

1. Compute in-degrees $ID[1..n]$ of all vertices (How would you do this?)
2. Scan $ID[]$ to produce a set $S$ of all vertices of in-degree 0: $O(n)$ time
3. Update $S$: When $v$ is deleted, decrement $ID[u]$ for each neighbor $u$; if $ID[u] = 0$, add $u$ to $S$: $O(outdeg(v))$ time
Finding $v$ quickly

1. Compute in-degrees $ID[1..n]$ of all vertices (How would you do this?)
2. Scan $ID[]$ to produce a set $S$ of all vertices of in-degree 0: $O(n)$ time
3. Update $S$: When $v$ is deleted, decrement $ID[u]$ for each neighbor $u$; if $ID[u] = 0$, add $u$ to $S$: $O(outdeg(v))$ time
4. Total time for previous step over all vertices:
   \[ \sum_{v \in V} c \times outdeg(v) = c \sum_{v \in V} outdeg(v) = c \times m: O(m) \] time
Finding \( v \) quickly

1. Compute in-degrees \( ID[1..n] \) of all vertices (How would you do this?)

2. Scan \( ID[] \) to produce a set \( S \) of all vertices of in-degree 0: \( O(n) \) time

3. Update \( S \): When \( v \) is deleted, decrement \( ID[u] \) for each neighbor \( u \); if \( ID[u] = 0 \), add \( u \) to \( S \): \( O(outdeg(v)) \) time

4. Total time for previous step over all vertices:
\[
\sum_{v \in V} c \ast outdeg(v) = c \sum_{v \in V} outdeg(v) = c \ast m: O(m)
\]

Result: Topological Sorting takes \( O(n + m) \) time and space!
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
- Find all cut vertices of a graph
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex \( v \)
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph

All of this can be done in $O(|V| + |E|)$ space and time!
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex \( v \)
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph
  - A bridge is an edge whose removal increases the number of connected components

All of this can be done in \( O(|V| + |E|) \) space and time!
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph
  - A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph

All of this can be done in $O(|V| + |E|)$ space and time!
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems:

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph
  - A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph
  - A biconnected component is a maximal subgraph having no cut vertices
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems:

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex $v$
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph
  - A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph
  - A biconnected component is a maximal subgraph having no cut vertices
- Find all strongly connected components of a directed graph

All of this can be done in $\mathcal{O}(|V| + |E|)$ space and time!
Further Applications of Graph Traversal

BFS and/or DFS can also be used to solve many other problems:

- Find a (directed) cycle in a (directed) graph
- As above, but containing a specified vertex \( v \)
- Find all cut vertices of a graph
  - A cut vertex is one whose removal increases the number of connected components
- Find all bridges of a graph
  - A bridge is an edge whose removal increases the number of connected components
- Find all biconnected components of a graph
  - A biconnected component is a maximal subgraph having no cut vertices
- Find all strongly connected components of a directed graph

All of this can be done in \( O(|V| + |E|) \) space and time!