Depth-First Search, Directed Graphs & Applications

Algorithm Design & Analysis

Spring 2019
Outline

Announcements

Quick Review of Breadth-First Search

Application: Deciding Bipartiteness

Depth-First Search

Directed Graphs

Connectivity and Traversals in Directed Graphs

Applications
  Deciding Strong Connectivity
  DAGs and Topological Sorting
(Many) Announcements

Some problem set items

• Problem Set 0 due now!
(Many) Announcements

Some problem set items

- Problem Set 0 due now!
- Problem Set 1 is now online
  - Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
(Many) Announcements

Some problem set items

- Problem Set 0 due now!
- Problem Set 1 is now online
  - Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
- Problem Set 0 is now available on the course GLOW site
  - This is the only content that will be placed on GLOW
(Many) Announcements

Some problem set items

- Problem Set 0 due now!
- Problem Set 1 is now online
  - Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
- Problem Set 0 is now available on the course GLOW site
  - This is the only content that will be placed on GLOW

Some scheduling items
(Many) Announcements

Some problem set items

• Problem Set 0 due now!
• Problem Set 1 is now online
  • Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
• Problem Set 0 is now available on the course GLOW site
  • This is the only content that will be placed on GLOW

Some scheduling items

• TA hours remain in SSL 030A (T/Th) and in TCL 206 (W)
  • From 7:00-11:00pm each evening
(Many) Announcements

Some problem set items

- Problem Set 0 due now!
- Problem Set 1 is now online
  - Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
- Problem Set 0 is now available on the course GLOW site
  - This is the only content that will be placed on GLOW

Some scheduling items

- TA hours remain in SSL 030A (T/Th) and in TCL 206 (W)
  - From 7:00-11:00pm each evening
- The "post-mid-term-exam" cancelled class meeting has been moved from Monday 4/8 to Friday 4/12
(Many) Announcements

Some problem set items

- Problem Set 0 due now!
- Problem Set 1 is now online
  - Hand in via my CS cubby (3rd floor of TCL) by 5 pm 2/15
- Problem Set 0 is now available on the course GLOW site
  - This is the only content that will be placed on GLOW

Some scheduling items

- TA hours remain in SSL 030A (T/Th) and in TCL 206 (W)
  - From 7:00-11:00pm each evening
- The "post-mid-term-exam" cancelled class meeting has been moved from Monday 4/8 to Friday 4/12
- I will be traveling during the week of 3/4-3/8
  - Class will still meet; guest lecturer will be Carl Rustad ’18
  - Attendance will be required
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
**BFS: An Example**

Three types of vertex: **unvisited**, **visited**, and **explored**.
**BFS: An Example**

Three types of vertex: unvisited, visited, and explored.
**BFS: An Example**

Three types of vertex: **unvisited**, **visited**, and **explored**.
**BFS: An Example**

Three types of vertex: unvisited, visited, and explored.
BFS: An Example

Three types of vertex: unvisited, visited, and explored.
Algorithm 1 Better Breadth-First Search of $G$ from vertex $r$

procedure BBFS2($G, r$)
Algorithm 2 Better Breadth-First Search of $G$ from vertex $r$

**procedure** $BBFS2(G, r)$

- Mark all $v \in V$ and all $e \in E$ as *unvisited*
- Initialize an empty queue $Q$

```plaintext
Mark all $v \in V$ and all $e \in E$ as *unvisited*
Initialize an empty queue $Q$
```
**Algorithm 3** Better Breadth-First Search of $G$ from vertex $r$

**procedure** BBFS2($G, r$)

- Mark all $v \in V$ and all $e \in E$ as *unvisited*
- Initialize an empty queue $Q$
- Mark $r$ as *visited*; $Q.enqueue(r)$
Algorithm 4 Better Breadth-First Search of $G$ from vertex $r$

**procedure** $\text{BBFS2}(G, r)$

Mark all $v \in V$ and all $e \in E$ as *unvisited*

Initialize an empty queue $Q$

Mark $r$ as *visited*; $Q$.enqueue($r$)

**while** $Q$ is not empty **do**

\[
\text{current} \leftarrow Q.\text{dequeue}()
\]

▷ current is now *explored*
Algorithm 5 Better Breadth-First Search of $G$ from vertex $r$

procedure BBFS2($G, r$)

Mark all $v \in V$ and all $e \in E$ as \textit{unvisited}

Initialize an empty queue $Q$

Mark $r$ as \textit{visited}; $Q$.enqueue($r$)

while $Q$ is not empty do

    $\text{current} \leftarrow Q$.dequeue() \quad \triangleright \text{current is now \textit{explored}}$

    for all neighbors $v$ of $\text{current}$ do


Algorithm 6 Better Breadth-First Search of $G$ from vertex $r$

procedure $BBFS2(G, r)$

Mark all $v \in V$ and all $e \in E$ as $\text{unvisited}$
Initialize an empty queue $Q$
Mark $r$ as $\text{visited}$; $Q$.enqueue($r$)

while $Q$ is not empty do

$\text{current} \leftarrow Q$.dequeue() \hfill \triangleright \text{current is now} \text{ explored}$

for all neighbors $v$ of $\text{current}$ do

if $v$ is $\text{unvisited}$ then

Mark $v$ as $\text{visited}$; $Q$.enqueue($v$)

end if

end for

end while

end procedure
Algorithm 7 Better Breadth-First Search of $G$ from vertex $r$

procedure BBFS2($G, r$)

Mark all $v \in V$ and all $e \in E$ as unvisited
Initialize an empty queue $Q$
Mark $r$ as visited; $Q$.enqueue($r$)

while $Q$ is not empty do

$\text{current} \leftarrow Q$.dequeue() $\triangleright$ current is now explored

for all neighbors $v$ of current do

if $v$ is unvisited then

Mark $v$ as visited; $Q$.enqueue($v$)

end if

if $\{\text{current}, v\}$ is unvisited then

Mark $\{\text{current}, v\}$ as visited

end if
Algorithm 8 Better Breadth-First Search of $G$ from vertex $r$

```
procedure BBFS2($G, r$)
    Mark all $v \in V$ and all $e \in E$ as unvisited
    Initialize an empty queue $Q$
    Mark $r$ as visited; $Q$.enqueue($r$)
    while $Q$ is not empty do
        $current \leftarrow Q$.dequeue() $\triangleright$ current is now explored
        for all neighbors $v$ of $current$ do
            if $v$ is unvisited then
                Mark $v$ as visited; $Q$.enqueue($v$)
            end if
            if $\{current, v\}$ is unvisited then
                Mark $\{current, v\}$ as visited
            end if
        end for
    end while
end procedure
```
Properties of BBFS

For a connected graph $G$
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it...
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - Ensures that each non-tree edge $e$ connects vertices whose levels differ by at most 1

Runs in optimal $O(n + m)$ time and space, even with all above tweaks!
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - Ensures that each non-tree edge $e$ connects vertices whose levels differ by at most 1
  - Constructs all of the connected components of a non-connected graph
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - **Ensures that each non-tree edge $e$ connects vertices whose levels differ by at most $1$**
  - Constructs all of the connected components of a non-connected graph
  - Provides shortest paths from every vertex back to $r$

- Runs in optimal $O(n + m)$ time and space, even with all above tweaks!
Properties of BBFS

For a connected graph $G$

- $BBFS2(G, r)$ visits every vertex and edge of $G$
- The queue $Q$ fulfills the role of the arrays $L[i]$ of levels in text.
- We can tweak $BBFS2$ so that it
  - Assigns each vertex a label (level) equal to its distance from $r$
  - Labels each edge as a tree-edge or a non-tree-edge
  - **Ensures that each non-tree edge $e$ connects vertices whose levels differ by at most 1**
  - Constructs all of the connected components of a non-connected graph
  - Provides shortest paths from every vertex back to $r$
- Runs in optimal $O(n + m)$ time and space, even with all above tweaks!
Application: Deciding Bipartiteness
Application: Deciding Bipartiteness

Definition

A bipartition of a set $X$ is a pair of subsets $X_1$, $X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$
Application: Deciding Bipartiteness

Definition
A bipartition of a set $X$ is a pair of subsets $X_1, X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$
Application: Deciding Bipartiteness

Definition

A bipartition of a set $X$ is a pair of subsets $X_1, X_2$ of $X$ such that

1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$.

Definition

A graph $G = (V, E)$ is bipartite if $V$ can be partitioned into two sets $V_1$ and $V_2$ so that every edge $e \in E$ has a vertex in each of $V_1$ and $V_2$. 
Application: Deciding Bipartiteness

Definition
A bipartition of a set $X$ is a pair of subsets $X_1$, $X_2$ of $X$ such that
1. $X_1 \cup X_2 = X$, and
2. $X_1 \cap X_2 = \emptyset$

A bipartition of $X$ is also called a partition of $X$ (into 2 parts) or a 2-coloring of $X$

Definition
A graph $G = (V, E)$ is bipartite if $V$ can be partitioned into two sets $V_1$ and $V_2$ so that every edge $e \in E$ has a vertex in each of $V_1$ and $V_2$.

Bipartite graphs are also called 2-colorable graphs.
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph G:

(a) G is bipartite
(b) Every circuit in G has even length
(c) No BFS tree has edges between vertices at the same level
(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Why? Take any BFS tree T.

• By (d), if T has no edge between vertices at the same level, then G is bipartite
• By (c), if T has some edge between vertices at the same level, then G is not bipartite
**Application: Deciding Bipartiteness**

*Theorem*

The following statements are equivalent for a connected graph $G$:

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at the same level
(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!
**Application: Deciding Bipartiteness**

**Theorem**

The following statements are equivalent for a connected graph $G$:

(a) $G$ is bipartite
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level
**Application: Deciding Bipartiteness**

**Theorem**

The following statements are equivalent for a connected graph $G$

1. $G$ is bipartite
2. Every circuit in $G$ has even length
3. No BFS tree has edges between vertices at same level
4. Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Why? Take any BFS tree $T$. 
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Why? Take any BFS tree $T$.

- By (d), if $T$ has no edge between vertices at same level, then $G$ is bipartite
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Why? Take any BFS tree $T$.
- By (d), if $T$ has no edge between vertices at same level, then $G$ is bipartite
- By (c), if $T$ has some edge between vertices at same level, then $G$ is not bipartite
Application: Deciding Bipartiteness

Theorem

The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$. 
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite

(b) Every circuit in $G$ has even length

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$.

(b) $\implies$ (c) Contradiction: Such an edge implies an odd circuit.
**Application: Deciding Bipartiteness**

**Theorem**
The following statements are equivalent for a connected graph $G$

1. $G$ is bipartite
2. Every circuit in $G$ has even length
3. No BFS tree has edges between vertices at same level
4. Some BFS tree has no edges between two vertices at the same level

**Proof.**

$(a) \implies (b)$ Vertices in circuit must alternate between $V_1$ and $V_2$.

$(b) \implies (c)$ Contradiction: Such an edge implies an odd circuit.

$(c) \implies (d)$ A rare, justified use of the term “obvious".
Application: Deciding Bipartiteness

Theorem
The following statements are equivalent for a connected graph $G$

(a) $G$ is bipartite
(b) Every circuit in $G$ has even length
(c) No BFS tree has edges between vertices at same level
(d) Some BFS tree has no edges between two vertices at the same level

Proof.

(a) $\implies$ (b) Vertices in circuit must alternate between $V_1$ and $V_2$.
(b) $\implies$ (c) Contradiction: Such an edge implies an odd circuit.
(c) $\implies$ (d) A rare, justified use of the term “obvious”.
(d) $\implies$ (a) Edges must span consecutive levels: levels provide bipartition of $G$. 
Implications of the Theorem
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$. 
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified $BFS$ still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + E|)$ time.
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + |E|)$ time
- Moreover, if $G$ is not bipartite, we can produce an odd circuit in $G$ as proof [Admire the awesomeness!]
Implications of the Theorem

So $G$ is bipartite iff no BFS tree for $G$ has two vertices at the same level that form an edge in $G$.

- When the BBFS algorithm visits an edge, we know the level of both of its endpoints.
- So when that edge is visited, if both ends have the same level, then STOP! $G$ is not bipartite.
- If the algorithm never discovers such an edge, $G$ is bipartite.
- This modified BFS still runs in $O(n + m)$ time.
- $G$ not connected? Run on each component: $O(|V| + |E|)$ time.
- Moreover, if $G$ is not bipartite, we can produce an odd circuit in $G$ as proof [Admire the awesomeness!]

Principle: Prefer algorithms that provide certificate of correctness!
**DFS: An Example**

Two types of vertex: *unvisited*, *visited*. **Green** is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
**DFS: An Example**

Two types of vertex: **unvisited**, **visited**. **Green** is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
**DFS: An Example**

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
DFS: An Example

Two types of vertex: unvisited, visited. Green is just for emphasis!
Recursive Depth-First Search

Algorithm 9 Depth-First Search of $G$ from vertex $r$

Require: all vertices are unvisited; $T = \{r\}$ is a 1-vertex tree

procedure DFS($G, r, T$)\hspace{1cm} $\triangleright G = (V, E)$
Algorithm 10 Depth-First Search of $G$ from vertex $r$

Require: all vertices are unvisited; $T = \{r\}$ is a 1-vertex tree

procedure $\text{DFS}(G, r, T)$ \hspace{1cm} $\triangleright G = (V, E)$

Mark $r$ as visited

for all neighbors $v$ of $r$ do
Recursive Depth-First Search

Algorithm 11 Depth-First Search of $G$ from vertex $r$

Require: all vertices are unvisited; $T = \{r\}$ is a 1-vertex tree

procedure DFS($G, r, T$)

Mark $r$ as visited

for all neighbors $v$ of $r$ do

if $v$ is unvisited then

Add $\{r, v\}$ to $T$

DFS($G, v, T$)

end if

end for

end procedure

\[ G = (V, E) \]
Recursive Depth-First Search

**Algorithm 12** Depth-First Search of $G$ from vertex $r$

**Require:** all vertices are unvisited; $T = \{r\}$ is a 1-vertex tree

```
procedure DFS(G, r, T)
    Mark $r$ as visited
    for all neighbors $v$ of $r$ do
        if $v$ is unvisited then
            Add $\{r, v\}$ to $T$
            DFS(G, v, T)
        end if
    end for
end procedure
```

**Ensure:** $T$ is a spanning tree for the component of $G$ containing $r$
Properties of DFS
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$. 
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
- $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
Properties of DFS

- When algorithm terminates, \( T \) forms a spanning tree having root \( r \) of the component of \( G \) containing \( r \).
  - \( T \) is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - \( T \) contains every vertex reachable from \( r \)
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
Properties of DFS

• When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
  • $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  • $T$ contains every vertex reachable from $r$
    • Induction on distance of reachable vertex from $r$
    • If $v$ is visited, so are its neighbors
• Now consider $T$ as a rooted tree with root $r
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$. 
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
- Now consider $T$ as a rooted tree with root $r$
  - Every vertex visited during a call to $DFS(G, v)$ is a descendent of $v$ in $T$
Properties of DFS

• When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
  • $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  • $T$ contains every vertex reachable from $r$
    • Induction on distance of reachable vertex from $r$
    • If $v$ is visited, so are its neighbors

• Now consider $T$ as a rooted tree with root $r$
  • Every vertex visited during a call to $DFS(G, v)$ is a descendent of $v$ in $T$
    • We consider any vertex to be (trivially) a descendent of itself
Properties of DFS

- When algorithm terminates, $T$ forms a spanning tree having root $r$ of the component of $G$ containing $r$.
  - $T$ is a tree because (i) it is connected and (ii) it has one more vertex than edge (see theorem from text)
  - $T$ contains every vertex reachable from $r$
    - Induction on distance of reachable vertex from $r$
    - If $v$ is visited, so are its neighbors
- Now consider $T$ as a rooted tree with root $r$
  - Every vertex visited during a call to $DFS(G, v)$ is a descendent of $v$ in $T$
    - We consider any vertex to be (trivially) a descendent of itself
  - For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$. 

The proof

For every edge \( e = \{u, v\} \) in \( G \), one of \( u \) or \( v \) is an ancestor of the other in \( T \).

Proof.
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
The proof

For every edge \( e = \{u, v\} \) in \( G \), one of \( u \) or \( v \) is an ancestor of the other in \( T \).

Proof.

- Clear if \( e \) is in \( T \), so assume not.
- Assume DFS is called on \( u \) before \( v \). When the For loop inspected \( v \), \( v \) must have been already visited.
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
- Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
  - Or else $v$ becomes a descendent of $u$
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
- Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
  - Or else $v$ becomes a descendent of $u$
- But $v$ wasn’t visited when DFS was called on $u$. 
The proof

For every edge $e = \{u, v\}$ in $G$, one of $u$ or $v$ is an ancestor of the other in $T$.

Proof.

- Clear if $e$ is in $T$, so assume not.
- Assume DFS is called on $u$ before $v$. When the For loop inspected $v$, $v$ must have been already visited.
  - Or else $v$ becomes a descendent of $u$
- But $v$ wasn’t visited when DFS was called on $u$.
- Thus $v$ was visited during the call $DFS(G, u)$ and so it’s a descendent of $u$. 

Algorithm 13 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)
Algorithm 14 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)
    Initialize an empty stack S; S.push(r)
    Initialize $T = \{r\}$ as a 1-vertex tree
Algorithm 15 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)
  Initialize an empty stack S; S.push(r)
  Initialize $T = \{r\}$ as a 1-vertex tree
  while S is not empty do
    $current \leftarrow S.pop()$
Algorithm 16 Depth-First Search Using a Stack

**Require:** all vertices are *unvisited*

**procedure** $\text{DFS}(G, r)$

1. Initialize an empty stack $S$; $S.push(r)$
2. Initialize $T = \{r\}$ as a 1-vertex tree
3. **while** $S$ is not empty **do**
   - $current \leftarrow S.pop()$
   - **if** $current$ is *unvisited* **then**
Algorithm 17 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)
    Initialize an empty stack $S$; $S.pop(r)$
    Initialize $T = \{r\}$ as a 1-vertex tree
    while $S$ is not empty do
        $current \leftarrow S.pop()$
        if $current$ is unvisited then
            Mark $current$ as visited
            for all neighbors $v$ of $current$ do
                $S.push(v)$; Add $\{current, v\}$ to $T$
            end for
        end if
    end while
end procedure
Algorithm 18 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)
    Initialize an empty stack S; S.push(r)
    Initialize $T = \{r\}$ as a 1-vertex tree
    while S is not empty do
        current ← S.pop()
        if current is unvisited then
            Mark current as visited
            for all neighbors v of current do
                if v is unvisited then
                    S.push(v); Add \{current, v\} to $T$
            end if
        end if
    end while
end procedure
Algorithm 19 Depth-First Search Using a Stack

Require: all vertices are unvisited

procedure DFS(G, r)

    Initialize an empty stack $S$; $S$.push($r$)
    Initialize $T = \{r\}$ as a 1-vertex tree

    while $S$ is not empty do
        current ← $S$.pop()
        if current is unvisited then
            Mark current as visited
            for all neighbors $v$ of current do
                if $v$ is unvisited then
                    $S$.push($v$); Add $\{current, v\}$ to $T$
                end if
            end for
        end if
    end while
end procedure
Directed Graphs

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

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 \( G \)

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

Example:

A directed walk in \( G \) is a sequence \( P = v_0, v_1, \ldots, v_n = v \) 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 \( v \).
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.
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 \( e_i = (v_{i-1}, v_i) \)
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 $\sim$ on a set $X$ is an equivalence relation on $X$ if $\sim$ has the following properties

**Reflexive** For all $x \in X$, $x \sim x$
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 $x, y \in X$, $x \simeq y \Leftrightarrow y \simeq x$
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 $\iff$ 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
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 $x \in S$, $x \in S_x$
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 $S$. 

**Equivalence Relations $\Leftrightarrow$ Equivalence Classes**
Equivalence Relations $\Leftrightarrow$ 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 $S_x$: $[x]$
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 $V$
**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
Mutual 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

Announcements

Quick Review of Breadth-First Search

Application: Deciding Bipartiteness

Depth-First Search

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 $v$. 

Analysis

- First step: one call to BFS: $O(n + m)$ time.
- Second step: $n - 1$ calls to BFS: $O((n^2 + nm))$ time.

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

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
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
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 $(v, u) \in E$. 

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 $(v, u) \in E$.
- Observe: There is a directed path from $v$ to $u$ in $G_{\text{rev}}$ 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

Announcements

Quick Review of Breadth-First Search

Application: Deciding Bipartiteness

Depth-First Search

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 \( i < j \).
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. \( \rightarrow \leftarrow \)
- \( w \) 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.....
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 $G$. 

**Topological Sorting Algorithm**

**Algorithm 20** Topological Sorting

```plaintext
procedure TS(G) ▷ \( G = (V, E) \) is a DAG
    \( T[1..n] \leftarrow 0; \ i \leftarrow 0 \)
    while \( V \) is not empty do
        \( i \leftarrow i + 1 \)
        Find a vertex \( v \in V \) with \( \text{indeg}(v) = 0 \)
        \( T[i] \leftarrow v \)
        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 $\nu$ 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 $\nu$ is deleted, decrement $ID[u]$ for each neighbor $u$; if $ID[u] = 0$, add $u$ to $S$: $O(outdeg(\nu))$ time
4. Total time for previous step over all vertices:
   $$\sum_{\nu \in V} c \times outdeg(\nu) = c \sum_{\nu \in V} outdeg(\nu) = c \times m: O(m)$$
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:
   \[ \sum_{v \in V} c \times outdeg(v) = c \sum_{v \in V} outdeg(v) = c \times m: \ O(m) \] time

Result: Topological Sorting takes \( O(n + m) \) time and space!