Algorithm Design
Graphs

Prof. Dr. Brahim Hnich

March 18, 2013

1

\(^1\)Slides based on those by Kevin Wayne. Copyright ©2005 Pearson-Addison Wesley. All rights reserved.
Basic definitions and concepts
Outline

- Basic definitions and concepts
- Graph Traversal
Outline

- Basic definitions and concepts
- Graph Traversal
- Testing Bipartiteness
Outline

- Basic definitions and concepts
- Graph Traversal
- Testing Bipartiteness
- Connectivity in Directed Graphs
Outline

- Basic definitions and concepts
- Graph Traversal
- Testing Bipartiteness
- Connectivity in Directed Graphs
- DAGs and Topological Ordering
Undirected graphs

- $G = (V, E)$

- $V$ = nodes.
- $E$ = edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: $n = |V|, m = |E|$. 

Prof. Dr. Brahim Hnich

Algorithm Design Graphs
Undirected graphs

- $G = (V, E)$
  - $V = \text{nodes.}$
  - $E = \text{edges between pairs of nodes.}$
Undirected graphs

\[ G = (V, E) \]

- \( V \) = nodes.
- \( E \) = edges between pairs of nodes.
- Captures pairwise relationship between objects.

Graph size parameters:

\( n = |V| \), \( m = |E| \).
Undirected graphs

- \( G = (V, E) \)
  - \( V \) = nodes.
  - \( E \) = edges between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: \( n = |V|, m = |E| \).
Undirected graphs

- \( G = (V, E) \)
  - \( V \) = nodes.
  - \( E \) = edges between pairs of nodes.
  - Captures pairwise relationship between objects.
  - Graph size parameters: \( n = |V|, m = |E| \).
# Graphs applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
Graph representation: adjacency matrix

- Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ iff $(u, v)$ is an edge.
Graph representation: adjacency matrix

- Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ iff $(u, v)$ is an edge.
  - Two representations of each edge.

- Space proportional to $n^2$.
- Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
Graph representation: adjacency matrix

- Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ iff $(u, v)$ is an edge.
  - Two representations of each edge.
  - Space proportional to $n^2$. 
Graph representation: adjacency matrix

- Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ iff $(u, v)$ is an edge.
  - Two representations of each edge.
  - Space proportional to $n^2$.
  - Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
Graph representation: adjacency matrix

- Adjacency matrix. $n$-by-$n$ matrix with $A_{uv} = 1$ iff $(u, v)$ is an edge.
  - Two representations of each edge.
  - Space proportional to $n^2$.
  - Checking if $(u, v)$ is an edge takes $\Theta(1)$ time.
  - Identifying all edges takes $\Theta(n^2)$ time.
Graph representation: adjacency matrix

```
Graph:
1 -- 2 -- 3
|     |     |
1     7
|     |     |
2     3
|     |     |
4     5
|     |     |
5     6
|     |     |
6     8

Adjacency Matrix:
<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Graph representation: adjacency list

- Adjacency list. Node indexed array of lists.
Graph representation: adjacency list

- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
Graph representation: adjacency list

- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to $n + m$.
Graph representation: adjacency list

- Adjacency list. Node indexed array of lists.
  - Two representations of each edge.
  - Space proportional to $n + m$.
  - Checking if $(u, v)$ is an edge takes $O(deg(u))$ time.
Adjacency list. Node indexed array of lists.

- Two representations of each edge.
- Space proportional to $n + m$.
- Checking if $(u, v)$ is an edge takes $O(deg(u))$ time.
- Identifying all edges takes $\Theta(m + n)$ time.
Graph representation: adjacency list
Path. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $(v_i, v_{i+1})$ is joined by an edge in $E$. 
Paths and connectivity

Path. A path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \) with the property that each consecutive pair \((v_i, v_{i+1})\) is joined by an edge in \( E \).

Simple path. A path is simple if all nodes are distinct.
Path. A path in an undirected graph $G = (V, E)$ is a sequence $P$ of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair $(v_i, v_{i+1})$ is joined by an edge in $E$.

Simple path. A path is simple if all nodes are distinct.

Connectedness. An undirected graph is connected if for every pair of nodes $u$ and $v$, there is a path between $u$ and $v$. 
Paths and connectivity
Cycle. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.
Cycle. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.
Cycle. A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first $k - 1$ nodes are all distinct.

Cycle $C = 1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 3 \rightarrow 1$
**Tree.** An undirected graph is a tree if it is connected and does not contain a cycle.
Tree. An undirected graph is a tree if it is connected and does not contain a cycle.
**Tree.** An undirected graph is a tree if it is connected and does not contain a cycle.

Cycle $C = 1 - 2 - 4 - 5 - 3 - 1$
**Rooted tree.** Given a tree $T$, choose a root node $r$ among the internal nodes and orient each edge away from $r$.
Rooted tree. Given a tree $T$, choose a root node $r$ among the internal nodes and orient each edge away from $r$. 
Phylogeny tree. Describe evolutionary history of species.
Phylogeny trees

Phylogeny tree. Describe evolutionary history of species.
s – t connectivity problem. Given two node s and t, is there a path between s and t?
Connectivity

$s - t$ connectivity problem. Given two node $s$ and $t$, is there a path between $s$ and $t$?

$s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?
Connectivity

$s - t$ connectivity problem. Given two node $s$ and $t$, is there a path between $s$ and $t$?

$s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Applications.

▶ Friendster: a social gaming site.
▶ Maze traversal.
▶ Kevin Bacon number.
▶ Fewest number of hops in a communication network.
Connectivity

$s - t$ connectivity problem. Given two node $s$ and $t$, is there a path between $s$ and $t$?

$s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Applications.

- Friendster: a social gaming site.
Connectivity

$s - t$ connectivity problem. Given two node $s$ and $t$, is there a path between $s$ and $t$?

$s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Applications.

- Friendster: a social gaming site.
- Maze traversal.
$s - t$ connectivity problem. Given two node $s$ and $t$, is there a path between $s$ and $t$?

$s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Applications.

- Friendster: a social gaming site.
- Maze traversal.
- Kevin Bacon number.
**Connectivity**

- **s − t connectivity problem.** Given two node \( s \) and \( t \), is there a path between \( s \) and \( t \)?
- **s − t shortest path problem.** Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?

**Applications.**
- Friendster: a social gaming site.
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
Breadth-first search

**BFS intuition.** Explore outward from $s$ in all possible directions, adding nodes one ”layer” at a time.
Breadth-first search

BFS intuition. Explore outward from $s$ in all possible directions, adding nodes one ”layer” at a time.
Breadth-first search

- BFS algorithm.

Theorem

For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \).

There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Breadth-first search

- BFS algorithm.

1. $L_0 = \{s\}$
Breadth-first search

- BFS algorithm.
  1. $L_0 = \{s\}$
  2. $L_1 =$ all neighbors of $L_0$

Theorem
For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$.
There is a path from $s$ to $t$ iff $t$ appears in some layer.
Breadth-first search

- BFS algorithm.
  1. \( L_0 = \{s\} \)
  2. \( L_1 = \) all neighbors of \( L_0 \)
  3. \( L_2 = \) all nodes that do not belong to \( L_0 \) or \( L_1 \), and that have an edge to a node in \( L_1 \)

▶ Theorem
For each \( i \), \( L_i \) consists of all nodes at distance exactly \( i \) from \( s \).
There is a path from \( s \) to \( t \) iff \( t \) appears in some layer.
Breadth-first search

BFS algorithm.

1. $L_0 = \{s\}$
2. $L_1 =$ all neighbors of $L_0$
3. $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$
4. $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$

Theorem

For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$.

There is a path from $s$ to $t$ iff $t$ appears in some layer.
Breadth-first search

- **BFS algorithm.**
  1. $L_0 = \{s\}$
  2. $L_1 =$ all neighbors of $L_0$
  3. $L_2 =$ all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$
  4. $L_{i+1} =$ all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$

- **Theorem**
  For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Property. Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
BFS(s):
    Set Discovered[s] = true and Discovered[v] = false for all other v
    Initialize $L[0]$ to consist of the single element $s$
    Set the layer counter $i = 0$
    Set the current BFS tree $T = \emptyset$
    While $L[i]$ is not empty
        Initialize an empty list $L[i+1]$
        For each node $u \in L[i]$
            Consider each edge $(u, v)$ incident to $u$
            If Discovered[v] = false then
                Set Discovered[v] = true
                Add edge $(u, v)$ to the tree $T$
                Add $v$ to the list $L[i+1]$
            Endif
        Endfor
        Increment the layer counter $i$ by one
    Endwhile
Theorem

The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.

- Easy to prove $O(n^2)$ running time:
Breadth-first search: analysis

Theorem
The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$,
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$
Theorem
The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.
- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs less than $n$ times
Breadth-first search: analysis

Theorem
The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs less than $n$ times
  - when we consider node $u$, there are less than $n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$, and the total time processing edges is $\sum_{u \in V} \deg(u) = 2m$
Breadth-first search: analysis

Theorem
The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs less than $n$ times
  - when we consider node $u$, there are less than $n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
Breadth-first search: analysis

Theorem

The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Proof.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs less than $n$ times
  - when we consider node $u$, there are less than $n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\text{deg}(u)$ incident edges $(u, v)$
Breadth-first search: analysis

Theorem

*The above implementation of BFS runs in \( O(m + n) \) time if the graph is given by its adjacency representation.*

Proof.

- **Easy to prove** \( O(n^2) \) running time:
  - at most \( n \) lists \( L[i] \)
  - each node occurs on at most one list; for loop runs less than \( n \) times
  - when we consider node \( u \), there are less than \( n \) incident edges \((u, v)\), and we spend \( O(1) \) processing each edge

- **Actually runs in** \( O(m + n) \) time:
  - when we consider node \( u \), there are \( \text{deg}(u) \) incident edges \((u, v)\)
  - total time processing edges is \( \sum_{u \in V} \text{deg}(u) = 2m \)
Connected component. Find all nodes reachable from $s$. 
Connected component. Find all nodes reachable from $s$. 

Connected component containing node $1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$. 
Connected component. Find all nodes reachable from $s$.

Connected component containing node $1 = \{1, 2, 3, 4, 5, 6, 7, 8\}$. 
Example: flood fill

**Flood fill.** Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.
Example: flood fill

Node: pixel
Edge: two neighboring lime pixels
Blob: connected component of lime pixels
Example: flood fill

Node: pixel
Edge: two neighboring lime pixels
Blob: connected component of lime pixels
Upon termination, \( R \) is the connected component containing \( s \).

\[
\begin{align*}
R \text{ will consist of nodes to which } s \text{ has a path} \\
\text{Initially } R = \{s\} \\
\text{While there is an edge } (u,v) \text{ where } u \in R \text{ and } v \notin R \\
\quad \text{Add } v \text{ to } R \\
\text{Endwhile}
\end{align*}
\]
Upon termination, $R$ is the connected component containing $s$. 

- $R$ will consist of nodes to which $s$ has a path.
- Initially $R = \{s\}$
- While there is an edge $(u, v)$ where $u \in R$ and $v \notin R$
  - Add $v$ to $R$
- Endwhile
DFS(s):
  Initialize S to be a stack with one element s
  While S is not empty
    Take a node u from S
    If Explored[u] = false then
      Set Explored[u] = true
      For each edge (u, v) incident to u
        Add v to the stack S
    Endfor
  Endif
Endwhile
Connected component: DFS example
Figure 3.5 The construction of a depth-first search tree $T$ for the graph in Figure 3.2, with (a) through (g) depicting the nodes as they are discovered in sequence. The solid edges are the edges of $T$; the dotted edges are edges of $G$ that do not belong to $T$. 
Definition: An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.
**Definition:** An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications:** Stable marriage: men = red, women = blue; Scheduling: machines = red, jobs = blue.
**Definition:** An undirected graph $G = (V, E)$ is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications:** Stable marriage: men = red, women = blue; Scheduling: machines = red, jobs = blue.
Testing bipartiteness. Given a graph $G$, is it bipartite?
Testing bipartiteness. Given a graph $G$, is it bipartite?

Many graph problems become
Testing bipartiteness. Given a graph $G$, is it bipartite?

Many graph problems become

- easier if the underlying graph is bipartite
  (matching)
Testing bipartiteness. Given a graph $G$, is it bipartite?

Many graph problems become

- easier if the underlying graph is bipartite (matching)
- tractable if the underlying graph is bipartite (independent set)
Testing bipartiteness. Given a graph $G$, is it bipartite?

Many graph problems become

- easier if the underlying graph is bipartite (matching)
- tractable if the underlying graph is bipartite (independent set)
Lemma

If a graph $G$ is bipartite, it cannot contain an odd length cycle.

Proof.

Not possible to 2-color the odd cycle, let alone $G$. 

bipartite (2-colorable)  

not bipartite (not 2-colorable)
**Lemma**

Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

- (i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
- (ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

![Case (i) and Case (ii) illustrations](image_url)
Proof.
(i) Suppose no edge joins two nodes in the same layer. By previous lemma, this implies all edges join nodes on adjacent level. Bipartition: red = nodes on odd levels, blue = nodes on even levels.

[Diagram showing a graph with nodes divided into layers L1, L2, L3, with red and blue nodes indicating the bipartition.]
Proof.

(ii) Suppose \((x, y)\) is an edge with \(x, y\) in same level \(L_j\). Let \(z\) be the lowest common ancestor of \((x, y)\). Let \(L_i\) be level containing \(z\). Consider cycle that takes edge from \(x\) to \(y\), then path from \(y\) to \(z\), then path from \(z\) to \(x\). Its length is \(1 + (j - i) + (j - i)\), which is odd

\[\square\]
Corollary

A graph $G$ is bipartite iff it contain no odd length cycle.

bipartite (2-colorable)

not bipartite (not 2-colorable)
Directed graphs

- Web graph - hyperlink points from one web page to another
Directed graphs

- Web graph - hyperlink points from one web page to another
  - Directedness of graph is crucial
Directed graphs

- Web graph - hyperlink points from one web page to another
  - Directedness of graph is crucial
  - Modern web search engines exploit hyperlink structure to rank web pages by importance
Directed graphs

- Web graph - hyperlink points from one web page to another
  - Directedness of graph is crucial
  - Modern web search engines exploit hyperlink structure to rank web pages by importance
Directed reachability. Given a node $s$, find all nodes reachable from $s$. 
Graph search

**Directed reachability.** Given a node $s$, find all nodes reachable from $s$.

**Directed $s-t$ shortest path problem.** Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?
Directed reachability. Given a node $s$, find all nodes reachable from $s$.

Directed $s - t$ shortest path problem. Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Graph search. BFS extends naturally to directed graphs.
Directed reachability. Given a node $s$, find all nodes reachable from $s$.

Directed $s - t$ shortest path problem. Given two node $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page $s$. Find all web pages linked from $s$, either directly or indirectly.
**Definition.** Node $u$ and $v$ are *mutually reachable* if there is a path from $u$ to $v$ and also a path from $v$ to $u$. 
Definiton. Node $u$ and $v$ are *mutually reachable* if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

Definition. A graph is strongly connected if every pair of nodes is mutually reachable.
Lemma

Let $s$ be any node. $G$ is strongly connected iff every node is reachable from $s$, and $s$ is reachable from every node.

Proof.

(⇒) Follows from definition.

(⇐) Path from $u$ to $v$: concatenate $u - s$ path with $s - v$ path.
Path from $v$ to $u$: concatenate $v - s$ path with $s - u$ path.
Theorem

Can determine if $G$ is strongly connected in $O(m + n)$ time.

Proof.

1. Pick any node $s$; (2) Run BFS from $s$ in $G$.
2. (3) Run BFS from $s$ in $G^{rev}$.\(^2\)
3. (4) Return true iff all nodes reached in both BFS executions.

\(^2\)reverse orientation of every edge in $G$
**Definition.** An *DAG* is a directed graph that contains no directed cycles. E.g. Precedence constraints: edge (vi, vj) means vi must precede vj.
Definition. An *DAG* is a directed graph that contains no directed cycles. E.g. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Definition. A *topological order* of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Definition. An DAG is a directed graph that contains no directed cycles. E.g. Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

Definition. A topological order of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Lemma

If $G$ has a topological order, then $G$ is a DAG.

Proof.

(by contradiction) Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge. By our choice of $i$, we have $i < j$. On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction.
Lemma

If $G$ has a topological order, then $G$ is a DAG.

Does every DAG have a topological ordering? If so, how do we compute one?
Lemma
If $G$ is a DAG, then $G$ has a node with no incoming edges.

Proof.
(by contradiction) Suppose that $G$ is a DAG and every node has at least one incoming edge. Let’s see what happens. Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$. Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$. Repeat until we visit a node, say $w$, twice. Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle.
Lemma

If $G$ is a DAG, then $G$ has a topological ordering.

Proof.
(by induction) Base case: true if $n = 1$. Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges. $G - v$ is a DAG, since deleting $v$ cannot create cycles. By inductive hypothesis, $G - v$ has a topological ordering. Place $v$ first in topological ordering; then append nodes of $G - v$ in topological order. This is valid since $v$ has no incoming edges.

Prof. Dr. Brahim Hnich

Algorithm Design | Graphs
Lemma

If $G$ is a DAG, then $G$ has a topological ordering.

To compute a topological ordering of $G$:
Find a node $v$ with no incoming edges and order it first
Delete $v$ from $G$
Recursively compute a topological ordering of $G - \{v\}$
and append this order after $v$
Topological sorting algorithm: example

Topological order:
Topological sorting algorithm: example

Topological order: $v_1$
Topological sorting algorithm: example

Topological order: $v_1, v_2$
Topological sorting algorithm: example

Topological order: $v_1, v_2, v_3$
Topological sorting algorithm: example

Topological order: $v_1, v_2, v_3, v_4$
Topological sorting algorithm: example

Topological order: $v_1, v_2, v_3, v_4, v_5$
Topological sorting algorithm: example

Topological order: $v_1, v_2, v_3, v_4, v_5, v_6$
Topological sorting algorithm: example

Figure 3.8 Starting from the graph in Figure 3.7, nodes are deleted one by one so as to be added to a topological ordering. The shaded nodes are those with no incoming edges; note that there is always at least one such edge at every stage of the algorithm’s execution.
Theorem

Algorithm finds a topological order in $O(m + n)$ time.

Proof.

Maintain the following information:

- $\text{count}[w]$ : remaining number of incoming edges
- $S$: set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and add $w$ to $S$ if $\text{count}[w]$ hits 0
  - this is $O(1)$ per edge