Outline

1. Graphs
2. Connectivity and Graph Traversal
   - Testing Bipartiteness
3. Topological Ordering
4. Properties of BFS and DFS trees
Examples of Graphs

Figure: Road Networks

Figure: Social Networks

Figure: Internet

Figure: Transition Graphs
(Undirected) Graph \( G = (V, E) \)

- \( V \): set of vertices (nodes);
- \( E \): pairwise relationships among \( V \);
  - (undirected) graphs: relationship is symmetric, \( E \) contains subsets of size 2
(Undirected) Graph \( G = (V, E) \)

- **\( V \):** set of vertices (nodes);
  - \( V = \{1, 2, 3, 4, 5, 6, 7, 8\} \)

- **\( E \):** pairwise relationships among \( V \);
  - (undirected) graphs: relationship is symmetric, \( E \) contains subsets of size 2
  - \( E = \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 4\}, \{2, 5\}, \{3, 5\}, \{3, 7\}, \{3, 8\}, \{4, 5\}, \{5, 6\}, \{7, 8\}\} \)
Directed Graph  $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - directed graphs: relationship is asymmetric, $E$ contains ordered pairs
Directed Graph $G = (V, E)$

- $V$: set of vertices (nodes);
  - $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $E$: pairwise relationships among $V$;
  - directed graphs: relationship is asymmetric, $E$ contains ordered pairs
    - $E = \{(1, 2), (1, 3), (3, 2), (4, 2), (2, 5), (5, 3), (3, 7), (3, 8), (4, 5), (5, 6), (6, 5), (8, 7)\}$
Abuse of Notations

- For (undirected) graphs, we often use \((i, j)\) to denote the set \(\{i, j\}\).
- We call \((i, j)\) an unordered pair; in this case \((i, j) = (j, i)\).

\[
E = \{(1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 7), (3, 8), (4, 5), (5, 6), (7, 8)\}
\]
Social Network: Undirected
Transition Graph: Directed
Road Network: Directed or Undirected
Internet: Directed or Undirected
Adjacency matrix

- \( n \times n \) matrix, \( A[u, v] = 1 \) if \((u, v) \in E\) and \( A[u, v] = 0 \) otherwise
- \( A \) is symmetric if graph is undirected
Adjacency matrix
- $n \times n$ matrix, $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise
- $A$ is symmetric if graph is undirected

Linked lists
- For every vertex $v$, there is a linked list containing all neighbours of $v$. 

Adjacency matrix

1: \[ 2 \rightarrow 3 \]
2: \[ 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \]
3: \[ 1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 8 \]
4: \[ 2 \rightarrow 5 \]
5: \[ 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \]
6: \[ 5 \]
7: \[ 3 \rightarrow 8 \]
8: \[ 3 \rightarrow 7 \]
**Comparison of Two Representations**

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td></td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- $n$: number of vertices
- $m$: number of edges, assuming $n - 1 \leq m \leq n(n - 1)/2$
- $d_v$: number of neighbors of $v$

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>$O(n^2)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>time to check $(u, v) \in E$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of $v$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u,v) \in E)</td>
<td>( O(1) )</td>
<td></td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparision of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u, v) \in E)</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u,v) \in E)</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td>( O(n) )</td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Two Representations

- Assuming we are dealing with undirected graphs
- \( n \): number of vertices
- \( m \): number of edges, assuming \( n - 1 \leq m \leq n(n - 1)/2 \)
- \( d_v \): number of neighbors of \( v \)

<table>
<thead>
<tr>
<th></th>
<th>Matrix</th>
<th>Linked Lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory usage</td>
<td>( O(n^2) )</td>
<td>( O(m) )</td>
</tr>
<tr>
<td>time to check ((u,v) \in E)</td>
<td>( O(1) )</td>
<td>( O(d_u) )</td>
</tr>
<tr>
<td>time to list all neighbours of ( v )</td>
<td>( O(n) )</td>
<td>( O(d_v) )</td>
</tr>
</tbody>
</table>
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Properties of BFS and DFS trees
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  
two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)

two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
Connectivity Problem

**Input:** graph $G = (V, E)$, (using linked lists)
  two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
- Breadth-First Search (BFS)
Connectivity Problem

**Input:** graph $G = (V,E)$, (using linked lists)
  
two vertices $s, t \in V$

**Output:** whether there is a path connecting $s$ to $t$ in $G$

- Algorithm: starting from $s$, search for all vertices that are reachable from $s$ and check if the set contains $t$
  - Breadth-First Search (BFS)
  - Depth-First Search (DFS)
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \cdots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers \( L_0, L_1, L_2, L_3, \ldots \)
- \( L_0 = \{s\} \)
- \( L_{j+1} \) contains all nodes that are not in \( L_0 \cup L_1 \cup \cdots \cup L_j \) and have an edge to a vertex in \( L_j \)
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Breadth-First Search (BFS)

- Build layers $L_0, L_1, L_2, L_3, \ldots$
- $L_0 = \{s\}$
- $L_{j+1}$ contains all nodes that are not in $L_0 \cup L_1 \cup \cdots \cup L_j$ and have an edge to a vertex in $L_j$
Implementing BFS using a Queue

**BFS(s)**

1. $head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. **while** $head \geq tail$ **do**
4. \hspace{1em} $v \leftarrow queue[tail], tail \leftarrow tail + 1$
5. \hspace{1em} **for** all neighbours $u$ of $v$ **do**
6. \hspace{2em} **if** $u$ is “unvisited” **then**
7. \hspace{3em} $head \leftarrow head + 1, queue[head] = u$
8. \hspace{3em} mark $u$ as “visited”

- Running time: $O(n + m)$. 
Example of BFS via Queue

[Diagram of a graph with nodes 1, 2, 3, 4, 5, 6, 7, 8, and arrows indicating connections]

[Diagram of a queue with a square box labeled '1' and arrows labeled 'head' and 'tail']
Example of BFS via Queue
Example of BFS via Queue

![Graph and Queue Diagram]

- The graph on the left illustrates a sample graph with vertices connected by edges.
- The vertex labeled with a red arrow is the starting point for the Breadth-First Search (BFS) algorithm.
- The queue on the right represents the order in which vertices are visited during BFS, with 'head' and 'tail' indicating the front and back of the queue, respectively.

The BFS process involves visiting all the vertices at the current level before moving on to the next level. The queue plays a crucial role in managing the order of vertices to be visited.
Example of BFS via Queue

\[ v \]

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & \\
\end{array}
\]

\[
\begin{array}{c}
1 \\
2 \\
3 \\
\end{array}
\]

head

tail
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

- V
- 1 2 3 4 5
- 6
- 7
- 8
- Head
- Tail
Example of BFS via Queue
Example of BFS via Queue

1. 2 3
2. 4 5
3. 7
4. 8
5. 6
6. head
7. tail
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

Graph representation:

```
1 2 3 4 5 7 8 6
```

Queue representation:

```
head
```

```
Example of BFS via Queue

1
2 3
4 5
7
8
6
head
tail
v
2 3 4 5 7 8 6
Example of BFS via Queue

1 2 3 4 5 7 8 6

head

tail
Example of BFS via Queue

```
  1 2 3 4 5 7 8 6
  ^   ^
head  tail
```

```
1 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
1 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```

```
  1
 2 3 4 5 7 8 6
   
  1
 2 3 4 5 7 8 6
```
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
**Depth-First Search (DFS)**

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back

![Graph Diagram]
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex (“dead-end”), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from $s$
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Depth-First Search (DFS)

- Starting from \( s \)
- Travel through the first edge leading out of the current vertex
- When reach an already-visited vertex ("dead-end"), go back
- Travel through the next edge
- If tried all edges leading out of the current vertex, go back
Implementing DFS using Recurrsion

**DFS(s)**
1. mark all vertices as “unvisited”
2. recursive-DFS(s)

**recursive-DFS(v)**
1. mark v as “visited”
2. for all neighbours u of v do
3. if u is unvisited then recursive-DFS(u)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Properties of BFS and DFS trees
Def. A graph $G = (V, E)$ is a bipartite graph if there is a partition of $V$ into two sets $L$ and $R$ such that for every edge $(u, v) \in E$, we have either $u \in L, v \in R$ or $v \in L, u \in R$. 
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)

\( \square \)
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)

Report “not a bipartite graph” if contradiction was found

If \( G \) contains multiple connected components, repeat above algorithm for each component
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- …
Testing Bipartiteness

- Taking an arbitrary vertex $s \in V$
- Assuming $s \in L$ w.l.o.g
- Neighbors of $s$ must be in $R$
- Neighbors of neighbors of $s$ must be in $L$
- ... 
- Report "not a bipartite graph" if contradiction was found
Testing Bipartiteness

- Taking an arbitrary vertex \( s \in V \)
- Assuming \( s \in L \) w.l.o.g
- Neighbors of \( s \) must be in \( R \)
- Neighbors of neighbors of \( s \) must be in \( L \)
- \( \ldots \)
- Report “not a bipartite graph” if contradiction was found
- If \( G \) contains multiple connected components, repeat above algorithm for each component
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness
Test Bipartiteness

bad edges!
Testing Bipartiteness using BFS

**BFS(s)**

1: \( \text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s \)

2: mark \( s \) as “visited” and all other vertices as “unvisited”

3: while \( \text{head} \geq \text{tail} \) do

4: \( v \leftarrow \text{queue}[\text{tail}], \text{tail} \leftarrow \text{tail} + 1 \)

5: for all neighbours \( u \) of \( v \) do

6: if \( u \) is “unvisited” then

7: \( \text{head} \leftarrow \text{head} + 1, \text{queue}[\text{head}] = u \)

8: mark \( u \) as “visited”
# Testing Bipartiteness using BFS

**test-bipartiteness**($s$)

1. $head \leftarrow 1, tail \leftarrow 1, queue[1] \leftarrow s$
2. mark $s$ as “visited” and all other vertices as “unvisited”
3. $color[s] \leftarrow 0$
4. **while** $head \geq tail$ **do**
5. \hspace{1em} $v \leftarrow queue[tail], tail \leftarrow tail + 1$
6. \hspace{1em} **for** all neighbours $u$ of $v$ **do**
7. \hspace{2em} **if** $u$ is “unvisited” **then**
8. \hspace{3em} $head \leftarrow head + 1, queue[head] = u$
9. \hspace{3em} mark $u$ as “visited”
10. \hspace{1em} $color[u] \leftarrow 1 - color[v]$
11. \hspace{1em} **else if** $color[u] = color[v]$ **then**
12. \hspace{2em} print(“$G$ is not bipartite”) and exit
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex $v \in V$ do
3:   if $v$ is “unvisited” then
4:       test-bipartiteness($v$)
5:   print(“$G$ is bipartite”)
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: for each vertex \( v \in V \) do
3:    if \( v \) is “unvisited” then
4:        test-bipartiteness(\( v \))
5:    print(“\( G \) is bipartite”)

Obs. Running time of algorithm = \( O(n + m) \)
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Properties of BFS and DFS trees
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \cdots, n\}$, so that
- if $(u, v) \in E$ then $\pi(u) < \pi(v)$

![Graph Diagram](image_url)
Topological Ordering Problem

**Input:** a directed acyclic graph (DAG) $G = (V, E)$

**Output:** 1-to-1 function $\pi : V \rightarrow \{1, 2, 3 \ldots, n\}$, so that

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- **Algorithm**: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- **Algorithm:** each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.
Topological Ordering

- Algorithm: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A: Use linked-lists of outgoing edges, maintain the in-degree of vertices, and maintain a queue (or stack) of vertices with $d_v = 0$. 


Algorithms: each time take a vertex without incoming edges, then remove the vertex and all its outgoing edges.

Q: How to make the algorithm as efficient as possible?

A:
- Use linked-lists of outgoing edges
- Maintain the in-degree $d_v$ of vertices
- Maintain a queue (or stack) of vertices $v$ with $d_v = 0$
topological-sort($G$)

1: let $d_v \leftarrow 0$ for every $v \in V$
2: for every $v \in V$ do
3: for every $u$ such that $(v, u) \in E$ do
4: $d_u \leftarrow d_u + 1$
5: $S \leftarrow \{v : d_v = 0\}$, $i \leftarrow 0$
6: while $S \neq \emptyset$ do
7: $v \leftarrow$ arbitrary vertex in $S$, $S \leftarrow S \setminus \{v\}$
8: $i \leftarrow i + 1$, $\pi(v) \leftarrow i$
9: for every $u$ such that $(v, u) \in E$ do
10: $d_u \leftarrow d_u - 1$
11: if $d_u = 0$ then add $u$ to $S$
12: if $i < n$ then output “not a DAG”

- $S$ can be represented using a queue or a stack
- Running time $= O(n + m)$
### $S$ as a Queue or a Stack

<table>
<thead>
<tr>
<th>DS</th>
<th>Queue</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialization</td>
<td>$\text{head} \leftarrow 0$, $\text{tail} \leftarrow 1$</td>
<td>$\text{top} \leftarrow 0$</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>$\text{head} \geq \text{tail}$</td>
<td>$\text{top} &gt; 0$</td>
</tr>
<tr>
<td>Add($\nu$)</td>
<td>$\text{head} \leftarrow \text{head} + 1$, $S[\text{head}] \leftarrow \nu$</td>
<td>$\text{top} \leftarrow \text{top} + 1$, $S[\text{top}] \leftarrow \nu$</td>
</tr>
<tr>
<td>Retrieve $\nu$</td>
<td>$\nu \leftarrow S[\text{tail}]$, $\text{tail} \leftarrow \text{tail} + 1$</td>
<td>$\nu \leftarrow S[\text{top}]$, $\text{top} \leftarrow \text{top} - 1$</td>
</tr>
</tbody>
</table>
Example

![Diagram of a graph with nodes labeled a, b, c, d, e, f, and g, and edges connecting these nodes.]

<table>
<thead>
<tr>
<th>queue:</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

A graph with nodes labeled a, b, c, d, e, f, and g, connected by arrows, indicating the direction of the edges. The diagram also includes a queue labeled as follows:

- **Queue**: a
- **Head**
- **Tail**

A table is shown with two columns:

<table>
<thead>
<tr>
<th>degree</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

**Diagram:**
- Nodes: $e$, $b$, $d$, $c$, $f$, $g$
- Edges: $e ightarrow f$, $f ightarrow g$, $g ightarrow e$, $b ightarrow e$, $d ightarrow e$, $d ightarrow c$, $c ightarrow d$

**Queue:**
- Head: $a$
- Tail: $g$

**Degree Table:**

<table>
<thead>
<tr>
<th></th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>degree</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example
Example
Example

```
Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Example

Exa
Example

![Diagram of a graph with nodes labeled a, b, c, d, e, f, and g connected by arrows.]

- Degree distribution:
  - Node a: 0
  - Node b: 0
  - Node c: 0
  - Node d: 1
  - Node e: 1
  - Node f: 1
  - Node g: 3

- Queue:
  - head: a
  - tail: g

---

28/37
Example

- Diagram showing a graph with nodes labeled as follows: e -> d, e -> g, d -> f, g -> f.

- Queue representation: head ▶ a ▶ b ▶ c ▶ tail

- Degree sequence: a: 0, b: 0, c: 0, d: 1, e: 0, f: 1, g: 3.
Example

![Diagram of a graph with nodes e, d, f, and g, and arrows between them.]

Queue:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>f</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Degree:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

![Graph Diagram]

Queue:

<table>
<thead>
<tr>
<th>queue:</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>f</td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Degree:

<table>
<thead>
<tr>
<th>degree</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
</table>
Example

queue:

\[
\begin{array}{ccccccc}
\text{head} & a & b & c & d & f & \text{tail} \\
\end{array}
\]

degree:

\[
\begin{array}{cccccccc}
\text{degree} & a & b & c & d & e & f & g \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
\end{array}
\]
Example

queue: [a, b, c, d, f, e]

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>degree</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
Example

queue: $\begin{array}{cccccc}
a & b & c & d & f & e \\
\end{array}$

<table>
<thead>
<tr>
<th>degree</th>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

head

tail

degree

$e \rightarrow g \rightarrow f$
Example

queue:

\[
\begin{array}{ccccccc}
 a & b & c & d & f & e \\
\hline
\text{head} & \text{tail} \\
\end{array}
\]

degree

\[
\begin{array}{ccccccc}
 a & b & c & d & e & f & g \\
\hline
\text{degree} & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array}
\]

\begin{tikzpicture}
  \node (e) at (0,0) [shape=circle,draw=black,thick] {$e$};
  \node (g) at (1,0) [shape=circle,draw=black,thick] {$g$};
  \draw[->, thick] (e) -- (g);
\end{tikzpicture}
Example
Example

queue: $a \quad b \quad c \quad d \quad f \quad e$

<table>
<thead>
<tr>
<th>$a$</th>
<th>$b$</th>
<th>$c$</th>
<th>$d$</th>
<th>$e$</th>
<th>$f$</th>
<th>$g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

degree
Example

queue: [a, b, c, d, f, e, g]

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

degree
Example

queue: [a, b, c, d, f, e, g]

degree: [0, 0, 0, 0, 0, 0, 0]
Outline

1. Graphs

2. Connectivity and Graph Traversal
   - Testing Bipartiteness

3. Topological Ordering

4. Properties of BFS and DFS trees
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$.

- A vertical edge $(u, v)$, $u \geq 2$ levels above $v$, is not possible. $v$ should be a child of $u$.
- A horizontal edge $(u, v)$, $u \geq 2$ levels above $v$, is not possible. $v$ should be a child of $u$.
- A horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$, is not possible. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

Can there be a vertical edge $(u,v)$, $u \geq 2$ levels above $v$? No. $v$ should be a child of $u$

Can there be a horizontal edge $(u,v)$, $u \geq 2$ levels above $v$? No. $v$ should be a child of $u$.

Can there be a horizontal edge $(u,v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$? No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

Can there be a vertical edge $(u,v)$, $u \geq 2$ levels above $v$?
No. $v$ should be a child of $u$.

Can there be a horizontal edge $(u,v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$?
No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?

- No. $v$ should be a child of $u$.

- Can there be a **horizontal edge** $(u, v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$?

- No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a vertical edge $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a vertical edge $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$

- Can there be a horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$?
  - No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$
- Can there be a **horizontal edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$. 

![Diagram of BFS tree with vertices and edges labeled](image)
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$

- Can there be a **horizontal edge** $(u, v)$ $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$

- Can there be a **horizontal edge** $(u, v)$ $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v), u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$

- Can there be a **horizontal edge** $(u, v) u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$.

- Can there be a horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$?
  - No. $v$ should be a child of $u$. 

Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$

- Can there be a **horizontal edge** $(u, v)$, $u \geq 2$ levels above $v$?
  - No. $v$ should be a child of $u$.

- Can there be a horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$'s parent is to the right of $u$?
  - No. $v$ should be a child of $u$. 
Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$

- Can there be a **vertical edge** $(u, v)$, $u \geq 2$ levels above $v$?  
  - No. $v$ should be a child of $u$.

- Can there be a **horizontal edge** $(u, v)$, $u \geq 2$ levels above $v$?  
  - No. $v$ should be a child of $u$.

- Can there be a horizontal edge $(u, v)$, where $u$ is 1 level above $v$, but $v$’s parent is to the right of $u$?  
  - No. $v$ should be a child of $u$.  

Properties of a BFS Tree

Given a BFS tree $T$ of a connected graph $G$, other than the tree edges, we only have horizontal edges $(u, v)$, where

- either $u$ and $v$ are at the same level
- or $u$ is 1 level above $v$, and $v$’s parent is to the left of $u$, (or vice versa)
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, 

No.

All non-tree edges are vertical edges.

A vertical edge $(u,v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Given a tree DFS tree $T$ of a graph (connected) $G$, All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, no horizontal edge $(u,v)$ exists. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its tree edges on the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Given a tree DFS tree $T$ of a graph (connected) $G$, no horizontal edges $(u,v)$ exist. All non-tree edges are vertical edges. Any vertical edge $(u,v)$ and its tree edges form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, there cannot be a horizontal edge $(u, v)$. All non-tree edges are vertical edges. A vertical edge $(u, v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, no horizontal edge $(u,v)$ can exist. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, all non-tree edges are vertical edges. A vertical edge $(u,v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, no horizontal edge $(u, v)$ exists. All non-tree edges are vertical edges. A vertical edge $(u, v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, 

No. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,
Can there be a horizontal edge \((u,v)\)?

No. All non-tree edges are vertical edges.

A vertical edge \((u,v)\) and its the edges in the path from \(u\) to \(v\) in \(T\) form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, there can be no horizontal edge $(u,v)$. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Given a tree DFS tree $T$ of a graph (connected) $G$, No. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, there cannot be a horizontal edge $(u,v)$. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its neighbors form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, no horizontal edge $(u,v)$ exists. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, there cannot be a horizontal edge $(u,v)$. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$, there cannot be a horizontal edge $(u,v)$. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its incident edges form a cycle; we call it a canonical cycle.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
- No.
Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
  - No.
Given a tree DFS tree $T$ of a graph (connected) $G$, 

- Can there be a horizontal edge $(u, v)$? 
- No.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
- No.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
- No.
- All non-tree edges are vertical edges.
Properties of a DFS Tree

Given a tree DFS tree $T$ of a graph (connected) $G$,

- Can there be a horizontal edge $(u, v)$?
- No.
- All non-tree edges are vertical edges.
- A vertical edge $(u, v)$ and its the edges in the path from $u$ to $v$ in $T$ form a cycle; we call it a canonical cycle.
Lemma  If $G$ contains a cycle, then it has a canonical cycle.
**Lemma** If $G$ contains a cycle, then it has a canonical cycle.

**Proof.**
- If $G$ contains a cycle, then it must have at least one non-tree edge.
- W.r.t DFS tree $T$, we can only have vertical + tree edges.
- There is at least one vertical edge.
- There is a canonical cycle.
Lemma  If $G$ contains a cycle, then it has a canonical cycle.

Proof.
- If $G$ contains a cycle, then it must have at least one non-tree edge.
- W.r.t DFS tree $T$, we can only have vertical + tree edges
- $\exists$ at least one vertical edge
- There is a canonical cycle
- There might or might not be non-canonical ones.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

Can there be a horizontal (directed) edge $(u,v)$ where $u$ is visited before $v$? No. However, there can be horizontal edges $(u,v)$ where $u$ is visited after $v$. 
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$, there cannot be a horizontal (directed) edge $(u,v)$ where $u$ is visited before $v$. However, there can be horizontal edges $(u,v)$ where $u$ is visited after $v$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

No.

However, there can be horizontal edges $(u,v)$ where $u$ is visited after $v$. 
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

No. However, there can be horizontal edges $(u,v)$ where $u$ is visited after $v$. 
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$,
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
- No.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
- No.

![Diagram of DFS tree with starting vertex $s^*$ and directed edges]
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
- No.
Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
- No.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
- No.
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Can there be a horizontal (directed) edge $(u, v)$ where $u$ is visited before $v$?
  - No.

- However, there can be horizontal edges $(u, v)$ where $u$ is visited after $v$. 
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Other than tree edges, there are two types of edges:
  - vertical edges directed to ancestors
  - horizontal edges $(u, v)$ where $u$ is visited after $v$. 
Properties of a DFS Tree Over a Directed Graph

Given a tree DFS tree $T$ of a directed graph $G$, assuming all vertices can be reached from the starting vertex $s^*$

- Other than tree edges, there are two types of edges:
  - vertical edges directed to ancestors
  - horizontal edges $(u, v)$ where $u$ is visited after $v$.

- An vertical edge $(u, v)$ and the tree edges in the tree path from $v$ to $u$ form a cycle, and we call it a **canonical cycle**.
**Lemma**  If there is a cycle in the directed graph $G$, then there must be a canonical one.

**Proof.**
Properties of a DFS Tree Over a Directed Graph

**Lemma** If there is a cycle in the directed graph $G$, then there must be a canonical one.

**Proof.**
- Focus on tree edges and horizontal edges
Lemma If there is a cycle in the directed graph $G$, then there must be a canonical one.

Proof.
- Focus on tree edges and horizontal edges.
- Post-order-traversal of $T$ gives a reversed topological ordering.
Lemma If there is a cycle in the directed graph $G$, then there must be a canonical one.

Proof.
- Focus on tree edges and horizontal edges
- post-order-traversal of $T$ gives a reversed topological ordering
- Without vertical edges, $G$ has no cycles
Lemma  If there is a cycle in the directed graph \( G \), then there must be a canonical one.

Proof.

- Focus on tree edges and horizontal edges
- post-order-traversal of \( T \) gives a reversed topological ordering
- Without vertical edges, \( G \) has no cycles
- Again, there might be non-canonical cycles.
Algorithm 1 Check-Cycle-Directed

1: add a source $s^*$ to $G$ and edges from $s^*$ to all other vertices.
2: $\text{visited} \leftarrow \text{boolean array over } V$, with $\text{visited}[v] = \text{false}, \forall v$
3: $\text{instack} \leftarrow \text{boolean array over } V$, with $\text{instack}[v] = \text{false}, \forall v$
4: DFS($s^*$)
5: return “no cycle”

Algorithm 2 DFS($v$)

1: $\text{visited}[v] \leftarrow \text{true}, \text{instack}[v] \leftarrow \text{true}$
2: for every outgoing edge $(v, u)$ of $v$ do
3: if $\text{inqueue}[u]$ then $\triangleright$ Find a vertical edge
4: exit the whole algorithm, by returning “there is a cycle”
5: else if $\text{visited}[u] = \text{false}$ then
6: DFS($u$)
7: $\text{instack}[v] \leftarrow \text{false}$