CSE 431/531: Algorithm Analysis and Design (Spring 2022)

Graph Basics

Lecturer: Shi Li

Department of Computer Science and Engineering
University at Buffalo
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
Representation of Graphs

- **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 $v$. 

![Graph Diagram]

1: $\begin{array}{c} 2 \rightarrow 3 \\ 4 \rightarrow 5 \end{array}$
2: $\begin{array}{c} 1 \rightarrow 3 \rightarrow 4 \rightarrow 5 \end{array}$
3: $\begin{array}{c} 1 \rightarrow 2 \rightarrow 5 \rightarrow 7 \rightarrow 8 \end{array}$
4: $\begin{array}{c} 2 \rightarrow 5 \end{array}$
5: $\begin{array}{c} 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \end{array}$
6: $\begin{array}{c} 5 \end{array}$
7: $\begin{array}{c} 3 \rightarrow 8 \end{array}$
8: $\begin{array}{c} 3 \rightarrow 7 \end{array}$
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>( 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>
### 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>
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></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. \(\text{head} \leftarrow 1, \text{tail} \leftarrow 1, \text{queue}[1] \leftarrow s\)
2. mark \(s\) as “visited” and all other vertices as “unvisited”
3. \(\text{while} \ \text{head} \geq \text{tail} \ \text{do}\)
4. \(v \leftarrow \text{queue}[\text{tail}], \text{tail} \leftarrow \text{tail} + 1\)
5. \(\text{for} \ \text{all} \ \text{neighbours} \ u \ \text{of} \ v \ \text{do}\)
6. \(\text{if} \ u \ \text{is “unvisited” then}\)
7. \(\text{head} \leftarrow \text{head} + 1, \text{queue}[\text{head}] = u\)
8. \(\text{mark} \ u \ \text{as “visited”}\)

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

1. Enqueue node 2 (the root).
2. Dequeue node 2 and process it.
3. Enqueue nodes 1, 3, 4, 5, and 6.
4. Dequeue node 1 (next level) and process it.
5. Enqueue nodes 7 and 8.
6. Dequeue node 3 (next level) and process it.
7. Dequeue node 4 and process it.
8. Dequeue node 5 and process it.
9. Dequeue node 6 and process it.
10. Dequeue node 7 and process it.
11. Dequeue node 8 and process it.

The queue is represented as follows:

Head: 1
Tail: 5

Example of BFS via Queue
Example of BFS via Queue

A depth-first search is useful when you need to explore a graph, but you don't know how far you need to go. It's useful if you only need to explore a small portion of the graph, or if the graph is too large to explore in its entirety.

BFS (Breadth-First Search) is a useful algorithm for exploring a graph. It's useful when you need to explore the graph in a systematic way, and you don't care if you get stuck in a loop.

BFS is implemented using a queue. The algorithm starts at a root node, and then explores all of its neighbors before moving on to the next level of nodes. This continues until the queue is empty.

Here's an example of how BFS works:

1. Start at node 1
2. Visit node 2, then node 3
3. Visit node 4, then node 5
4. Visit node 6
5. Visit node 7
6. Visit node 8

The queue looks like this:

```
head <- 1 <- 2 <- 3 <- 4 <- 5 <- 6 <- 7 <- 8 <- tail
```

Here's a diagram of the graph:

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

And here's a diagram of the queue:

```
1 2 3 4 5 6 7
```
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue
Example of BFS via Queue

[Diagram showing a graph and a queue with nodes 1, 2, 3, 4, 5, 7, 8, 6 in a queue order with arrows indicating the head and tail.]
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 illustration of Depth-First Search](image)
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
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 Recursion

**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
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 $R$

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 \( L \)
- \( \ldots \)
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
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

\[ \text{BFS}(s) \]

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. \( \text{while} \ \text{head} \geq \text{tail} \ \text{do} \)
4. \( \ v \leftarrow \text{queue}[\text{tail}], \text{tail} \leftarrow \text{tail} + 1 \)
5. \( \text{for} \ \text{all neighbours} \ u \ \text{of} \ v \ \text{do} \)
6. \( \ \text{if} \ u \ \text{is “unvisited” then} \)
7. \( \ \text{head} \leftarrow \text{head} + 1, \text{queue}[\text{head}] = u \)
8. \( \ \text{mark} \ u \ \text{as “visited”} \)
Testing Bipartiteness using BFS

test-bipartiteness(s)

1:  head ← 1, tail ← 1, queue[1] ← s
2:  mark s as “visited” and all other vertices as “unvisited”
3:  color[s] ← 0
4:  while head ≥ tail do
5:     v ← queue[tail], tail ← tail + 1
6:     for all neighbours u of v do
7:         if u is “unvisited” then
8:             head ← head + 1, queue[head] = u
9:             mark u as “visited”
10:    end if
11:    else if color[u] = color[v] then
12:        print(“G is not bipartite”) and exit
Testing Bipartiteness using BFS

1: mark all vertices as “unvisited”
2: \textbf{for} each vertex $v \in V$ \textbf{do}
3: \quad \textbf{if} $v$ is “unvisited” \textbf{then}
4: \quad \quad \text{test-bipartiteness}(v)
5: \quad \text{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 \( \pi : V \to \{1, 2, 3 \ldots, n\} \), so that

- if \((u, v) \in E\) then \(\pi(u) < \pi(v)\)
Topological Ordering Problem

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

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

- if $(u, v) \in E$ then $\pi(u) < \pi(v)$
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.
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.
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.
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.

**Q:** How to make the algorithm as efficient as possible?
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 $d_v$ 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>( head \leftarrow 0, tail \leftarrow 1 )</td>
<td>( top \leftarrow 0 )</td>
</tr>
<tr>
<td>Non-Empty?</td>
<td>( head \geq tail )</td>
<td>( top &gt; 0 )</td>
</tr>
<tr>
<td>Add(( v ))</td>
<td>( head \leftarrow head + 1 ) '&lt;br&gt;( S[head] \leftarrow v )</td>
<td>( top \leftarrow top + 1 ) '&lt;br&gt;( S[top] \leftarrow v )</td>
</tr>
<tr>
<td>Retrieve ( v )</td>
<td>( v \leftarrow S[tail] ) '&lt;br&gt;( tail \leftarrow tail + 1 )</td>
<td>( v \leftarrow S[top] ) '&lt;br&gt;( top \leftarrow top - 1 )</td>
</tr>
</tbody>
</table>
Example

```
  \[\text{queue:} \begin{array}{c}
    a \\
    b \\
    c \\
    d \\
    e \\
    f \\
    g \\
  \end{array}\]
```

```
  \[\begin{array}{c|ccccccc}
    \text{head} & a & b & c & d & e & f & g \\
    \text{tail} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} & \text{null} \\
    \text{degree} & 0 & 1 & 1 & 1 & 2 & 1 & 3 \\
  \end{array}\]
```
Example

The image shows a graph with nodes labeled 'a', 'b', 'c', 'd', 'e', 'f', and 'g', connected by arrows. The graph is a directed graph with edges connecting 'a' to 'b', 'c', and 'd'; 'b' to 'c' and 'e'; 'c' to 'd' and 'f'; 'd' to 'e' and 'f'; 'e' to 'g'; and 'f' to 'g'.

Below the graph, there is a queue represented as a list: 'a', 'b', 'c', 'd', 'e', 'f', 'g'.

A table shows the degree of each node:

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

The 'head' of the queue points to 'a', and the 'tail' points to 'g'.
Example

The diagram shows a directed graph with vertices labeled from 'a' to 'g'. The graph includes edges from 'b' to 'c', 'c' to 'd', 'd' to 'f', 'f' to 'g', and 'g' to 'e'.

The queue is represented as a list: 'a'.

The degree table is as follows:

<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>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

[Diagram with labeled nodes and edges]

Queue:

- head
- tail

<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>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Degree:

- b
- c
- d
- e
- f
- g

[Table with node degrees]
Example

![Graph Example]

- **Queue:** `a b c`
- **Degree:**
  - `a`: 0
  - `b`: 0
  - `c`: 0
  - `d`: 1
  - `e`: 2
  - `f`: 1
  - `g`: 3

**Head** and **Tail** indicators are shown.

---

28/37
Example

```
- e
  └── d
    └── c
- f
- g
  └── head
      └── queue: a b c
        └── degree: 0 0 0 1 1 1 3
      └── tail
```
Example

```
Example

Example
```
Example

queue: a b c

degree: 0 0 0 0 1 0 3

head

tail
Example

```
queue: a b c d f

degree | a | b | c | d | e | f | g
-------|---|---|---|---|---|---|---
     0 | 0 | 0 | 0 | 1 | 0 | 3

head

tail

```

```
Example

**Diagram:**
- Nodes: e, d, f, g
- Edges: e → d, e → f, d → g, f → g

**Queue:**
- Structure: a, b, c, d, f
  - Head: a
  - Tail: f

**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>deg</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

**Diagram:**
- Nodes: `e`, `g`, `f`
- Edges: `e` to `g`, `f`

**Queue:**
- `head`: `a`, `b`, `c`, `d`, `f`, ...
- `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>deg</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
Example
Example

queue: \[ \begin{array}{ccccccc}
& a & b & c & d & f & e \\
\end{array} \]

degree

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

\[ \begin{array}{cccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{array} \]
Example

\[ \text{queue:}\quad a \quad b \quad c \quad d \quad f \quad e \]

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

queue: a b c d f 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: g

head

tail
Example

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

| degree | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

\( g \)
Example

Queue: 

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

Degree: 

<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>

Head: 

Tail:
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 $G$
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 \( G \)
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 $G$
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 $G$

- 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** $(u, v)$, $u \geq 2$ 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 $u$.  

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.
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) $G$,
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 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) $G$,
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) $G$, 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) $G$, 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) $G$, 

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) $G$, 

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) $G$, 

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) $G$, 

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) $G$, no horizontal edge $(u,v)$ exists. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and 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) $G$, 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.
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 edges are present. All non-tree edges are vertical edges. A vertical edge $(u,v)$ and its ancestor-descendant 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 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.
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) $G$,

- 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.
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 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
**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
- ∃ 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 $s^*$,

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 $v$. 
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 \( s^* \)

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 $s^*$.
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 $s^*$.
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 $s^*$

No. However, there can be horizontal edges $(u,v)$ where $u$ is visited after $v$. 
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 $v$?

![Diagram of DFS tree over a directed graph]

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 $s^*$

- 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.
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.
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 $s^*$

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

[Diagram of DFS tree with vertical and horizontal 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 $s^*$

- 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.
**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: $\text{visited} \leftarrow$ boolean array over $V$, with $\text{visited}[v] = \text{false}, \forall v$
3: $\text{instack} \leftarrow$ boolean array over $V$, with $\text{instack}[v] = \text{false}, \forall v$
4: $\text{DFS}(s^*)$
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: \hspace{1em} if $\text{inqueue}[u]$ then \hspace{1em} $\triangleright$ Find a vertical edge
4: \hspace{1em} exit the whole algorithm, by returning “there is a cycle”
5: \hspace{1em} else if $\text{visited}[u] = \text{false}$ then
6: \hspace{2em} $\text{DFS}(u)$
7: \hspace{1em} $\text{instack}[v] \leftarrow \text{false}$