Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
Many problems in CS can be modeled as graph problems.

Algorithms for solving graph problems are fundamental to the field of algorithm design.
Graph Algorithms

- Many problems in CS can be modeled as graph problems.
- Algorithms for solving graph problems are fundamental to the field of algorithm design.

Definition

- A graph $G = (V, E)$ consists of a vertex set $V$ and an edge set $E$. $|V| = n$ and $|E| = m$.
- Each edge $e = (u, v) \in E$ is an unordered pair of vertices.
- If $(u, v) \in E$, we say $v$ is a neighbor of $u$.
- The degree $deg(u)$ of a vertex $u$ is the number of edges incident to $u$. 
Fact

\[ \sum_{v \in V} \deg(v) = 2m \]

This is because, for each \( e = (u, v) \), \( e \) is counted twice in the sum, once for \( \deg(v) \) and once for \( \deg(u) \).
Directed Graphs

Definition

- If the two end vertices of \( e \) are ordered, the edge is directed, and we write \( e = \vec{uv} \).
- If all edges are directed, then \( G \) is a directed graph.
- The in-degree \( deg_{in}(u) \) of a vertex \( u \) is the number of edges that are directed into \( u \).
- The out-degree \( deg_{out}(u) \) of a vertex \( u \) is the number of edges that are directed from \( u \).
Directed Graphs

Fact

\[ \sum_{v \in V} \text{deg}_{\text{in}}(v) = \sum_{v \in V} \text{deg}_{\text{out}}(v) = m \]

This is because, for each \( e = \overrightarrow{uv} \), \( e \) is counted once \( (\text{deg}_{\text{in}}(v)) \) in the sum of in-degrees, and once \( (\text{deg}_{\text{out}}(u)) \) in the sum of out-degrees.
The numbers $n (= |V|)$ and $m (= |E|)$ are two important parameters to describe the size of a graph.

It is easy to see $0 \leq m \leq n^2$. 
Graph Algorithms

- The numbers $n (= |V|)$ and $m (= |E|)$ are two important parameters to describe the size of a graph.
- It is easy to see $0 \leq m \leq n^2$.
- If $m$ is close to $n$, we say $G$ is sparse. If $m$ is close to $n^2$, we say $G$ is dense.
The numbers $n (= |V|)$ and $m (= |E|)$ are two important parameters to describe the size of a graph.

It is easy to see $0 \leq m \leq n^2$.

If $m$ is close to $n$, we say $G$ is sparse. If $m$ is close to $n^2$, we say $G$ is dense.

Because $n$ and $m$ are rather independent to each other, we usually use both parameters to describe the runtime of a graph algorithm. Such as $O(n + m)$ or $O(n^{1/2}m)$. 
Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
Graph Representations

We mainly use two graph representations.

**Adjacency Matrix Representation**

We use a 2D array $A[1..n, 1..n]$ to represent $G = (V, E)$:

$$A[i, j] = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{if } (v_i, v_j) \not\in E \end{cases}$$

- Sometimes, there are other information associated with the edges. For example, each edge $e = (v_i, v_j)$ may have a weight $w(e) = w(v_i, v_j)$ (for example, MST). In this case, we set $A[i, j] = w(v_i, v_j)$.
- For undirected graph, $A$ is always symmetric.
- The Adjacency Matrix Representation for directed graph is similar. $A[i, j] = 1$ (or $w(v_i, v_j)$ if $G$ has edge weights) iff $v_i \rightarrow v_j \in E$.
- For directed graphs, $A[*, *]$ is not necessarily symmetric.
Graph Representations

Adjacency List Representation

- For each vertex $v \in V$, there's a linked list $Adj[v]$. Each entry of $Adj[v]$ is a vertex $w$ such that $(v, w) \in E$.

- If there are other information associated with the edges (such as edge weight), they can be stored in the entries of the adjacency list.

- For undirected graphs, each edge $e = (u, v)$ has two entries in this representation, one in $Adj[u]$ and one in $Adj[v]$.

- The Adjacency List Representation for directed graphs is similar. For each edge $e = u \rightarrow v$, there is an entry in $Adj[u]$.

- For directed graphs, each edge has only one entry in the representation.
Comparisons of Representations

Graph algorithms often need the representation to support two operations.

### Neighbor Testing

Given two vertices \( u \) and \( v \), is \((u, v) \in E\)?

### Neighbor Listing

Given a vertex \( u \), list all neighbors of \( u \).

When deciding which representation to use, we need to consider:

- The space needed for the representation.
- How well the representation supports the two basic operations.
- How easy to implement.
Comparisons of Representations

Adjacency List

- **Space:**
  - Each entry in the list needs $O(1)$ space.
  - Each edge has two entries in the representation. So there are totally $2m$ entries in the representation.
  - We also need $O(n)$ space for the headers of the lists.
  - Total Space: $\Theta(m + n)$.

- **Neighbor Testing:** $O(\text{deg}(v))$ time. (We need to go through $\text{Adj}(v)$ to see if another vertex $u$ is in there.)

- **Neighbor Listing:** $O(\text{deg}(v))$. (We need to go through $\text{Adj}(v)$ to list all neighbors of $v$.)

- More complex.
## Comparisons of Representations

### Adjacency Matrix

- **Space:** $\Theta(n^2)$, independent from the number of edges.
- **Neighbor Testing:** $O(1)$ time. *(Just look at $A[i, j]$.)*
- **Neighbor Listing:** $\Theta(n)$. *(We have to look the entire row $i$ in $A$ to list the neighbors of the vertex $i$.)*
- Easy to implement.

- If an algorithm needs **neighbor testing more often than the neighbor listing**, we should use Adj Matrix.
- If an algorithm needs **neighbor testing less often than the neighbor listing**, we should use Adj List.
- If we use Adj Matrix, the algorithm takes at least $\Omega(n^2)$ time since even set up the representation data structure requires this much time.
- If we use Adj List, it is possible the algorithm can run in **linear $\Theta(m + n)$ time.***
Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
Breadth First Search (BFS)

BFS is a simple algorithm that travels the vertices of a given graph in a systematic way. Roughly speaking, it works like this:

- It starts at a given starting vertex \( s \).
- From \( s \), we visits all neighbors of \( s \).
- These neighbors are placed in a queue \( Q \).
- Then the first vertex \( u \) in \( Q \) is considered. All neighbors of \( u \) that have not been visited yet are visited, and are placed in \( Q \) ...
- When finished, it builds a spanning tree (called BFS tree).
Breadth First Search (BFS)

BFS is a simple algorithm that travels the vertices of a given graph in a systematic way. Roughly speaking, it works like this:

- It starts at a given starting vertex $s$.
- From $s$, we visit all neighbors of $s$.
- These neighbors are placed in a queue $Q$.
- Then the first vertex $u$ in $Q$ is considered. All neighbors of $u$ that have not been visited yet are visited, and are placed in $Q$ ...
- When finished, it builds a spanning tree (called BFS tree).

Before describing details, we need to pick a graph representation. Because we need to visit all neighbors of a vertex, it seems we need the neighbor listing operation. So we use Adj list representation.
BFS

**Input:** An undirected graph $G = (V, E)$ given by Adj List. $s$: the starting vertex.
**Input:** An undirected graph $G = (V, E)$ given by Adj List. $s$: the starting vertex.

**Basic Data Structures:** For each vertex $u \in V$, we have

- $\text{Adj}[u]$: the Adj list for $u$. 
- $\text{color}[u]$: It can be one of the following; white (u has not been visited yet.), grey (u has been visited, but some neighbors of u have not been visited yet.), black (u and all neighbors of u have been visited.)
- $\pi[u]$: the parent of $u$ in the BFS tree.
- $d[u]$: the distance from $u$ to the starting vertex $s$. 
In addition, we also use a queue $Q$ as mentioned earlier.
**BFS**

**Input:** An undirected graph $G = (V, E)$ given by Adj List. $s$: the starting vertex.

**Basic Data Structures:** For each vertex $u \in V$, we have

- $\text{Adj}[u]$: the Adj list for $u$.  
- $\text{color}[u]$: It can be one of the following: 
  - white, ($u$ has not been visited yet.)
Input: An undirected graph $G = (V, E)$ given by Adj List.
$s$: the starting vertex.

Basic Data Structures: For each vertex $u \in V$, we have
- $\text{Adj}[u]$: the Adj list for $u$.
- $\text{color}[u]$: It can be one of the following:
  - white, ($u$ has not been visited yet.)
  - grey, ($u$ has been visited, but some neighbors of $u$ have not been visited yet.)
**BFS**

**Input:** An undirected graph $G = (V, E)$ given by Adj List.
$s$: the starting vertex.

**Basic Data Structures:** For each vertex $u \in V$, we have
- $\text{Adj}[u]$: the Adj list for $u$.
- $\text{color}[u]$: It can be one of the following:
  - white, ($u$ has not been visited yet.)
  - grey, ($u$ has been visited, but some neighbors of $u$ have not been visited yet.)
  - black, ($u$ and all neighbors of $u$ have been visited.)
**Input:** An undirected graph $G = (V, E)$ given by Adj List. $s$: the starting vertex.

**Basic Data Structures:** For each vertex $u \in V$, we have

- $\text{Adj}[u]$: the Adj list for $u$.
- $\text{color}[u]$: It can be one of the following;
  - white, ($u$ has not been visited yet.)
  - grey, ($u$ has been visited, but some neighbors of $u$ have not been visited yet.)
  - black, ($u$ and all neighbors of $u$ have been visited.)
- $\pi[u]$: the parent of $u$ in the BFS tree.
**BFS**

**Input:** An undirected graph $G = (V, E)$ given by Adj List. 
$s$: the starting vertex.

**Basic Data Structures:** For each vertex $u \in V$, we have
- $\text{Adj}[u]$: the Adj list for $u$.
- $\text{color}[u]$: It can be one of the following;
  - white, ($u$ has not been visited yet.)
  - grey, ($u$ has been visited, but some neighbors of $u$ have not been visited yet.)
  - black, ($u$ and all neighbors of $u$ have been visited.)
- $\pi[u]$: the parent of $u$ in the BFS tree.
- $d[u]$: the distance from $u$ to the starting vertex $s$.

In addition, we also use a queue $Q$ as mentioned earlier.
BFS: Algorithm

\textbf{BFS}(G, s)

1. \( Q \leftarrow \emptyset \)
2. \textbf{for each} \( u \in V \setminus \{s\} \) \textbf{do}
3. \hspace{1cm} \( \pi[u] = \text{NIL}; \quad d[u] = \infty; \quad \text{color}[u] = \text{white} \)
4. \( d[s] = 0; \quad \text{color}[s] = \text{grey}; \quad \pi[s] = \text{NIL} \)
5. \textbf{Enqueue}(Q, s)
6. \textbf{while} \( Q \neq \emptyset \) \textbf{do}
7. \hspace{1cm} \( u \leftarrow \text{Dequeue}(Q) \)
8. \hspace{1cm} \textbf{for each} \( v \in \text{Adj}[u] \) \textbf{do}
9. \hspace{2cm} \textbf{if} \( \text{color}[v] = \text{white} \)
10. \hspace{3cm} \textbf{then} \( \text{color}[v] = \text{grey}; \quad d[v] \leftarrow d[u] + 1; \quad \pi[v] \leftarrow u; \quad \text{Enqueue}(Q, v) \)
11. \( \text{color}[u] = \text{black} \)
BFS algorithm takes $\Theta(n + m)$ time.
Theorem

Let $G = (V, E)$ be a graph. Let $d[u]$ be the value computed by BFS algorithm. Then for any $(u, v) \in E$, $|d[u] - d[v]| \leq 1$. 

Proof:

First, we make the following observations:

- Each vertex $v \in V$ is enqueued and dequeued exactly once.
- Initially color $v$ is white. When it is enqueued, color $v$ becomes grey.
- When it is dequeued, color $v$ becomes black. The color remains black until the end.
- The $d[v]$ value is set when $v$ is enqueued. It is never changed again.

At any moment during the execution, the vertices in $Q$ consist of two parts, $Q_1$ followed by $Q_2$ (either of them can be empty).

For all $w \in Q_1$, $d[w] = k$ for some $k$.

For all $x \in Q_2$, $d[x] = k + 1$. 

©Hu Ding (Michigan State University)
BFS: Main Property

Theorem

Let $G = (V, E)$ be a graph. Let $d[u]$ be the value computed by BFS algorithm. Then for any $(u, v) \in E$, $|d[u] - d[v]| \leq 1$.

Proof: First, we make the following observations:

- Each vertex $v \in V$ is enqueued and dequeued exactly once.
- Initially color[$v$] = white. When it is enqueued, color[$v$] becomes grey. When it is dequeued, color[$v$] becomes black. The color remains black until the end.
- The $d[v]$ value is set when $v$ is enqueued. It is never changed again.
- At any moment during the execution, the vertices in $Q$ consist of two parts, $Q_1$ followed by $Q_2$ (either of them can be empty).
  - For all $w \in Q_1$, $d[w] = k$ for some $k$.
  - For all $x \in Q_2$, $d[x] = k + 1$. 
Without loss of generality, suppose that $u$ is visited by the algorithm before $v$. Consider the while loop in BFS algorithm, when $u$ is at the front of $Q$. There are two cases.

**Case 1: color$[v] = \text{white}$ at that moment.**

- Since $v \in \text{Adj}[u]$, the algorithm set $d[v] = d[u] + 1$, and color$[v]$ = grey.

**Case 2: color$[v] = \text{grey}$ at that moment.**

- Then $v$ is in $Q$ at that moment.
- By the previous observation, $d[u] = k$ for some $k$, and $d[v] = k$ or $k + 1$. Thus $d[v] - d[u] \leq 1$. 
BFS: Main Property

Definition

Let $G = (V, E)$ be a graph and $T$ a spanning tree of $G$ rooted at the vertex $s$. Let $x$ and $y$ be two vertices. Let $(u, v)$ be an edge of $G$.

- If $x$ is on the path from $y$ to $s$, we say $x$ is an ancestor of $y$, and $y$ is a descendent of $x$.
- If $(u, v) \in T$, we say $(u, v)$ is a tree edge.
- If $(u, v) \not\in T$ and $u$ is an ancestor of $v$, we say $(u, v)$ is a back edge.
- If neither $u$ is an ancestor of $v$, nor $v$ is an ancestor of $u$, we say $(u, v)$ is a cross edge.

![Diagram showing tree edges, back edges, and cross edges]
BFS: Main Property

**Theorem**

Let $T$ be the BFS tree constructed by the BFS algorithm. Then there are no back edges for $T$. 

**Proof:**
Suppose there is a back edge $(u, v)$ for $T$. Then $|d[u] - d[v]| \geq 2$. This is impossible.
BFS: Main Property

**Theorem**
Let $T$ be the BFS tree constructed by the BFS algorithm. Then there are no back edges for $T$.

**Proof:** Suppose there is a back edge $(u, v)$ for $T$. Then $|d[u] - d[v]| \geq 2$. This is impossible.
BFS: Main Property

**Theorem**
Let $T$ be the BFS tree constructed by the BFS algorithm. Then there are no back edges for $T$.

**Proof:** Suppose there is an back edge $(u, v)$ for $T$. Then $|d[u] - d[v]| \geq 2$. This is impossible.

**Shortest Path Problem**
Let $G = (V, E)$ be a graph and $s$ a vertex of $G$. For each $u \in V$, let $\delta(s, u)$ be the length of the shortest path between $s$ and $u$.

Problem: For all $u \in V$, find $\delta(s, u)$ and the shortest path between $s$ and $u$. 
BFS: Applications

**Theorem**

Let \( d[u] \) be the value computed by BFS algorithm and \( T \) the BFS tree constructed by BFS algorithm. Then for each vertex \( u \in V \),

1. \( d[u] = \delta(s, u) \).
2. The tree path in \( T \) from \( u \) to \( s \) is the shortest path.
Theorem

Let $d[u]$ be the value computed by BFS algorithm and $T$ the BFS tree constructed by BFS algorithm. Then for each vertex $u \in V$,

- $d[u] = \delta(s, u)$.
- The tree path in $T$ from $u$ to $s$ is the shortest path.

Proof: Let $P = \{s = u_0, u_1, \ldots, u_k = u\}$ be the path from $s$ to $u$ in the BFS tree $T$. Then: $d[u] = d[u_k] = k$, $d[u_{k-1}] = k - 1, d[u_{k-2}] = k - 2 \ldots$

Suppose $P' = \{s = v_0, v_1, v_2, \ldots, v_t = u\}$ is the shortest path from $s$ to $u$ in $G$. We need to show $k = t$.

Toward a contradiction, suppose $t < k$. Then there must exist $(v_i, v_{i+1}) \in P'$ such that $|d[v_i] - d[v_{i+1}]| \geq 2$ (by Pigeonhole principle). This is impossible.
BFS algorithm solves the Single Source Shortest Path problem in $\Theta(n + m)$ time.
Connectivity Problem

Definition

- A graph $G = (V, E)$ is **connected** if for any two vertices $u$ and $v$ in $G$, there exists a path in $G$ between $u$ and $v$.
- A **connected component** of $G$ is a maximal subgraph of $G$ that is connected.
- $G$ is connected if and only if it has exactly one connected component.
Connectivity Problem

Definition

- A graph $G = (V, E)$ is **connected** if for any two vertices $u$ and $v$ in $G$, there exists a path in $G$ between $u$ and $v$.

- A **connected component** of $G$ is a maximal subgraph of $G$ that is connected.

- $G$ is connected if and only if it has exactly one connected component.

Connectivity Problem

Given $G = (V, E)$, is $G$ a connected graph? If not, find the connected components of $G$.

We can use BFS algorithm to solve the connectivity problem.
Connectivity Problem

In the **BFS** algorithm, delete the lines 2-3 (initialization of vertex variables).
Connectivity Problem

In the **BFS** algorithm, delete the lines 2-3 (initialization of vertex variables).

**Connectivity** \((G = (V, E))\)

1. **for** each \(i \in V\) **do**
2. \(\text{color}[i] = \text{white}; \ d[i] = \infty; \ \pi[i] = \text{nil};\)
3. \(\text{count} = 0;\)  \quad \text{(count will be the number of connected components)}
4. **for** \(i = 1\) **to** \(n\) **do**
5. \(\text{if} \ \text{color}[i] = \text{white} \ \text{then}\)
6. \(\text{call BFS}(G, i); \ \text{count} = \text{count}+1\)
7. **output** count;
8. **end**
Connectivity Problem

- This algorithm outputs \( \text{count} \), the number of connected components.
- If \( \text{count} = 1 \), \( G \) is connected. The algorithm also constructs a BFS tree.
- If \( \text{count} > 1 \), \( G \) is not connected. The algorithm also constructs a BFS spanning forest \( F \) of \( G \). \( F \) is a collection of trees.
- Each tree corresponds to a connected component of \( G \).
BFS for Directed Graphs

BFS algorithm can be applied to directed graphs without any change.

\[ d[u] \text{ value} \]

\[ \text{edges in BFS} \]

\[ \text{Other edges} \]
BFS for Directed Graphs

BFS algorithm can be applied to directed graphs without any change.

Definition

Let $G = (V, E)$ be a directed graph, $T$ a spanning tree rooted at $s$. An edge $e = u \rightarrow v$ is called:

- **tree edge** if $e = u \rightarrow v \in T$.
- **backward edge** if $u$ is a decedent of $v$.
- **forward edge** if $u$ is an ancestor of $v$.
- **cross edge** if $u$ and $v$ are unrelated.
Theorem

Let $G = (V, E)$ be a directed graph. Let $T$ be the BFS tree constructed by BFS algorithm. Then there are no forward edges with respect to $T$. 

Let $d[u]$ be the value computed by BFS algorithm and $T$ the BFS tree constructed by BFS algorithm. Then for each vertex $u \in V$, the tree path in $T$ from $s$ to $u$ is the shortest path.

\[ d[u] = \text{the length of the shortest path from } s \text{ to } u. \]
## BFS for Directed Graphs: Property

### Theorem
Let $G = (V, E)$ be a directed graph. Let $T$ be the BFS tree constructed by BFS algorithm. Then there are no forward edges with respect to $T$.

### Theorem
Let $d[u]$ be the value computed by BFS algorithm and $T$ the BFS tree constructed by BFS algorithm. Then for each vertex $u \in V$,

- The tree path in $T$ from $s$ to $u$ is the shortest path.
- $d[u] =$ the length of the shortest path from $s$ to $u$. 
Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
Similar to BFS, **Depth First Search (DFS)** is another systematic way for visiting the vertices of a graph.

It can be used on directed or undirected graphs. We discuss DFS for directed graphs first.

DFS has special properties, making it very useful in several applications.

As a high level description, the only difference between BFS and DFS: replace the **queue Q** in BFS algorithm by a **stack S**. So it works like this:
Similar to BFS, Depth First Search (DFS) is another systematic way for visiting the vertices of a graph.

It can be used on directed or undirected graphs. We discuss DFS for directed graph first.

DFS has special properties, making it very useful in several applications.

As a high level description, the only difference between BFS and DFS: replace the queue $Q$ in BFS algorithm by a stack $S$. So it works like this:

**High Level Description of DFS**

- Start at the starting vertex $s$.
- Visit a neighbor $u$ of $s$; visit a neighbor $v$ of $u$ . . .
- Go as far as you can go, until reaching a dead end.
- Backtrack to a vertex that still has unvisited neighbors, and continue
DFS: Recursive algorithm

- It is easier to describe the DFS by using a recursive algorithm.
- DFS also computes two variables for each vertex \( u \in V \):
  - \( d[u] \): The time when \( u \) is "discovered", i.e. pushed on the stack.
  - \( f[u] \): the time when \( u \) is "finished", i.e. popped from the stack.
- These variables will be used in applications.
DFS: Example

[$\{1,12\}$]  [$\{6,9\}$]  [$\{3,10\}$]  [$\{3,10\}$]  [$\{2,11\}$]  [$\{3,10\}$]  [$\{7,8\}$]  [$\{4,5\}$]  [$\{3,10\}$]  [$\{6,9\}$]

$\textbf{d}[u], \textbf{f}[u]$ values

- edges in DFS tree
- edges not in DFS tree

©Hu Ding (Michigan State University) CSE 331 Algorithm and Data Structures
DFS: Recursive algorithm

DFS$(G)$

1. for each vertex $u \in V$ do
2. \hspace{0.5cm} color$[u] \leftarrow$ white; \hspace{0.5cm} $\pi[u] = \text{NIL}$
3. \hspace{0.5cm} time $\leftarrow 0$
4. for each vertex $u \in V$ do
5. \hspace{1cm} if color$[u] = \text{white}$ then DFS-Visit$(u)$
DFS: Recursive algorithm

**DFS**($G$)

1. for each vertex $u \in V$ do
2. \hspace{1em} color[$u$] $\leftarrow$ white; $\pi[u]$ = NIL
3. $time \leftarrow 0$
4. for each vertex $u \in V$ do
5. \hspace{1em} if color[$u$] = white then DFS-Visit($u$)

**DFS-Visit**($u$)

1. color[$u$] $\leftarrow$ grey; $time \leftarrow time + 1$; $d[u] \leftarrow time$
2. for each vertex $v \in Adj[u]$ do
3. \hspace{1em} if color[$v$] = white then $\pi[v] \leftarrow u$; DFS-Visit($v$)
4. color[$u$] $\leftarrow$ black
5. $f[u] \leftarrow time \leftarrow time + 1$
Let $T$ be the DFS tree of $G$ by DFS algorithm. Let $[d[u], f[u]]$ be the time interval computed by DFS algorithm. Let $u \neq v$ be any two vertices of $G$.

- The intervals of $[d[u], f[u]]$ and $[d[v], f[v]]$ are either disjoint or one is contained in another.
- $[d[u], f[u]]$ is contained in $[d[v], f[v]]$ if and only if $u$ is a descendent of $v$ with respect to $T$. 
Classification of Edges

Let $G = (V, E)$ be a directed graph and $T$ a spanning tree of $G$. The edge $e = u \rightarrow v$ of $G$ can be classified as:

- **tree edge** if $e = u \rightarrow v \in T$.
- **back-edge** if $e \notin T$ and $v$ is an ancestor of $u$.
- **forward-edge** if $e \notin T$ and $u$ is an ancestor of $v$.
- **cross-edge** if $e \notin T$, $v$ and $u$ are unrelated with respect to $T$. 
Let $G = (V, E)$ be a directed graph and $T$ the spanning tree of $G$ constructed by DFS algorithm. The classification of the edges can be done as follows.

- When $e = u \rightarrow v$ is first explored by DFS, color $e$ by the color $[v]$.
- If color $[v]$ is white, then $e$ is white and is a tree edge.
- If color $[v]$ is grey, then $e$ is grey and is a back-edge.
- If color $[v]$ is black, then $e$ is black and is either a forward- or a cross-edge.

For DFS tree of directed graphs, all four types of edges are possible.
DFS: Applications

Definition

A directed graph $G = (V, E)$ is called a **directed acyclic graph (DAG for short)** if it contains no directed cycles.
DFS: Applications

Definition

A directed graph \( G = (V, E) \) is called a directed acyclic graph (DAG for short) if it contains no directed cycles.

DAG Testing

Given a directed graph \( G = (V, E) \), test if \( G \) is a DAG or not.
### Definition

A directed graph $G = (V, E)$ is called a directed acyclic graph (DAG for short) if it contains no directed cycles.

### DAG Testing

Given a directed graph $G = (V, E)$, test if $G$ is a DAG or not.

### Theorem

Let $G$ be a directed graph, and $T$ the DFS tree of $G$. Then $G$ is not a DAG $\iff$ there exists back edge.
DFS: Applications

**Proof:** Suppose \( e = u \rightarrow v \) is a back edge. Let \( P \) be the path in \( T \) from \( v \) to \( u \). Then the directed path \( P \) followed by \( e = u \rightarrow v \) is a directed cycle.
DFS: Applications

Proof: Suppose \( e = u \to v \) is a back edge. Let \( P \) be the path in \( T \) from \( v \) to \( u \). Then the directed path \( P \) followed by \( e = u \to v \) is a directed cycle.

\[ \begin{align*}
\text{Proof:} & \quad \leftarrow \quad \text{Suppose} \quad e = u \to v \quad \text{is a back edge. Let} \quad P \quad \text{be the path in} \quad T \quad \text{from} \quad v \quad \text{to} \quad u. \quad \text{Then the directed path} \quad P \quad \text{followed by} \quad e = u \to v \quad \text{is a directed cycle.} \\
\end{align*} \]
**Proof:** Suppose $e = u \rightarrow v$ is a back edge. Let $P$ be the path in $T$ from $v$ to $u$. Then the directed path $P$ followed by $e = u \rightarrow v$ is a directed cycle.

Suppose $C = u_1 \rightarrow u_2 \rightarrow \cdots u_k \rightarrow u_1$ is a directed cycle. Without loss of generality, assume $u_1$ is the first vertex visited by DFS. Then, the algorithm visits $u_2, u_3, \ldots u_k$, before it backtrack to $u_1$. So $u_k \rightarrow u_1$ is a back edge.
DFS: Applications

Proof: Suppose $e = u \rightarrow v$ is a back edge. Let $P$ be the path in $T$ from $v$ to $u$. Then the directed path $P$ followed by $e = u \rightarrow v$ is a directed cycle.

$\Rightarrow$ Suppose $C = u_1 \rightarrow u_2 \rightarrow \cdots u_k \rightarrow u_1$ is a directed cycle. Without loss of generality, assume $u_1$ is the first vertex visited by DFS. Then, the algorithm visits $u_2, u_3, \ldots u_k$, before it backtrack to $u_1$. So $u_k \rightarrow u_1$ is a back edge.

DAG Testing in $\Theta(n + m)$ time

1. Run DFS on $G$. Mark the edges “white”, “grey” or “black”,
2. If there is a grey edge, report “$G$ is not a DAG”. If not “$G$ is a DAG”.
Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
Let $G = (V, E)$ be a DAG. A topological sort of $G$ assigns each vertex $v \in V$ a distinct number $L(v) \in [1..n]$ such that if $u \rightarrow v$ then $L(u) < L(v)$.

Note: If $G$ is not a DAG, topological sort cannot exist.

Application

The directed graph $G = (V, E)$ specifies a job flow chart. Each $v \in V$ is a job. If $u \rightarrow v$, then the job $u$ must be done before the job $v$. A topological sort specifies the order to complete jobs.
Topological Sort

Let $G = (V, E)$ be a DAG. A topological sort of $G$ assigns each vertex $v \in V$ a distinct number $L(v) \in [1..n]$ such that if $u \rightarrow v$ then $L(u) < L(v)$.

Note: If $G$ is not a DAG, topological sort cannot exist.
Topological Sort

Let $G = (V, E)$ be a DAG. A topological sort of $G$ assigns each vertex $v \in V$ a distinct number $L(v) \in [1..n]$ such that if $u \rightarrow v$ then $L(u) < L(v)$.

Note: If $G$ is not a DAG, topological sort cannot exist.

Application

- The directed graph $G = (V, E)$ specifies a job flow chart.
- Each $v \in V$ is a job.
- If $u \rightarrow v$, then the job $u$ must be done before the job $v$.
- A topological sort specifies the order to complete jobs.
We can use DFS to find topological sort.

**Topological-Sort-by-DFS**

1. Run DFS on $G$ (the starting vertex should have 0 in-degree).
2. Number the vertices by decreasing order of $f[v]$ value. (This can be done as follows: During DFS, when a vertex $v$ is finished, insert $v$ in the front of a linked list.)
Topological Sort

We can use DFS to find topological sort.

**Topological-Sort-by-DFS**($G$)

1. Run DFS on $G$ (the starting vertex should have 0 in-degree).
2. Number the vertices by decreasing order of $f[v]$ value. (This can be done as follows: During DFS, when a vertex $v$ is finished, insert $v$ in the front of a linked list.)

Clearly, this algorithm takes $\Theta(m + n)$ time.
Outline

1. Graph Algorithms
2. Graph Representations
3. Breadth First Search (BFS)
4. Depth First Search (DFS)
5. Topological Sort
6. DFS for Undirected Graphs
DFS for Undirected Graphs

- DFS algorithm can be used on an undirected graph $G = (V, E)$ without any change.
- It constructs a DFS tree $T$ of $G$.
- Recall that: for an undirected graph $G = (V, E)$ and a spanning tree $T$ of $G$, the edges of $G$ can be classified as:
  - tree edges
  - back edges
  - cross edges
DFS for Undirected Graphs

- DFS algorithm can be used on an undirected graph $G = (V, E)$ without any change.
- It constructs a DFS tree $T$ of $G$.
- Recall that: for an undirected graph $G = (V, E)$ and a spanning tree $T$ of $G$, the edges of $G$ can be classified as:
  - tree edges
  - back edges
  - cross edges

**Theorem**

Let $G$ be an undirected graph, and $T$ the DFS tree of $G$ constructed by DFS algorithm. Then there are no cross edges.
Summary: Edge Types

For Directed Graphs

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Forward</th>
<th>Backward</th>
<th>Cross</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DFS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
## Summary: Edge Types

### For Directed Graphs

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Forward</th>
<th>Backward</th>
<th>Cross</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>DFS</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### For Undirected Graphs

<table>
<thead>
<tr>
<th></th>
<th>Tree</th>
<th>Back-edge</th>
<th>Cross</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFS</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>DFS</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>