Graphs

EECS3101 E:
Design and Analysis of Algorithms
Fall 2025

CHEN-WEI WANG

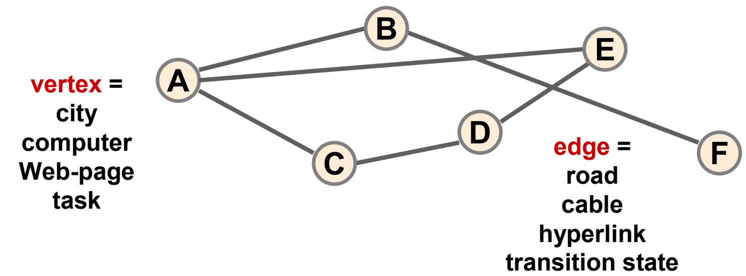
Learning Outcomes of this Lecture

This module is designed to help you understand:

- Vocabulary of the ***Graph*** ADT
- ***Properties*** of Graphs
- ***Algorithms*** on Graphs
  - Traversals: **Depth-**First Search vs. **Breadth-**First Search
  - Topological Sort
  - Minimum Spanning Trees (MST)
  - Dijkstra's Shortest Path Algorithm
- ***Proving*** Properties of Graphs
- ***Implementing*** Graphs in Java

Graphs: Definition

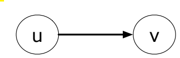
A *graph* $G = (V, E)$ represents ***relations*** that exist between **pairs** of objects.



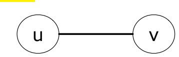
- A set V of ***objects***: ***vertices*** (***nodes***)
- A set E of ***connections*** between objects: ***edges*** (***arcs***)
  - Each ***edge*** (from E) is an **ordered pair** of ***vertices*** (from V).
- e.g., $G = (\{A, B, C, D, E, F\}, \{(A, B), (A, C), (A, E), (C, D), (D, E), (B, F)\})$

Directed vs. Undirected Edges

- An ***edge*** (u, v) connects two ***vertices*** u and v in the graph.
- ***Edge*** (u, v) is *directed* if it indicates the direction of travel.



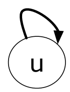
  - Vertex u is the ***origin***.
  - Vertex v is the ***destination***.
  - $(u, v) \neq (v, u)$
- ***Edge*** (u, v) is *undirected* if it does not indicate a direction.



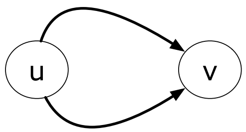
  - $(u, v) = (v, u)$
- 1 undirected edge $(u, v) \equiv$ 2 directed edges (u, v) and (v, u).
- ***Directions*** of ***edges*** represent dependency, order, or flow.

Self vs. Parallel Edges

- An edge (u, u), either directed or undirected, is called a ***self-edge*** (or a ***self-loop***).



- Edges that have the same two end vertices are ***parallel edges*** or ***multiple edges***.



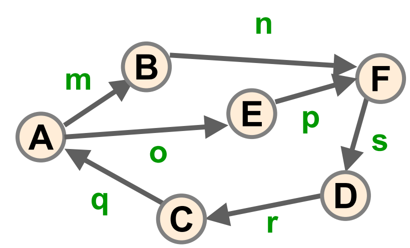
  e.g., In a flight network graph, there are more than one airlines flying between two Seoul and Vancouver.
- A ***simple graph*** has **no** ***self-loops*** and ***parallel edges***.

Vertices

Given an ***edge*** (u, v):

- Vertices u and v are its two ***End vertices*** (***Endpoints***).
- The two end vertices u and v is said to be ***adjacent***.
- Edge (u, v) is ***incident on*** the two end vertices u and v.
- When edge (u, v) is directed:
  - u is **origin** and v is **destination**
  - Edge (u, v) is an ***outgoing edge*** of the origin u
  - Edge (u, v) is an ***incoming edge*** of the destination u
- The ***degree*** of a vertex v is the number of edges ***incident on*** v.

Exercise (1)



- *End vertices* of edge *m*? [*A*, *B*]
- *Outgoing edges* of vertex *A*? [*m*, *o*]
- *Incoming edges* of vertex *A*? [*q*]
- Edges *incident* on vertex *A*? [*m*, *o*, *q*]
- *Degree* of vertex *A*? [3]

Directed vs. Undirected Graphs

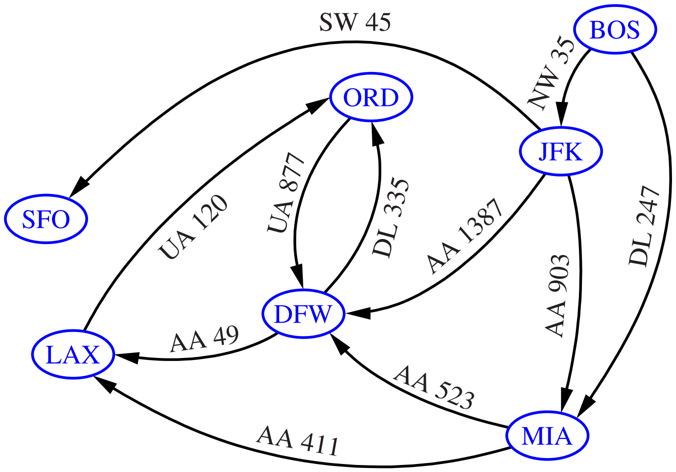
- In a ***directed graph***, **all** edges are directed.

  e.g., dependency graphs (inheritance relationships, method calls, *etc.*)
- In an ***undirected graph***, all edges are undirected.

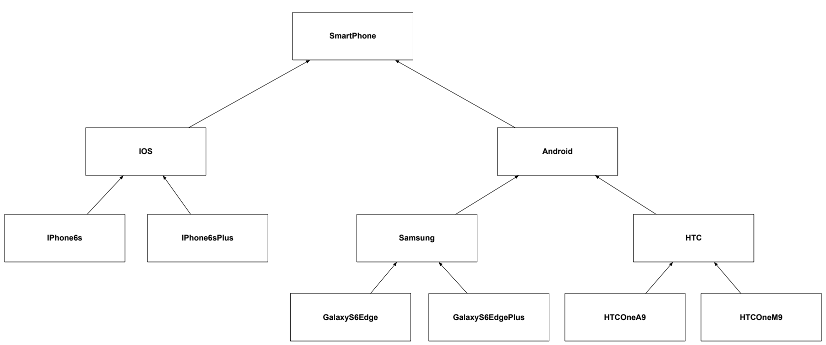
  e.g., Subway map of Young-University Line
- In a ***mixed graph***, **some** edges directed; **some** undirected.

  e.g., A city map has street intersections as vertices and streets as edges: each street may be one-way (a directed edge) or both-way (an undirected edge).

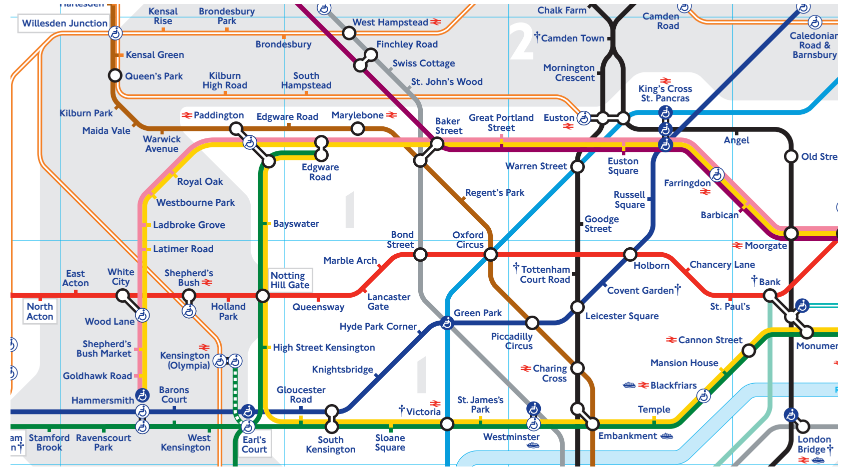
Directed Graph Example (1): A Flight Network



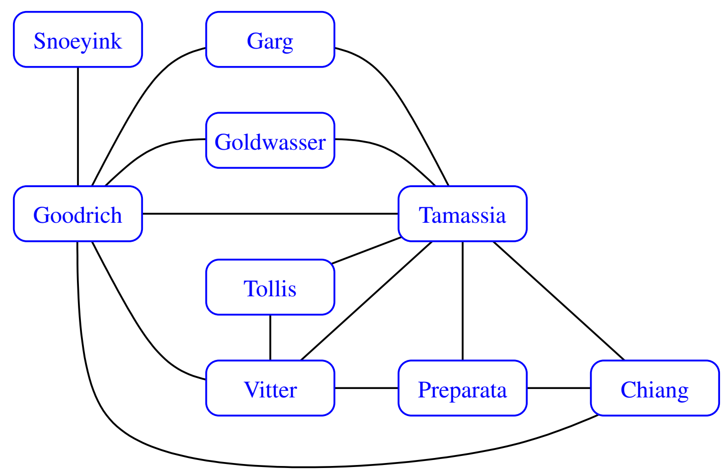
Directed Graph Example (2): Class Inheritance



Undirected Graph Example (1): London Tube



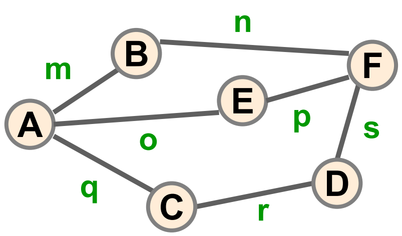
Undirected Graph Example (2): Co-authorship



Basic Properties of Graphs (1)

- Given a **simple**, **undirected** graph $G = (V, E)$ with $|E| = m$:

$$\sum_{v \in V} \texttt{degree}(v) = 2 \cdot m$$

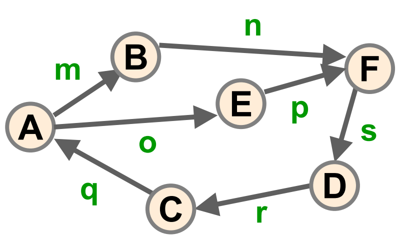


  - ***Intuition***: Each edge (u, v) contributes to degrees of both u and v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

Basic Properties of Graphs (2)

- Given a **simple**, **directed** graph $G = (V, E)$ with $|E| = m$:

$$\sum_{v \in V} \texttt{in-degree}(v) = \sum_{v \in V} \texttt{out-degree}(v)$$

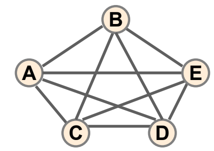


  - ***Intuition***: Each directed edge (u, v) contributes to the `out-degree` of origin u and the `in-degree` of destination v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

Basic Properties of Graphs (3)

- Given a **simple**, **undirected** graph $G = (V, E)$, $|V| = n$, $|E| = m$:

$$m \leq \frac{n \cdot (n-1)}{2}$$



- **Intuition**: Say $V = \{v_1, v_2, \dots, v_n\}$
  - ***Maximum*** value of m is obtained when **each** vertex is connected to **all other** $n-1$ vertices: $n \cdot (n-1)$
  - Since G is **undirected**, for each pair of vertices v_i and v_j, we have **double-counted** (v_i, v_j) and (v_j, v_i): $\frac{n \cdot (n-1)}{2}$
- G is a ***complete graph*** when $m = \frac{n \cdot (n-1)}{2}$

Paths and Cycles (1)

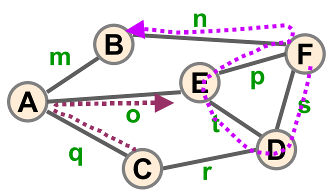
Given a graph $G = (V, E)$:

- A ***path*** of G is a sequence of **alternating** vertices and edges, which **starts** and **ends** at vertices:

$$\langle v_1, e_1, v_2, e_2, \dots, v_{n-1}, e_{n-1}, v_n \rangle \quad v_i \in V, 1 \leq i \leq n, e_j \in E, 1 \leq j < n$$

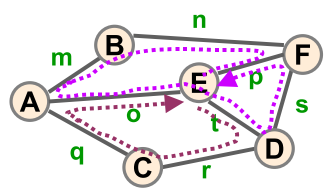
- A ***cycle*** of G is a ***path*** of G with the **same** vertex appearing more than once.
- A ***simple path*** of G is a ***path*** of G with **distinct** vertices.
- A ***simple cycle*** of G is a ***cycle*** of G with **distinct** vertices (except the beginning and end vertices that form the cycle).
- Given two vertices u and v in G, vertex v is ***reachable*** from vertex u if there exists a ***path*** of G such that its ***start vertex*** is u and ***end vertex*** is v.
  - Vertex v may be reachable from vertex u via more than one paths.
  - Any of the ***reachable paths*** from u to v contains a cycle $\Rightarrow$ An **infinite** number of reachable paths from u to v.

Paths and Cycles (2)



Path = (F, s, D, t, E, p, F, n, B)

Simple Path = (C, q, A, o, E)



Cycle = (E, p, F, n, B, m, A, o, E, t, D, s, F, p, E)

Simple Cycle = (E, t, D, r, C, q, A, o, E)

Vertex F is ***reachable*** from vertex A via:

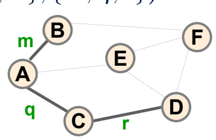
- (A, m, B, n, F)
- (A, o, E, p, F)
- (A, o, E, t, D, s, F)

. . .

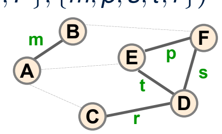
Subgraphs vs. Spanning Subgraphs

Given a graph $G = (V, E)$:

- A ***subgraph*** of G is another graph $G' = (\boldsymbol{V'}, \boldsymbol{E'})$ such that $\boxed{\boldsymbol{V'} \subseteq V}$ and that $\boldsymbol{E'} \subseteq E$.
  e.g., $G_1 = (\{A, B, C, D, E, F\}, \{m, q, r\})$



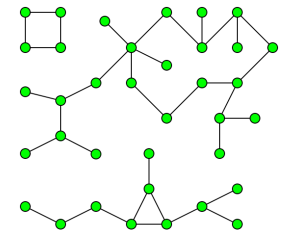
- A ***spanning subgraph*** of G is another graph $G' = (V', E')$ s.t. $\boxed{\boldsymbol{V'} = V}$ and that $\boldsymbol{E'} \subseteq E$.
  e.g., $G_2 = (\{A, B, C, D, E, F\}, \{m, p, s, t, r\})$



Connected Graph vs. Connected Components

Given a graph $G = (V, E)$:

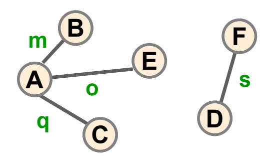
- G is ***connected***: there is a ***path*** between any two vertices of G.
  e.g., Spanning subgraph G_2 extended with the edge n, o, or q
- G's ***connected components***: G's **maximal** ***connected subgraphs***.
  A ***CC*** is **maximal** in that it **cannot** be expanded any further.
  e.g., How many ***connected components*** does the following graph have?



Answer: 3

Forests vs. Trees

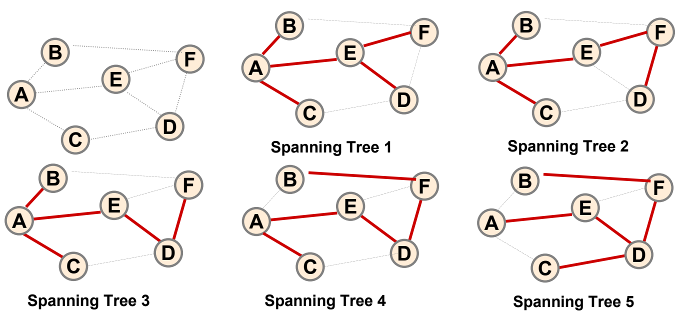
- A ***forest*** is an ***undirected*** graph **without** ***cycles***.



  - $\boxed{\textit{Acyclic}}$:
    Any two ***vertices*** are connected via **at most one** ***path***.
  - A ***forest*** may or may **not** be ***connected***.
    $(\exists v_1, v_2 \bullet \{v_1, v_2\} \subseteq V \land \neg\texttt{connected}(v_1, v_2)) \Rightarrow \neg\texttt{connected}(\textbf{\textit{Forest}}\ G)$
- A ***tree*** is a **connected** ***forest***.
  - $\boxed{\textit{Acyclic} \ \& \ \textit{Connected}}$:
    Any two ***vertices*** are connected via **exactly one** path.
  - e.g., Add either edge (E, F) or (E, D) to the above forest.

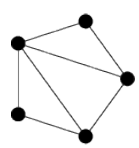
Spanning Trees

- A ***spanning tree*** of graph G: a ***spanning subgraph*** that is also a ***tree***
  - $\Rightarrow$ A ***spanning tree*** of G is a **connected** ***spanning subgraph*** of G that contains **no** ***cycles***.
  - $\Rightarrow \neg\texttt{connected}(G) \Rightarrow \neg(\exists G' \bullet G'$ is a *spanning tree* of G)

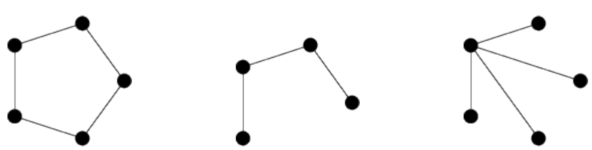


Exercise (2)

Given a graph



Which one of the following is a ***spanning tree***?



(a) (b) (c)

- **(a)**: ***spanning subgraph*** containing a ***cycle*** ($\therefore$ not a ***tree***).
- **(b)**: ***tree*** but **not** ***spanning***.

Basic Properties of Graphs (4)

Given $G = (V, G)$ an **undirected** graph with $|V| = n$, $|E| = m$:

$$\begin{cases} m = n - 1 & \textbf{if } \text{G is a } \textit{spanning tree} \\ m \leq n - 1 & \textbf{if } \text{G is a } \textit{forest} \\ m \geq n - 1 & \textbf{if } \text{G is } \textit{connected} \\ m \geq n & \textbf{if } \text{G contains a } \textit{cycle} \end{cases}$$

- Prove the ***spanning tree*** case via *mathematical induction on n*:
  - **Base Cases**: $n = 1 \Rightarrow m = 0$, $n = 2 \Rightarrow m = 1$, $n = 3 \Rightarrow m = 2$
  - **Inductive Cases**: Assume that a spanning tree has n vertices and $n - 1$ edges.
  - When adding a new vertex v' into the existing graph, we may only expand the **existing** ***spanning tree*** by connecting v' to **exactly one** of the existing vertices; otherwise there will be a ***cycle***.
  - This makes the new spanning tree contains $n + 1$ vertices and n edges.
- When G is a ***forest***, it may be **unconnected** $\Rightarrow m < n - 1$
- When G is **connected**, it may contain ***cycles*** $\Rightarrow m \geq n$

Graph Traversals: Definition

Given a graph G = (**V**, **E**):

- A ***traversal*** of G is a systematic procedure for examining **all** its vertices **V** and edges **E**.
- A ***traversal*** of G is considered ***efficient*** if its ***running time*** is ***linear*** on |**V**| and/or |**E**|. [e.g., $O(|V| + |E|)$]

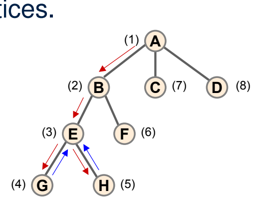
Graph Traversals: Applications

Fundamental questions about graphs involve ***reachability***.
Given a graph G = (**V**, **E**) (directed or undirected):

- Given a vertex **u**, find **all** other vertices in G ***reachable*** from **u**.
- Given a vertex **u** and a vertex **v**:
  - compute a ***path*** from **u** to **v**, or report that there is **no** such a path.
  - compute a ***path*** from **u** to **v** that involves the ***minimum*** number of edges, or report that there is **no** such a path.
- Determine whether or not G is ***connected***.
- Given that G is ***connected***, compute a ***spanning tree*** of G.
- Compute the ***connected components*** of G.
- Identify a ***cycle*** in G, or report that G is ***acyclic***.

Depth-First Search (DFS)

- A ***Depth-First Search*** (***DFS***) of graph G = (V, E), starting from some vertex $v \in V$, proceeds along a **path** from v.
  - The **path** is constructed by following **an** ***incident edge***.
  - The **path** is extended **as far as possible**, until **all** ***incident edges*** lead to vertices that have already been ***visited***.
  - Once the **path** originated from v **cannot be extended further**, ***backtrack*** to the **latest** vertex whose ***incident edges*** lead to some ***unvisited*** vertices.



- DFS resembles the ***preorder traversal*** in trees.
- Use a ***LIFO stack*** to keep track of the nodes to be visited.

DFS: Marking Vertices and Edges

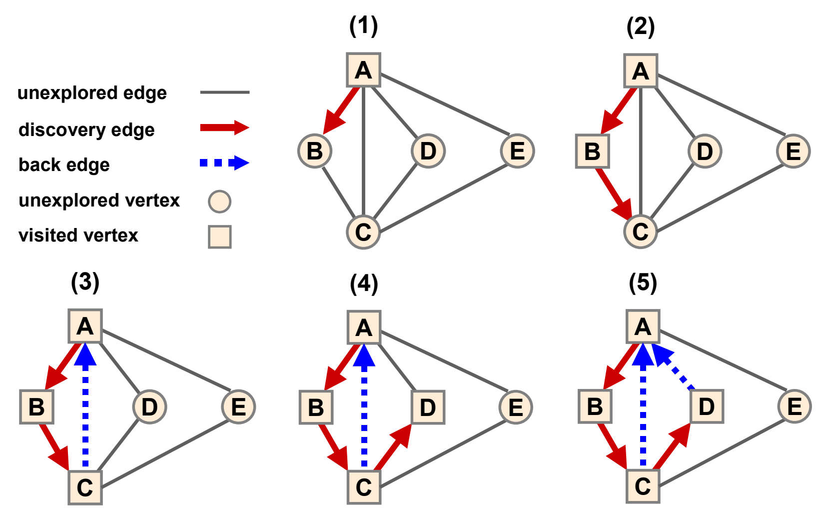
Before the ***DFS*** starts:

- All vertices are ***unvisited***.
- All edges are **unexplored**/**unmarked**.

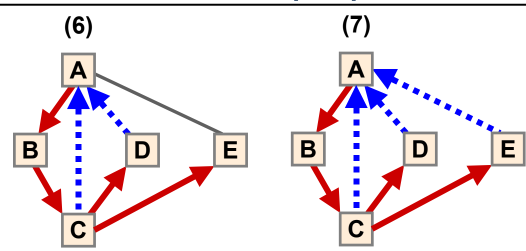
Over the course of a ***DFS***, we **mark** vertices and edges:

- A vertex v is marked ***visited*** when it is **first** encountered.
- Then, we iterate through each of v's **incident edges**, say e:
  - If edge e is already **marked**, then skip it.
  - Otherwise, mark edge e as:
    - A ***discovery*** edge if it leads to an ***unvisited*** vertex
    - A ***back*** edge if it leads to a ***visited*** vertex (i.e., an ancestor vertex)

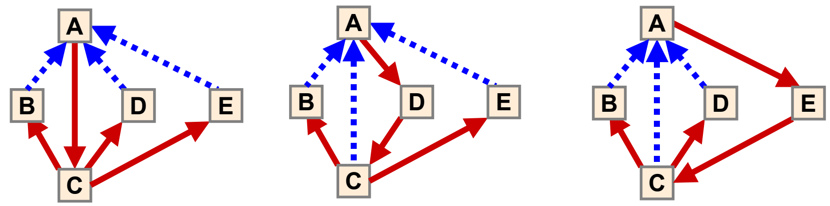
DFS: Illustration (1.1)



DFS: Illustration (1.2)

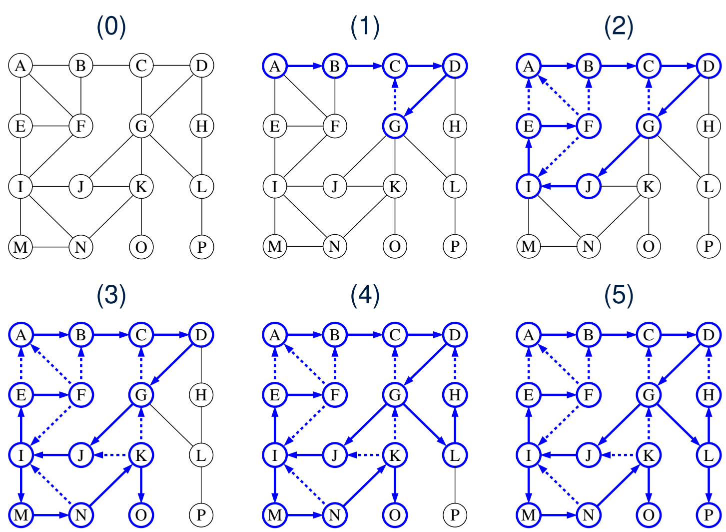


Other solutions (different ***incident edges*** on vertex ***A*** to get started):



edge AC visited first **edge AD visited first** **edge AE visited first**

DFS: Illustration (2)



DFS: Properties

1. Running Time?
   - Every **vertex** is set as ***visited*** **at most once**.
   - Each **edge** is set as either ***DISCOVERY*** or ***BACK*** **at most once**.

   $\Rightarrow$ $O(m+n)$
2. For a ***DFS*** starting from vertex u in a graph G = (V, E):
   2.1 $|visited\ nodes| = |V| \Rightarrow G$ is ***connected***
   2.2 $|visited\ nodes| < |V| \Rightarrow G$ has > 1 ***connected components***
   2.3 There are **no** ***back edges*** $\Rightarrow G$ is ***acyclic***
3. For a ***DFS*** starting from vertex u in an undirected graph G:
   3.1 The traversal visits **all** nodes in the ***connected component*** containing u.
   3.2 ***Discovery edges*** form a ***spanning tree*** (with $|V|-1$ edges) of the ***connected component*** containing u.
4. If a graph G is **not** ***connected***, then it takes **multiple** runs of ***DFS*** to identify **all** G's ***connected components***.

Graph Questions: Adapting DFS

- Given a (directed or undirected) graph $G = (V, E)$:
  - Find a ***path*** between vertex u and vertex v.

    Start a DFS from u and stop as soon as v is encountered.
  - Is vertex v ***reachable*** from vertex u?

    No if a DFS starting from u never encounters v.
  - Find all ***connected components*** of G.
    - Continuously apply ***DFS***'s until the entire set V is visited.
    - Each ***DFS*** produces a **subgraph** representing a new ***CC***.
  - Given that G is ***connected***, find a ***spanning tree*** of it.

    G is **connected**. $\Rightarrow$ G's only ***CC*** is its ***spanning tree***.
- Given an undirected graph $G = (V, E)$:
  - Is G ***connected***?
    - Start a ***DFS*** from an arbitrary vertex, and count # of visited nodes.
    - When the traversal completes, compare the counter value against $|V|$.
  - Is G ***acyclic***?
    - Start a ***DFS*** from an arbitrary vertex.
    - Return **no** (i.e., a ***cycle*** exists) as soon as a ***back edge*** is found.

Graphs in Java: DL Node and List

For each graph, maintain two ***doubly-linked lists*** for ***vertices*** and ***edges***.

```
public class DLNode<E> { /* Doubly-Linked Node */
  private E element;
  private DLNode<E> prev; private DLNode<E> next;
  public DLNode(E e, DLNode<E> p, DLNode<E> n) { ... }
  /* setters and getters for prev and next */
}
```

```
public class DoublyLinkedList<E> {
 private int size;
 private DLNode<E> header; private DLNode<E> trailer;
 public void remove (DLNode<E> node) {
  DLNode<E> pred = node.getPrev();
  DLNode<E> succ = node.getSucc();
  pred.setNext(succ); succ.setPrev(pred);
  node.setNext(null); node.setPrev(null);
  size --;
 }
}
```


Graphs in Java: Interface (1)

```
public interface Graph<V,E> {
 /* Number of vertices of the graph */
 public int numVertices();

 /* Number of edges of the graph */
 public int numEdges();

 /* Vertices of the graph */
 public Iterable<Vertex<V>> vertices();

 /** Edges of the graph */
 public Iterable<Edge<E, V>> edges();

 /* Number of edges leaving vertex v. */
 public int outDegree(Vertex<V> v);

 /* Number of edges for which vertex v is the destination. */
 public int inDegree(Vertex<V> v);

 public int degree(Vertex<V> v);
```


Graphs in Java: Vertex and Edge

```
public class Vertex<V> {
 private V element;
 public Vertex(V element) { this.element = element; }
 /* setter and getter for element */
}
```

```
public class Edge<E, V> {
 private E element;
 private Vertex<V> origin;
 private Vertex<V> dest;
 public Edge(E element) { this.element = element; }
 /* setters and getters for element, origin, and destination */
}
```


Graphs in Java: Interface (2)

```
 /* Edges for which vertex v is the origin. */
 public Iterable<Edge<E, V>> outgoingEdges(Vertex<V> v);

 /* Edges for which vertex v is the destination. */
 public Iterable<Edge<E, V>> incomingEdges(Vertex<V> v);

 /* The edge from u to v, or null if they are not adjacent. */
 public Edge<E, V> getEdge(Vertex<V> u, Vertex<V> v);
```

Graphs in Java: Interface (3)

```
  /* Inserts a new vertex, storing given element. */
  public Vertex<V> insertVertex(V element);

  /* Inserts a new edge between vertices u and v,
   * storing given element.
   */
  public Edge<E, V> insertEdge(Vertex<V> u, Vertex<V> v, E element);

  /* Removes a vertex and all its incident edges from the graph. */
  public void removeVertex(Vertex<V> v);

  /* Removes an edge from the graph. */
  public void removeEdge(Edge<E, V> e);
} /* end Graph */
```


Graphs in Java: Edge List (1)

Each ***vertex*** or ***edge*** stores a ***reference*** to its ***position*** in the respective vertex or edge list.

⇒ ***O(1)*** **deletion** of the vertex or edge from the list.

```
public class EdgeListVertex<V> extends Vertex<V> {
  public DLNode<Vertex<V>> vertextListPosition;
  /* setter and getter for vertexListPosition */
}
```

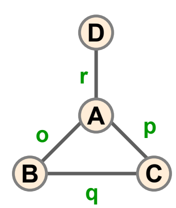
```
public class EdgeListEdge<E, V> extends Edge<E, V> {
  public DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

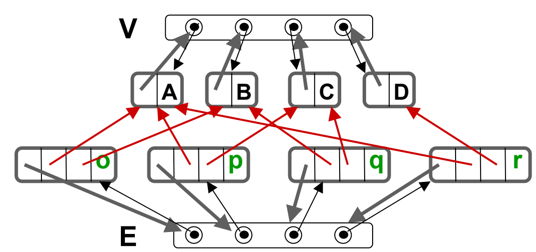

Graphs in Java: Edge List (2)

```
public class EdgeListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<EdgeListVertex<V>> vertices;
  private DoublyLinkedList<EdgeListEdge<E, V>> edges;
  private boolean isDirected;

  /* initialize an empty graph */
  public EdgeListGraph(boolean isDirected) {
    vertices = new DoublyLinkedList<>();
    edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
  ...
}
```


Graphs in Java: Edge List (3)





Index (1)

Index (2)

Index (3)

Index (4)