Graphs



EECS3101 Z:
Design and Analysis of Algorithms
Winter 2026

CHEN-WEI WANG

Learning Outcomes of this Lecture

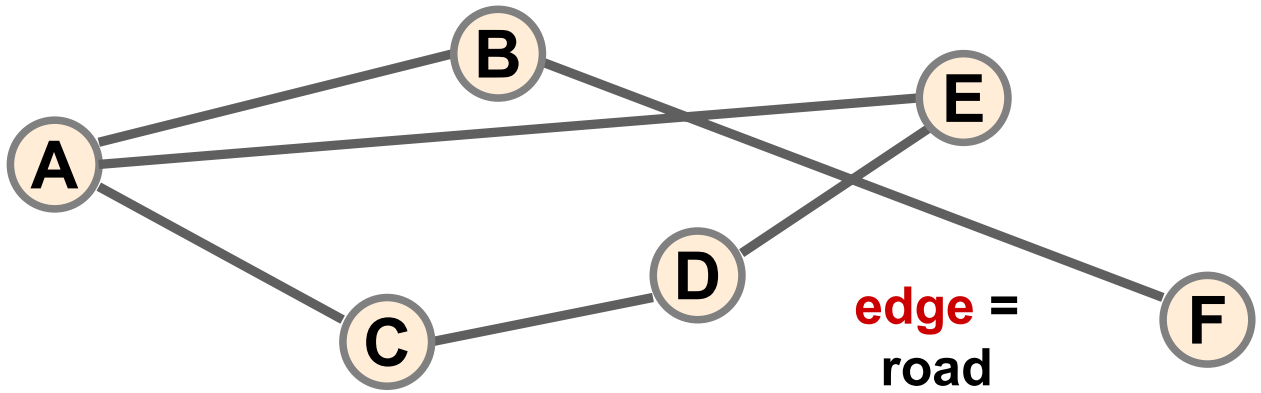
This module is designed to help you understand:

- 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.

vertex =
city
computer
Web-page
task

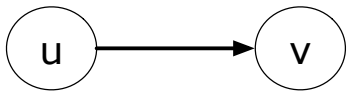


edge =
road
cable
hyperlink
transition state

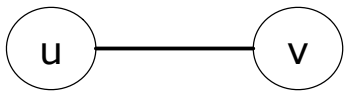
- A set V of ***objects***: ***vertices*** (***nodes***)
- A set E of ***connections*** between objects: ***edges*** (***arcs***)
  - Each ***edge*** (from E) is an **ordered pair** of ***vertices*** (from V).
- e.g., $G = (\{A, B, C, D, E, F\}, \{(A, B), (A, C), (A, E), (C, D), (D, E), (B, F)\})$

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.



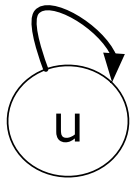
  - Vertex u is the ***origin***.
  - Vertex v is the ***destination***.
  - $(u, v) \neq (v, u)$
- ***Edge*** (u, v) is *undirected* if it does not indicate a direction.



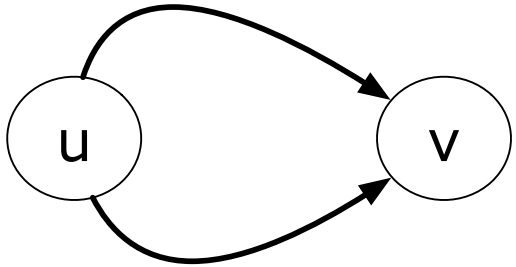
  - $(u, v) = (v, u)$
- 1 undirected edge $(u, v) \equiv$ 2 directed edges (u, v) and (v, u).
- ***Directions*** of ***edges*** represent dependency, order, or flow.

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.

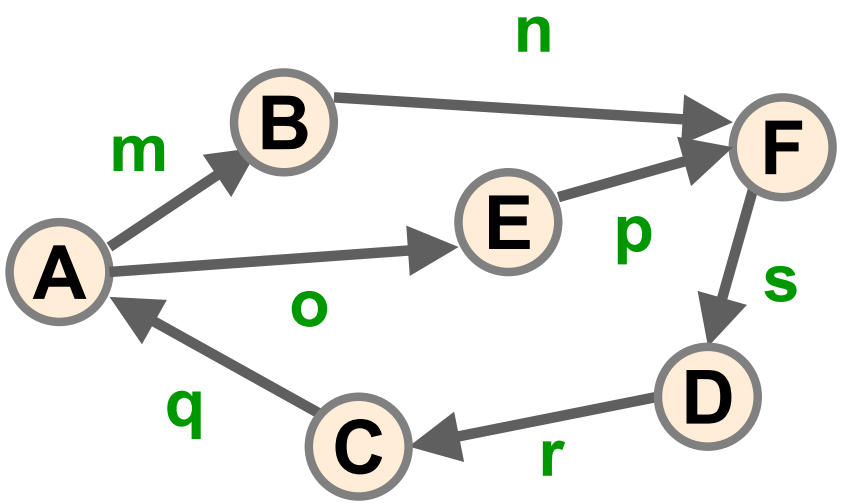
- A ***simple graph*** has **no** ***self-loops*** and ***parallel edges***.

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

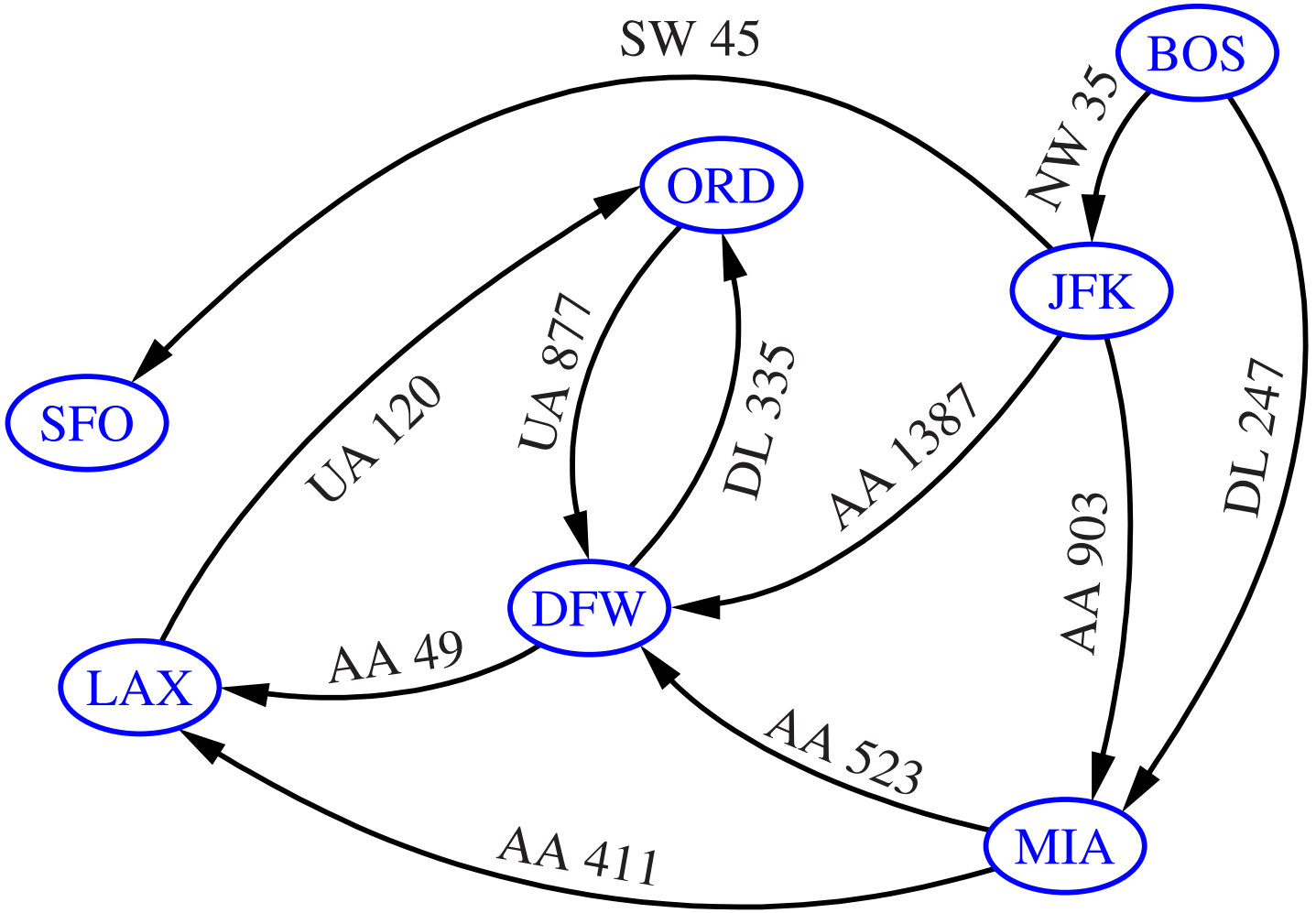
- In a ***directed graph***, **all** edges are directed.

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

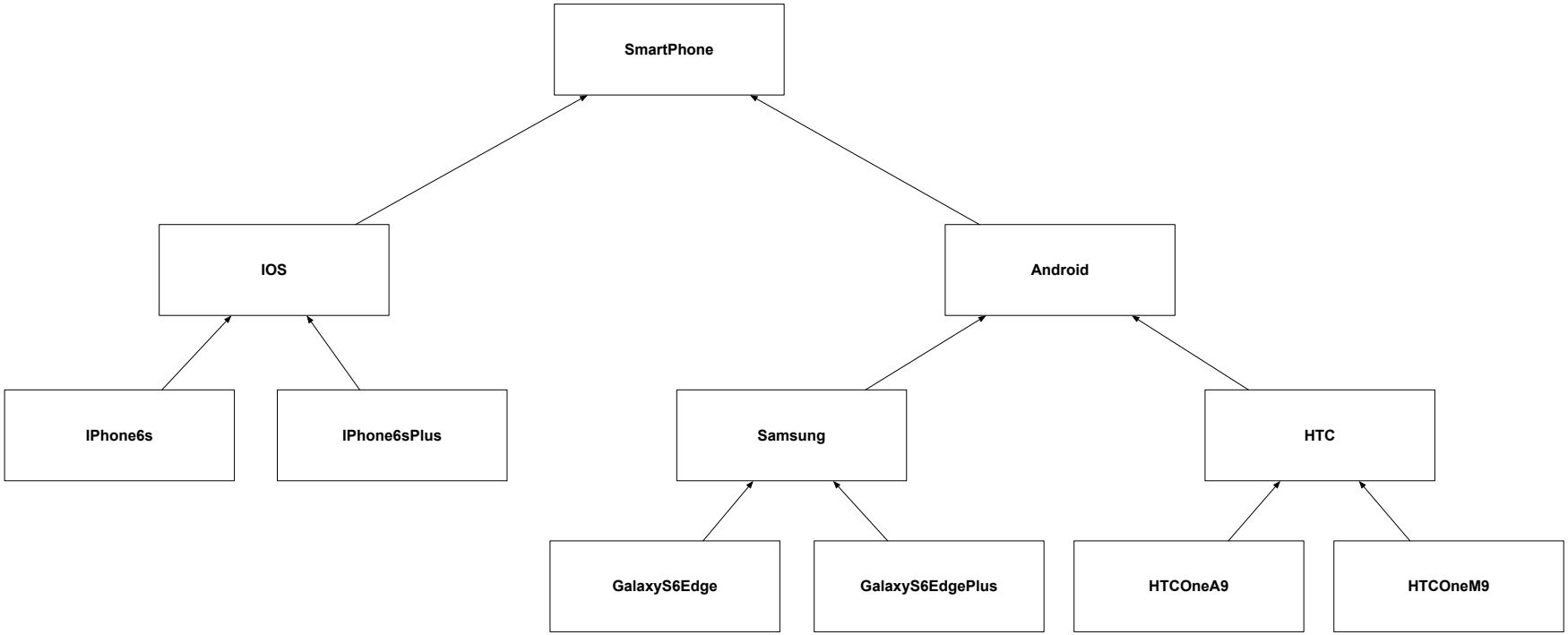
  e.g., Subway map of Young-University Line
- In a ***mixed graph***, **some** edges directed; **some** undirected.

  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).

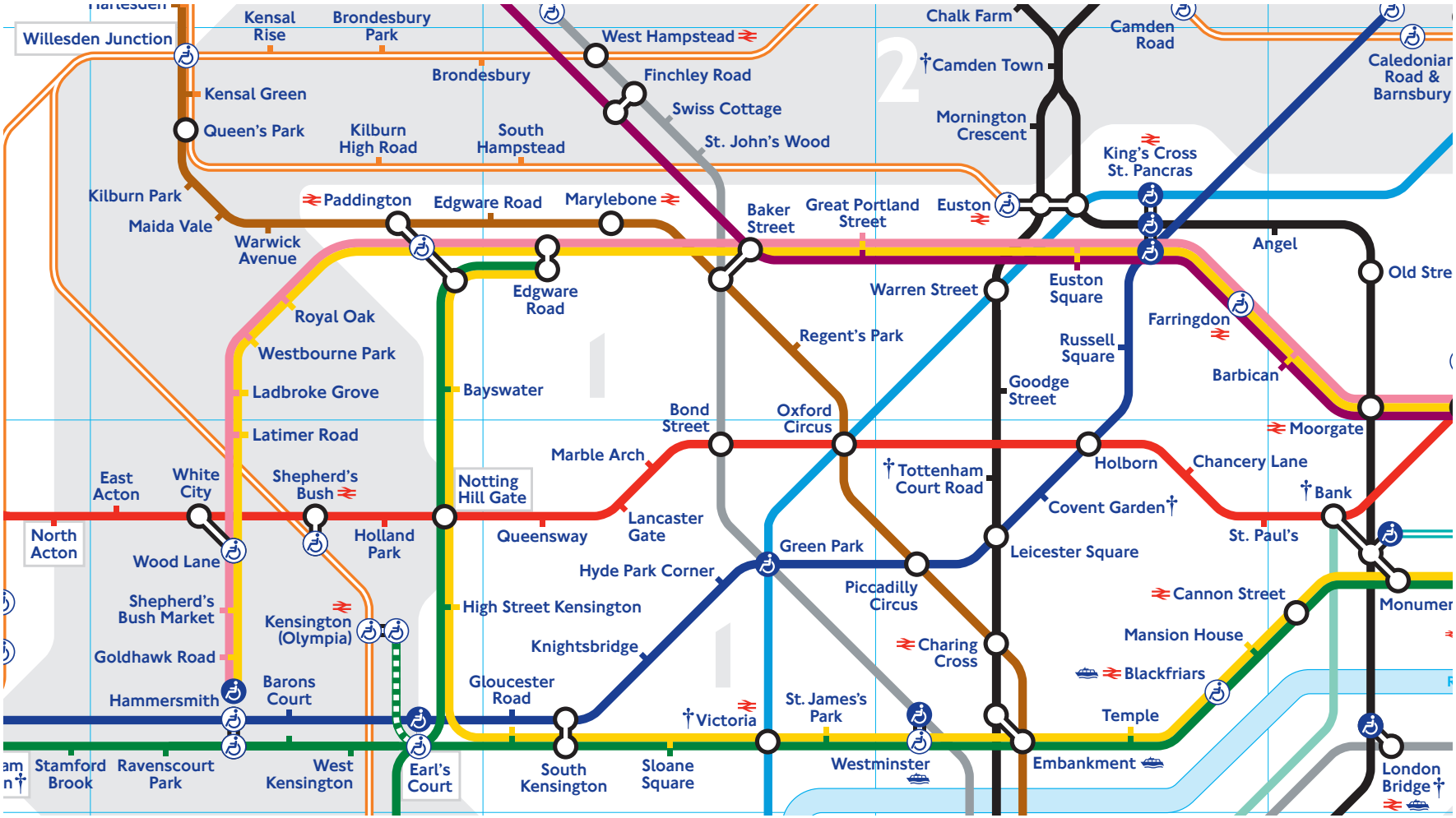
Directed Graph Example (1): A Flight Network



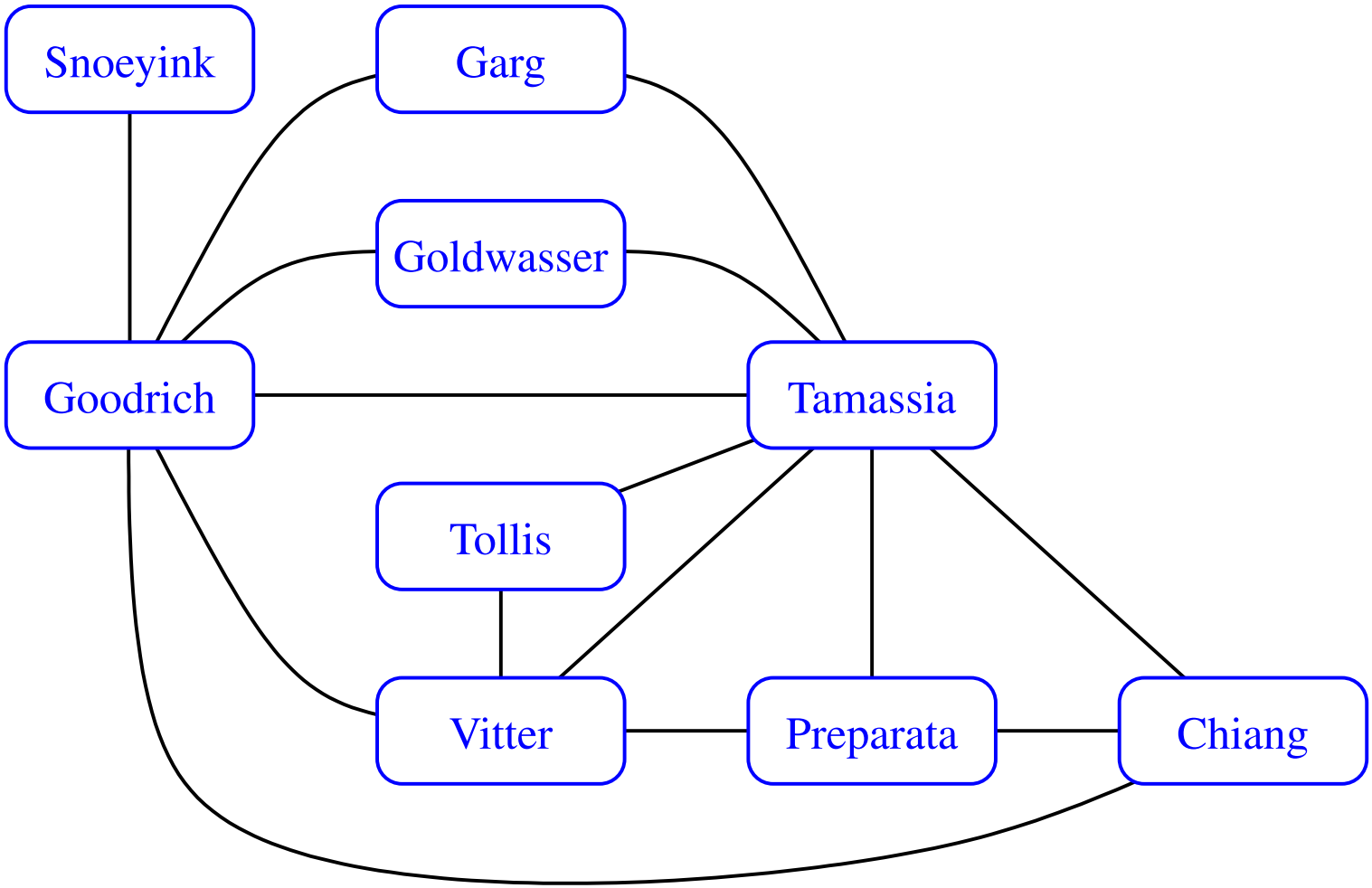
Directed Graph Example (2): Class Inheritance



Undirected Graph Example (1): London Tube



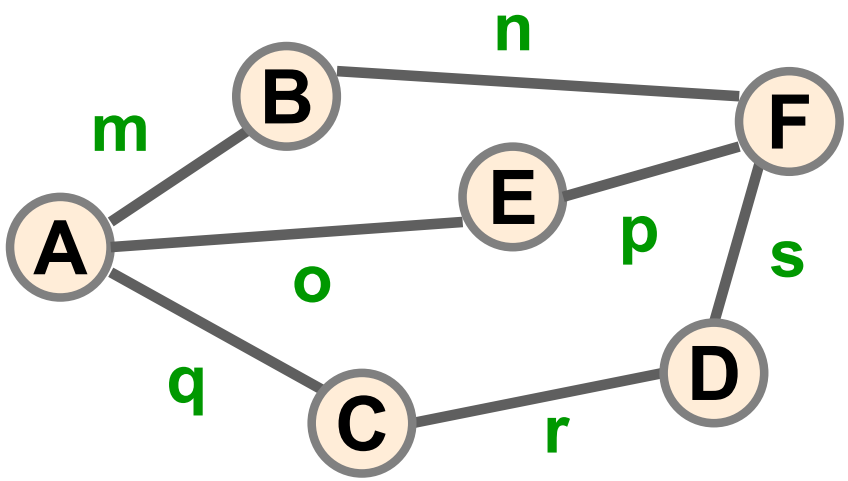
Undirected Graph Example (2): Co-authorship



Basic Properties of Graphs (1)

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

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

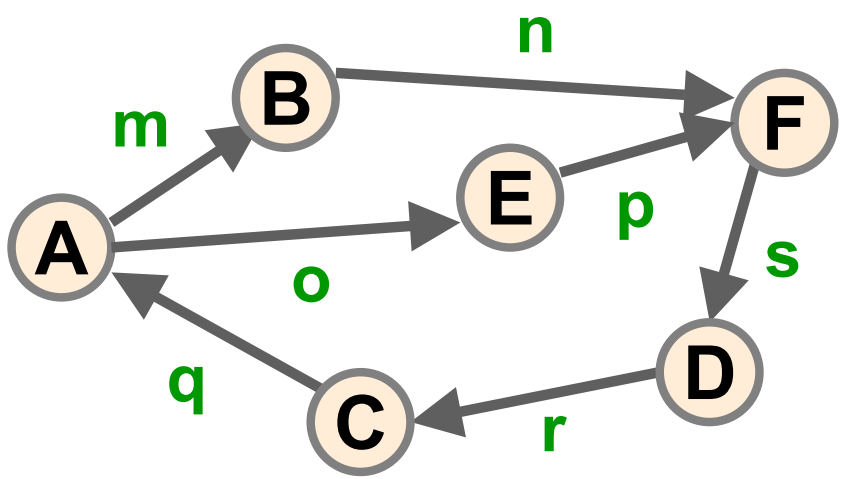


  - ***Intuition***: Each edge (u, v) contributes to degrees of both u and v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

Basic Properties of Graphs (2)

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

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

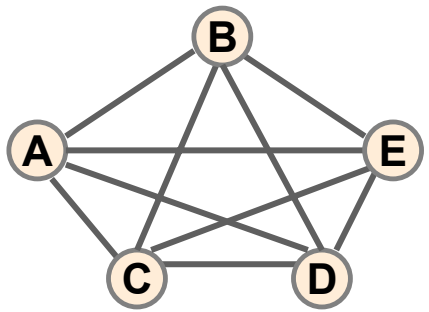


  - ***Intuition***: Each directed edge (u, v) contributes to the `out-degree` of origin u and the `in-degree` of destination v.
  - ***Formal Proof***: ***Mathematical inductoin*** on $|V|$.
- Prove that the claim still holds on graphs that are **not simple**.

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 $V = \{v_1, v_2, \ldots, v_n\}$
  - ***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)

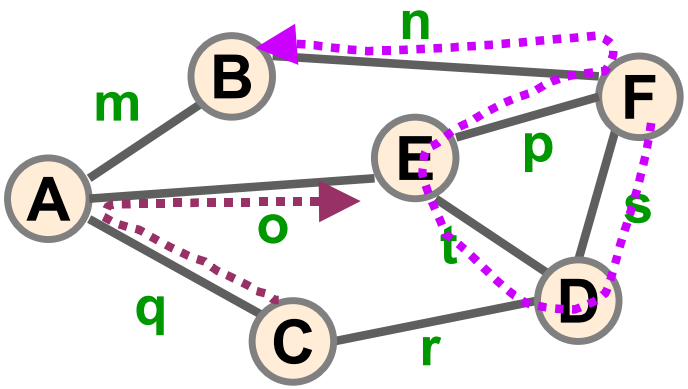
Given a graph $G = (V, E)$:

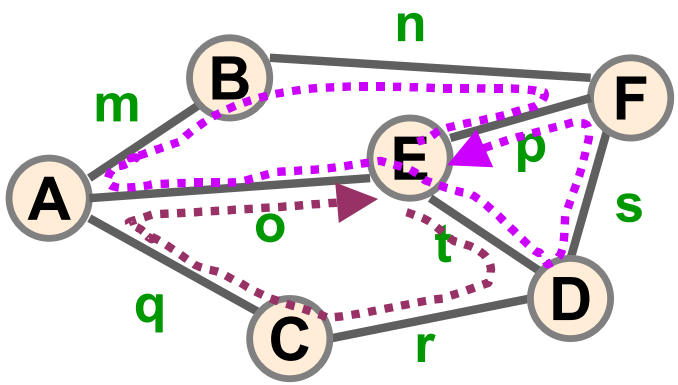
- A ***path*** of G is a sequence of **alternating** vertices and edges, which **starts** and **ends** at vertices:

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

- A ***cycle*** of G is a ***path*** of G with the **same** vertex appearing at its **both** ends (and the rest of it **may** also contain duplicates).
- 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:

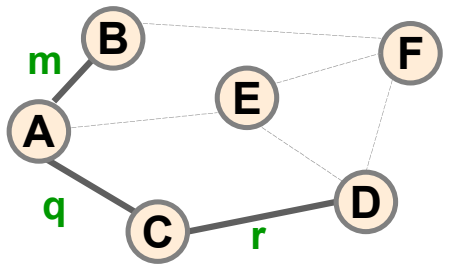
- (A, m, B, n, F)
- (A, o, E, p, F)
- (A, o, E, t, D, s, F)

...

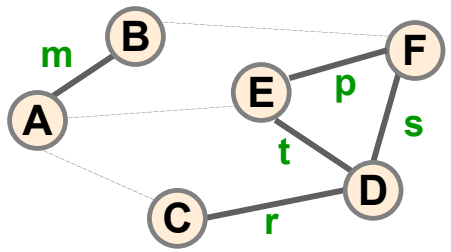
Subgraphs vs. Spanning Subgraphs

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

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



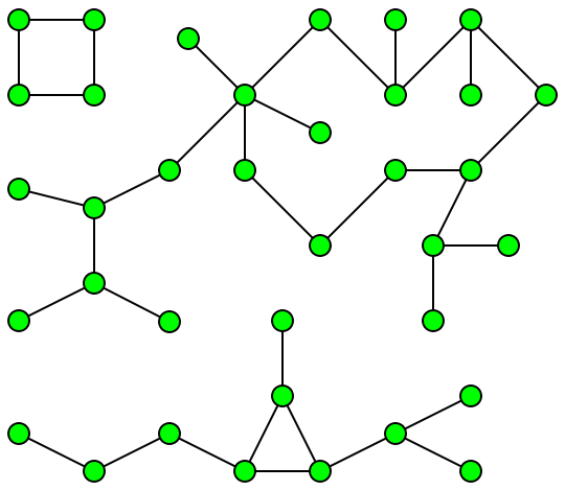
- A ***spanning subgraph*** of G is another graph $G' = (V', E')$ s.t. $V' = V$ and that $E' \subseteq E$.
  e.g., $G_2 = (\{A, B, C, D, E, F\}, \{m, p, s, t, r\})$



Connected Graph vs. Connected Components

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

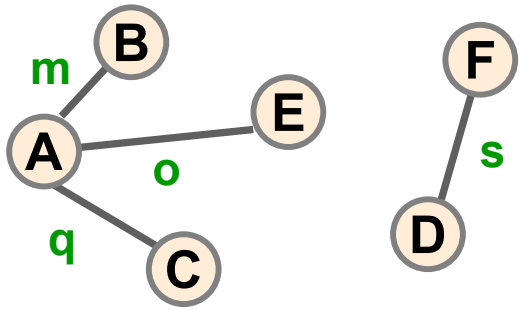
- G is ***connected***: there is a ***path*** between any two vertices of G.
  e.g., Spanning subgraph G_2 extended with the edge n, o, or q
- G's ***connected components***: G's **maximal** ***connected subgraphs***.
  A ***CC*** is **maximal** in that it **cannot** be expanded any further.
  e.g.,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



Answer: 3

Forests vs. Trees

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



  - ***Acyclic***:

    Any two ***vertices*** are connected via **at most one** ***path***.
  - A ***forest*** may or may **not** be ***connected***.

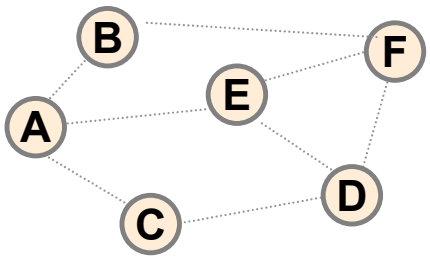
    $(\exists v_1, v_2 \bullet \{v_1, v_2\} \subseteq V \land \neg \texttt{connected}(v_1, v_2)) \Rightarrow \neg \texttt{connected}(\textbf{\textit{Forest}}\ G)$

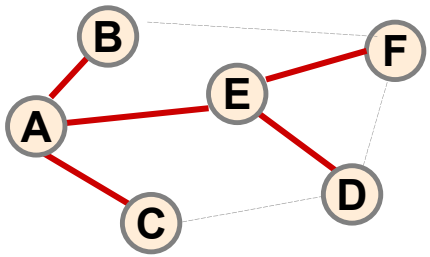
- A ***tree*** is a **connected** ***forest***.
  - ***Acyclic*** **&** ***Connected***:

    Any two ***vertices*** are connected via **exactly one** path.
  - e.g., Add either edge (E, F) or (E, D) to the above forest.

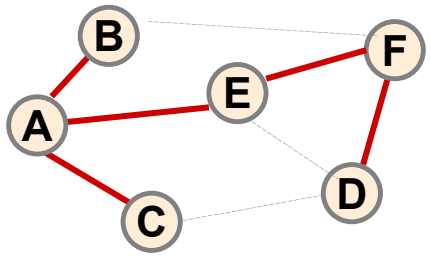
Spanning Trees

- A ***spanning tree*** of graph *G*: a ***spanning subgraph*** that is also a ***tree***
  - ⇒ A ***spanning tree*** of *G* is a **connected** ***spanning subgraph*** of *G* that contains **no** ***cycles***.
  - ⇒ ¬connected(*G*) ⇒ ¬(∃*G′* • G' is a ***spanning tree*** of G)

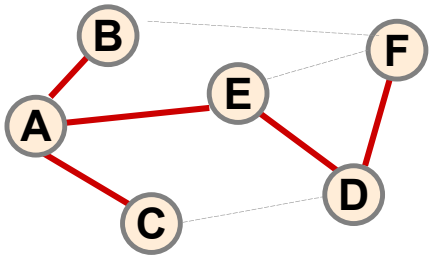




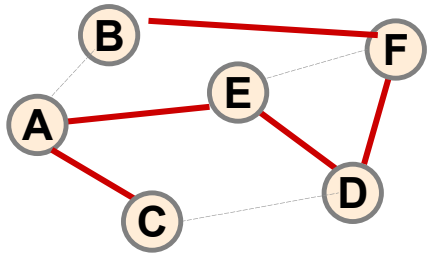
Spanning Tree 1



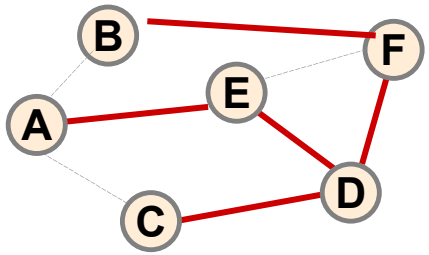
Spanning Tree 2



Spanning Tree 3



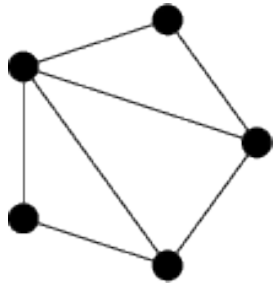
Spanning Tree 4



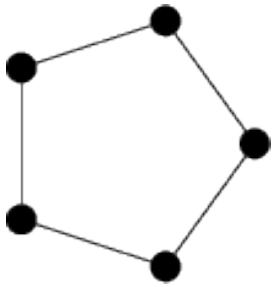
Spanning Tree 5

Exercise (2)

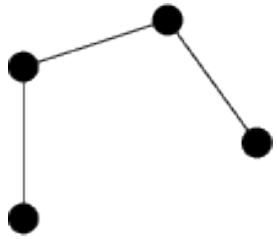
Given a graph



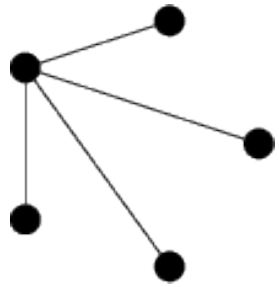
Which one of the following is a ***spanning tree***?



(a)



(b)



(c)

- **(a)**: ***spanning subgraph*** containing a ***cycle*** (∵ not a ***tree***).
- **(b)**: ***tree*** but **not** ***spanning***.

Basic Properties of Graphs (4)

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

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

- 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 a spanning tree with $|V| = n$ and $|E| = n - 1$.
  - 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** $\Rightarrow m < n - 1$
  - **connected & acyclic** $\Rightarrow m = n - 1$
- When G is **connected**, it may be:
  - **acyclic** $\Rightarrow m = n - 1$
  - **cyclic** $\Rightarrow m > n - 1$

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., $O(|V| + |E|)$]

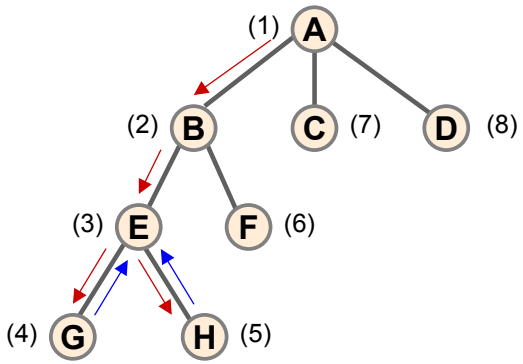
Graph Traversals: Applications

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.
- ***Contrast***: DFS on a **undirected** vs. **directed** graph.

DFS on Undirected Graph: Marking Vertices and Edges

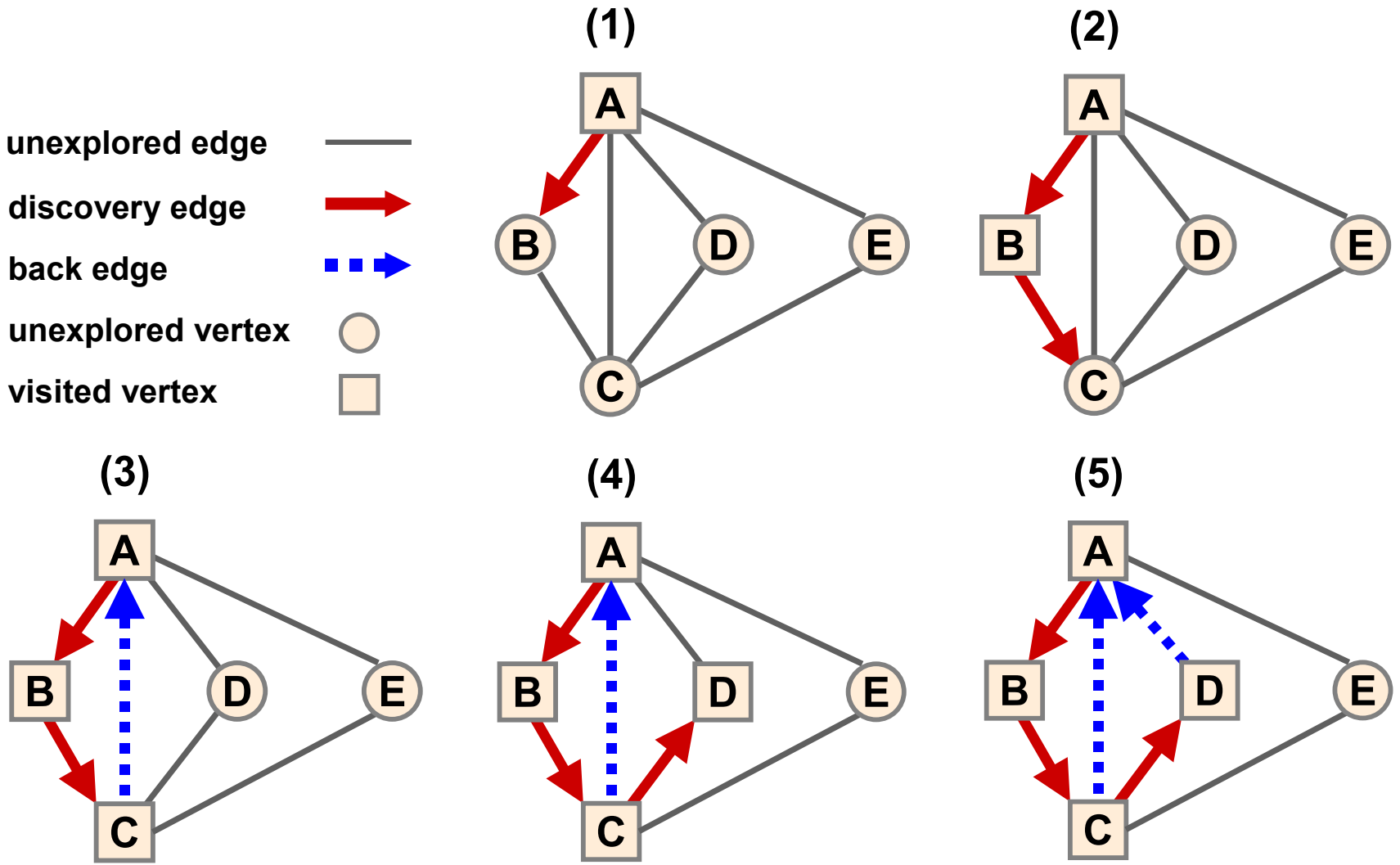
Before the ***DFS*** on a undirected graph starts:

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

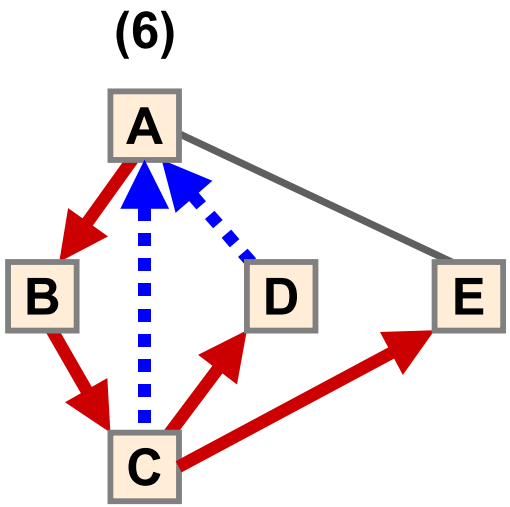
Over the course of a ***DFS***, we **mark** vertices and edges:

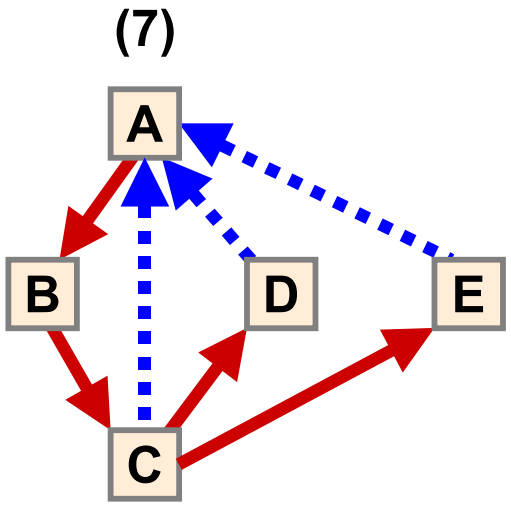
- A vertex *v* is marked ***visited*** when it is **first** encountered.
- Then, we iterate through each of *v*'s **incident edges**, say *e*:
  - If edge *e* is already **marked**, then skip it.
  - Otherwise, mark edge *e* as:
    - A ***discovery*** edge if it leads to an ***unvisited*** vertex
 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

DFS on Undirected Graph: Illustration (1.1)

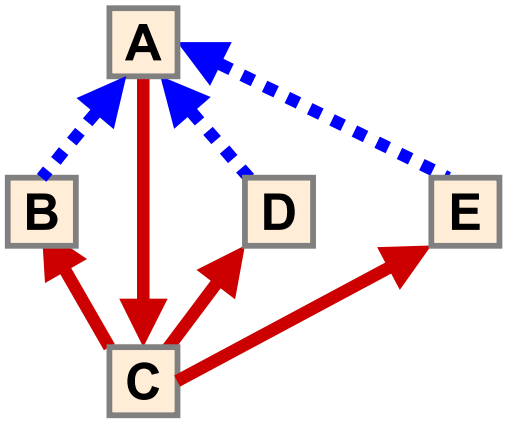


DFS on Undirected Graph: Illustration (1.2)

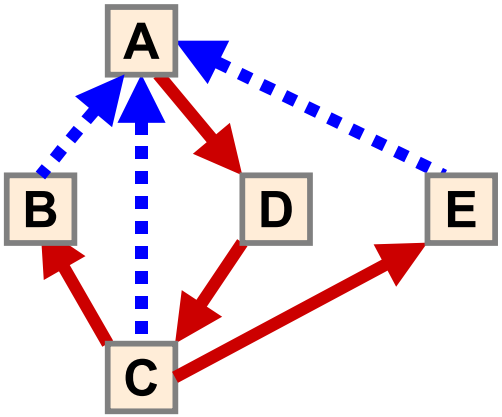




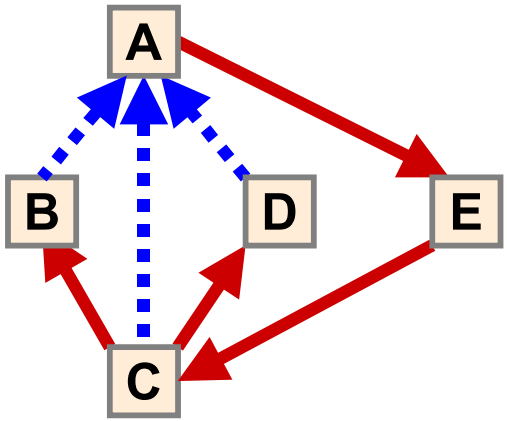
Other solutions (different ***incident edges*** on vertex ***A*** to get started):



edge AC visited first

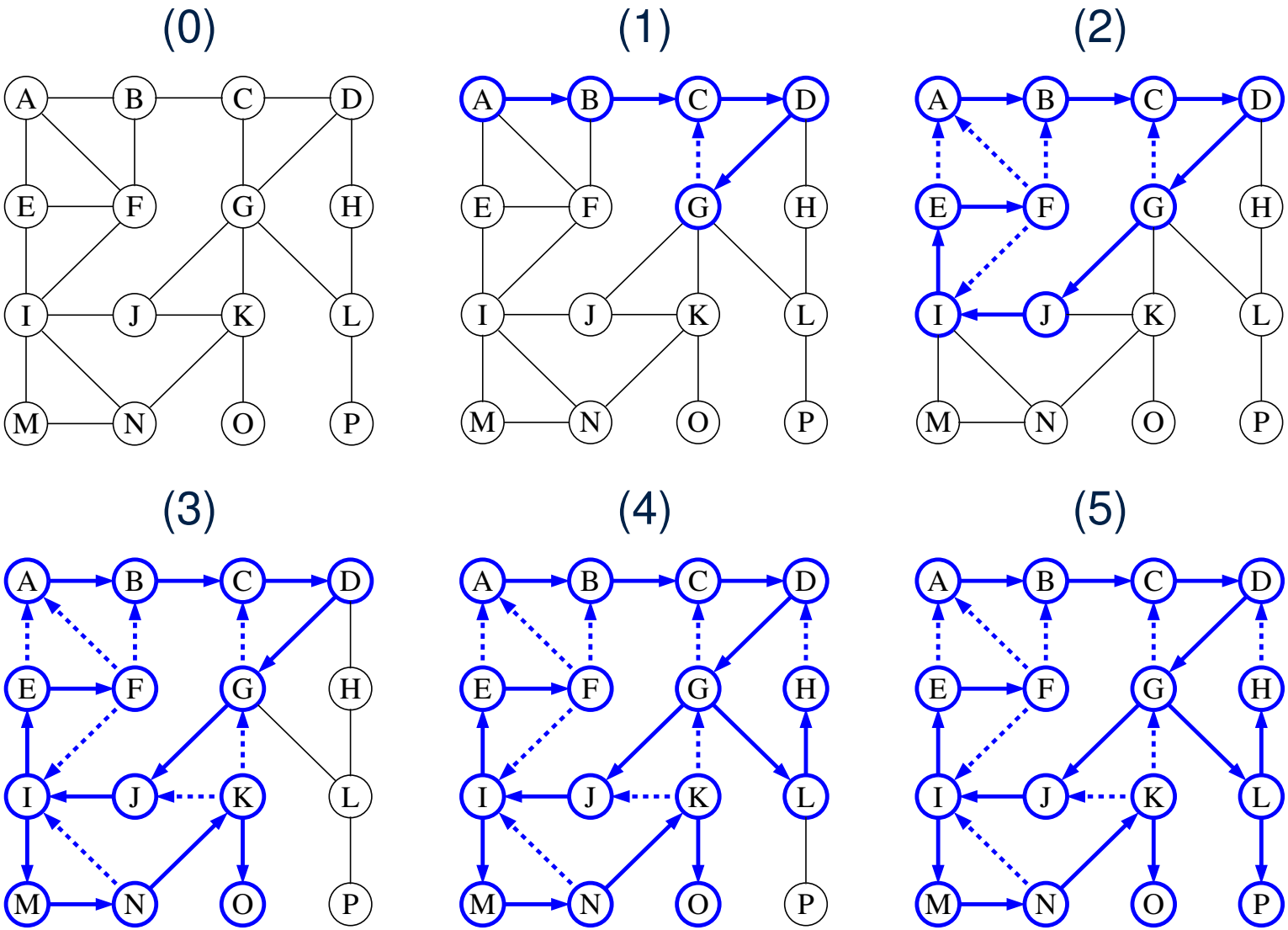


edge AD visited first



edge AE visited first

DFS on Undirected Graph: Illustration (2)



DFS on a Directed Graph: Notations (1)

- A ***DFS run*** refers to a **single** execution/application of ***DFS***.
  - ≥ 1 ***DFS runs*** (starting from different vertices) are needed to visit **all** vertices.
- Let u be **a** vertex **discovered** in the **current** ***DFS run***:
  - $T(u)$: ***DFS tree*** formed by the identified DISCOVERY/TREE edges
  - $r(u)$: the ***start vertex*** of the **current** ***DFS run***, and the ***root*** of $T(u)$
  - v is u's ***ancestor*** if v is on u's **ancestor path**, the **unique** path from u to $r(u)$.
    - v is a ***proper ancestor*** of u if $v \neq u$.
  - w is a ***descendant*** of u if u is w's **ancestor**: w lies in the ***subtree*** rooted at u.
    - w is a ***proper descendant*** of u if $w \neq u$.
- 3 possible ***colours*** for each ***vertex***:
  - WHITE [unvisited/undiscovered]
    - A vertex is white until it is first **pushed** to the ***DFS stack***.
  - GREY [visited/discovered, unfished]
    - A vertex becomes grey when it is first **pushed** to the ***DFS stack***.
    - A vertex remains grey while it is still in the ***DFS stack*** and has **some** of its ***outgoing*** edges **not yet** explored/processed.
  - BLACK [visited/discovered, finished]
    - A vertex becomes black when it is **popped** from the ***DFS stack*** and has **all** its ***outgoing*** edges explored/processed.

DFS on a Directed Graph: Notations (2)

- 2 ***timestamps*** for each ***vertex*** u:
  - DISCOVERY time: $d(u)$ [when u becomes grey]
  - FINISH time: $f(u)$ [when u becomes black]
  - **Requirement**: $d(u) < f(u)$
- 4 possible ***categories*** for each ***edge*** (u, v): [u is grey when examined]
  - DISCOVERY/TREE edge: v is white.
  - BACK edge: v is grey, and v is u's ***proper ancestor*** in $T(u)$.
  - FORWARD edge: v is black, and v is u's ***proper descendant*** in $T(u)$.
  - CROSS edge: v is black, and one of the following holds:
    - $v \in T(u)$: v is neither ancestor nor descendant of u in $T(u)$.
    - $T(v) \neq T(u)$: v is in a **different**, previously **finished** ***DFS tree***.

DFS on a Directed Graph: Exercises

- Can a vertex v's ***colour*** decide whether ***d***(v) or ***f***(v) is assigned yet?
  - WHITE $\Rightarrow$ both $d(v)$ and $f(v)$ undefined
  - GREY $\Rightarrow$ $d(v)$ defined but $f(v)$ undefined
  - BLACK $\Rightarrow$ both $d(v)$ and $f(v)$ defined
- For every vertex v that's **black**, what must be true for ***d***(v) and ***f***(v)?

$$d(v) < f(v)$$

- When ***categorizing*** an edge (u, v), ***colours*** of u and v? Are they related?
  - In all cases, u is GREY.
  - TREE/DISCOVERY edge: v is WHITE
  - BACK edge: v is GREY [$\equiv$ v is an ancestor of u]
  - FORWARD edge: v is BLACK $\wedge$ v is a proper descendant of u
  - CROSS edge: v is BLACK $\wedge$ v is neither an ancestor nor a descendant of u
- Decide if u is an ancestor, a descendant, or neither, of v via ***d*** and ***f***.
  - v is a descendant of $u \iff (d(u) < d(v) \wedge f(v) < f(u))$
  - v is an ancestor of $u \iff (d(v) < d(u) \wedge f(u) < f(v))$
  - v is neither an ancestor nor a descendant of $u \iff (f(u) < d(v) \vee f(v) < d(u))$

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: Vertex and Edge

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

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

Graphs in Java: Interface (1)

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

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

  /* Vertices of the graph */
  public Iterable<EdgeListVertex<V>> getVertices();

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

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

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

  public int getDegreeOf(EdgeListVertex<V> v);
```

Graphs in Java: Interface (2)

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

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

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

Graphs in Java: Interface (3)

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

  /* Inserts a new edge between vertices u and v,
   * storing given element.
   */
  public Edge<E, V> addEdge(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> {
  private DLNode<Vertex<V>> vertexListPosition;
  /* setter and getter for vertexListPosition */
}
```

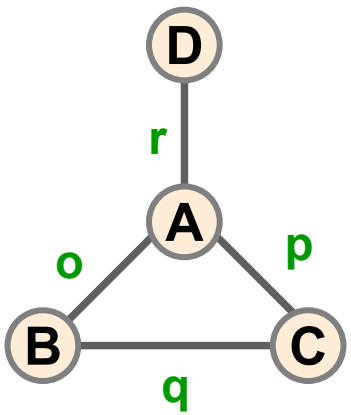
```
public class EdgeListEdge<E, V> extends Edge<E, V> {
  private DLNode<Edge<E, V>> edgeListPosition;
  /* setter and getter for edgeListPosition */
}
```

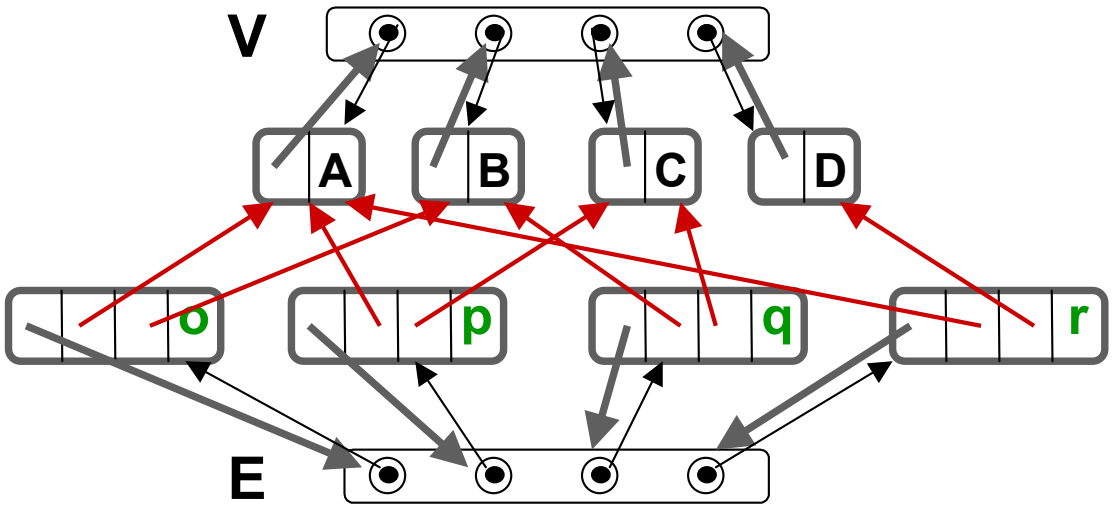
Graphs in Java: Edge List (2)

```
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) {
    this.vertices = new DoublyLinkedList<>();
    this.edges = new DoublyLinkedList<>();
    this.isDirected = isDirected;
  }
  ...
}
```

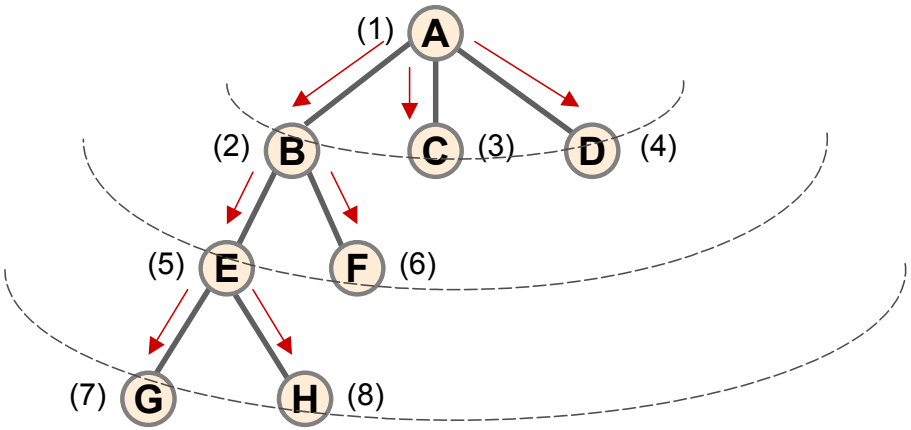
Graphs in Java: Edge List (3)





Breadth-First Search (BFS)

- A ***breadth-first search (BFS)*** of graph G = (V, E), starting from some vertex $v \in V$:
  - Visits every vertex ***adjacent*** to *v* before visiting any other (**more distant**) vertices



    - ***BFS*** attempts to stay as **close** as possible, whereas ***DFS*** attempts to move as **far** as possible
    - ***BFS*** proceeds in rounds and divides the vertices into ***levels***
  - **No backtracking** in ***BFS***: it is completed **as soon as** the **most distant level** of vertices from the start vertex *v* are visited.
- Use a ***FIFO queue*** to keep track of the nodes to be visited.

BFS in Java: Marking Vertices and Edges

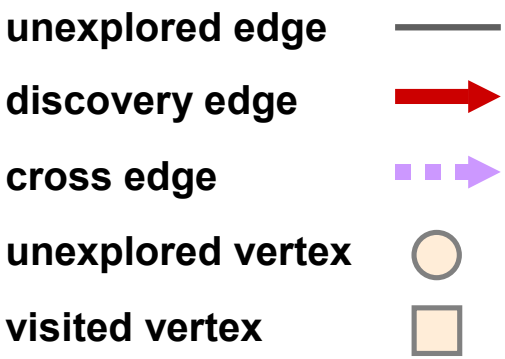
Before the ***BFS*** starts:

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

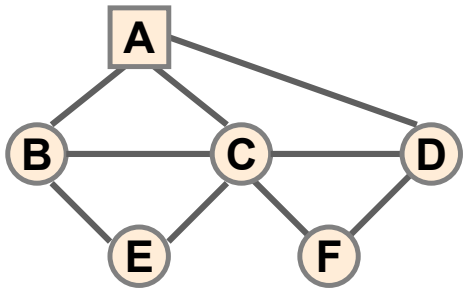
Over the course of a ***BFS***, we **mark** vertices and edges:

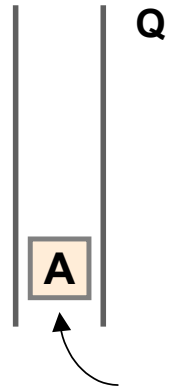
- A vertex is marked ***visited*** when it is **first** encountered.
- Then, we iterate through each of v's **incident edges**, say *e*:
  - If edge *e* is already **marked**, then skip it.
  - Otherwise, for an *undirected* graph, an edge is marked as:
    - A ***discovery*** edge if it leads to an ***unvisited*** vertex
    - A ***cross*** edge if it leads to a ***visited*** vertex
      (i.e., from a different ***branch*** at the same ***level***).
- A ***cross*** edge:
  - Always connects to vertices *at the same level*
  - Can not connect to vertices at an upper or a lower level
    ∵ It would've been or will be marked as a ***discovery edge***.

Iterative BFS: Illustration (1.1)

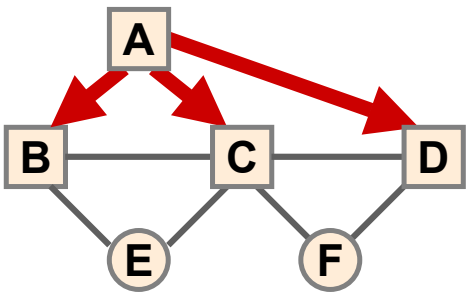


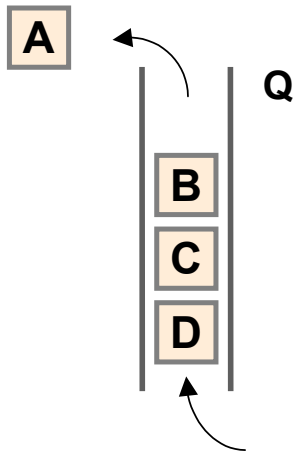
(0)



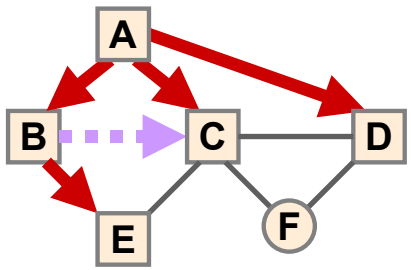


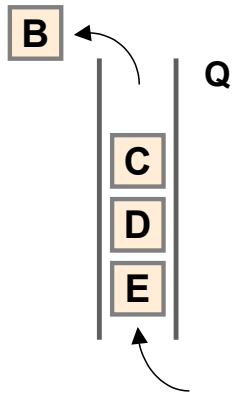
(1)



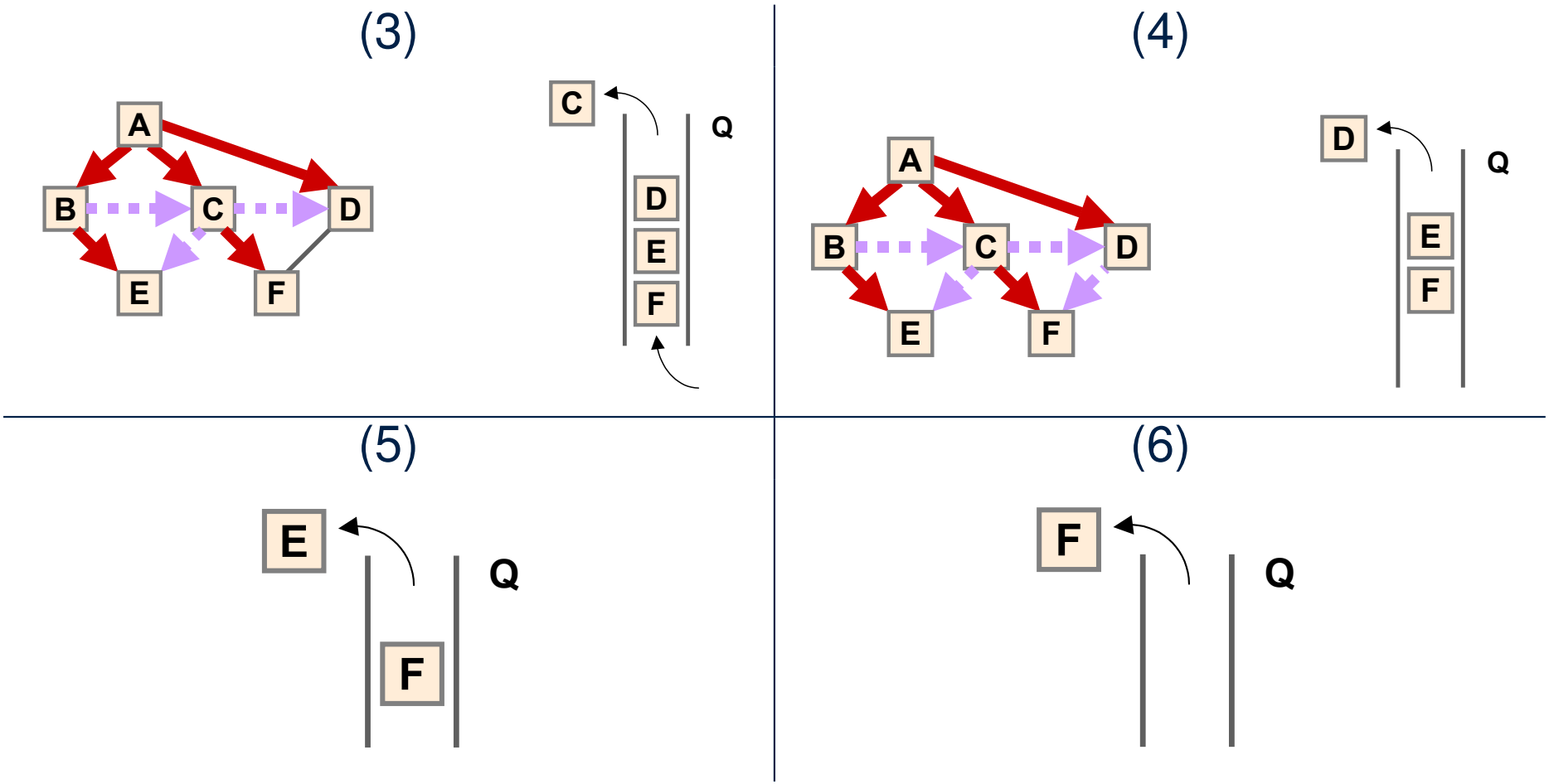


(2)

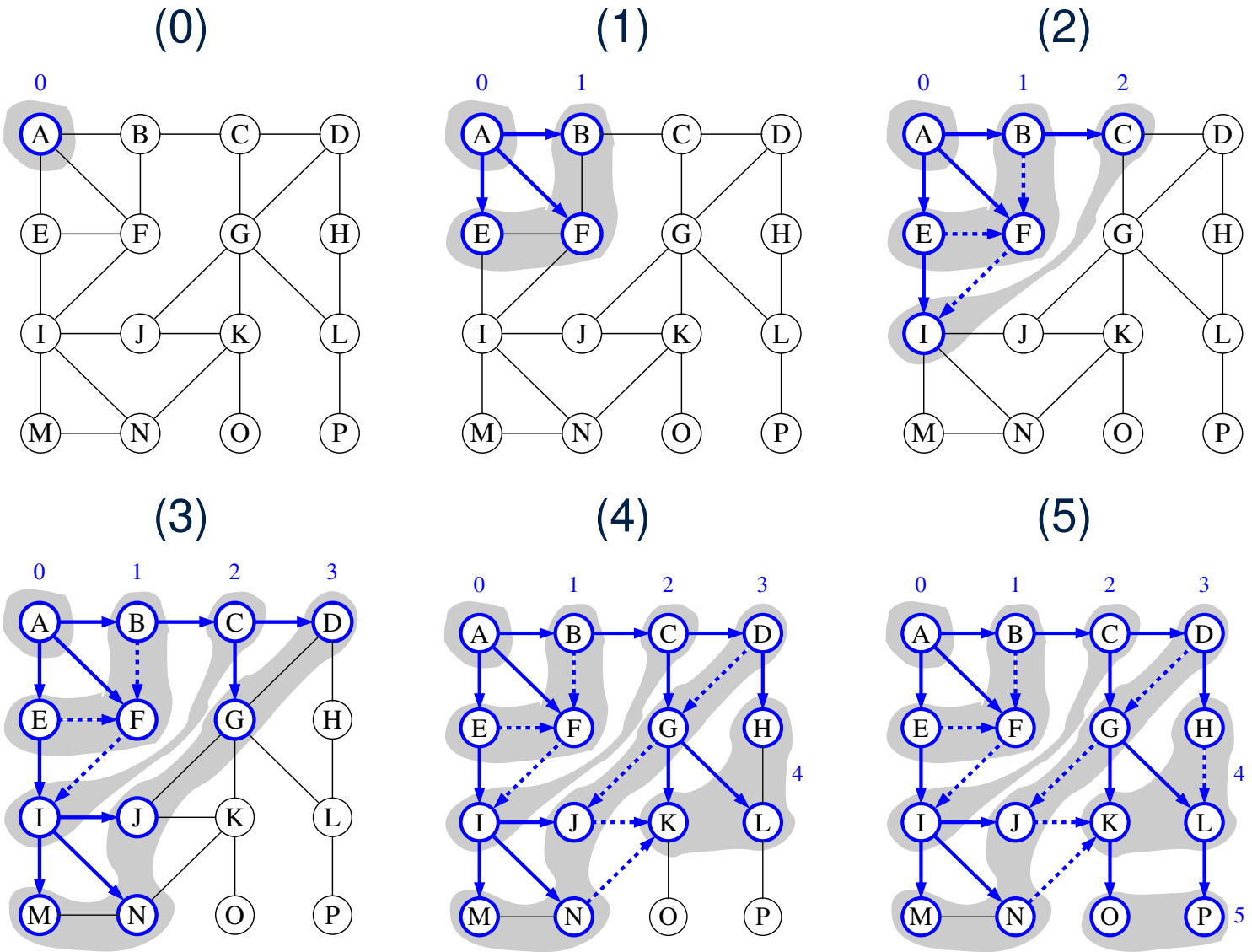




Iterative BFS: Illustration (1.2)



BFS: Illustration (2)



BFS: Properties

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

   $\Rightarrow$ $O(m+n)$
2. For a ***BFS*** 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** A ***cross edge*** connects vertices that in different ***branches***. In a directed graph, this does **not** necessarily creates a **cycle**.
3. For a ***BFS*** 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*** or ***level 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 ***BFS*** to identify **all** G's ***connected components***.

Graph Questions: Adapting BFS

- Given a (directed or undirected) graph $G = (V, E)$:
  - Find a ***shortest path*** (by **edges**) between vertex u and vertex v.
    Start a BFS from u and stop as soon as v is encountered.
  - Is vertex v ***reachable*** from vertex u?
    No if a BFS starting from u never encounters v.
  - Find all ***connected components*** of G.
    - Continuously apply ***BFS***'s until the entire set V is visited.
    - Each ***BFS*** 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 ***BFS*** from an arbitrary vertex, and count # of visited nodes.
    - When the traversal completes, compare the counter value against $|V|$.
  - Is an undirected G ***acyclic***?
    - Start a ***BFS*** from an arbitrary vertex.
    - Return **no** (i.e., a ***cycle*** exists) as soon as a ***cross edge*** is found.

Graphs in Java: Adjacency List (1)

- Extends the ***edge list*** structure
- Each ***vertex*** *v* also stores a list of ***incident edges***.
  ⇒ vertex-based methods such as `outgoingEdges` and `removeVertex` takes **O(d_v)** rather than $O(|E|)$
- Each ***edge*** also stores **references** to its ***positions*** in both ***lists of incident edges*** of its two end vertices.
  ⇒ **O(1)** deletion of the ***edge*** from the ***incident edges list***.

```
class AdjacencyListVertex<V> extends EdgeListVertex<V> {
  private DoublyLinkedList<AdjacencyListEdge<E, V>> incidentEdges;
  /* getter for incidentEdges */
}
```

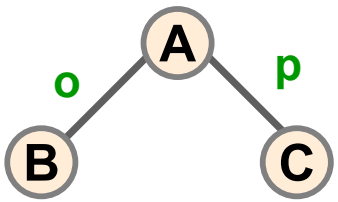
```
class AdjacencyListEdge<V> extends EdgeListEdge<V> {
  DLNode<Edge<E, V>> originIncidentListPos;
  DLNode<Edge<E, V>> destIncidentListPos;
}
```

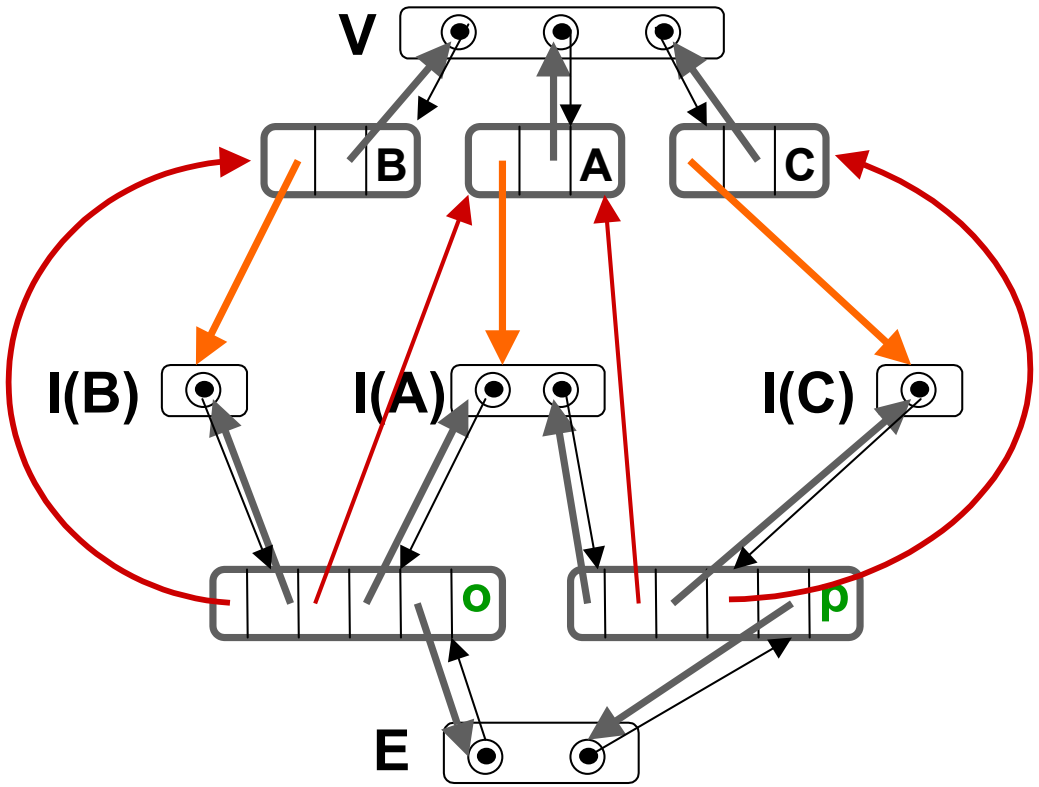
Graphs in Java: Adjacency List (2)

```
class AdjacencyListGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyListVertex<V>> vertices;
  private DoublyLinkedList<AdjacencyListEdge<E, V>> edges;
  private boolean isDirected;

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

Graphs in Java: Adjacency List (3)

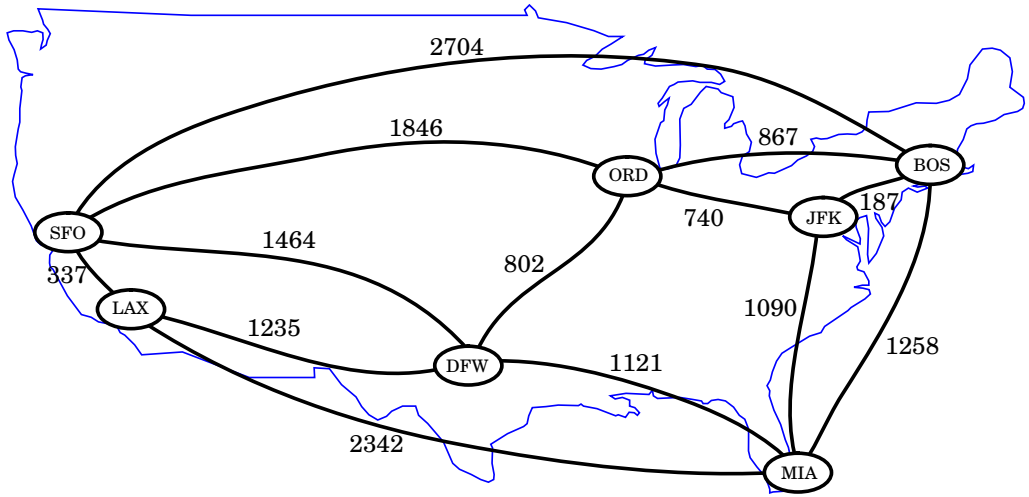




Weighted Graphs

A graph may usefully carry ***weights*** on its ***edges***:

- e.g., Edges of a graph of cities denote distances.



- In a ***weighted graph***, for each *edge* $e = (u, v)$, we write $w(u, v)$ to denote the numerical value of *edge e's weight*.
  e.g., $w(JFK, ORD) = 740$, $w(ORD, DFW) = 802$, $w(DFW, LAX) = 1235$
- ***Weights*** on edges may be considered as "***cost***".
  ⇒ When there are **more than one** ***paths*** existing between two ***vertices***, choose one whose "***total cost***" is the ***minimum***.
- We assume that all edge weights are **non-negative** (i.e., ≥ 0).

Shortest Paths in Weighted Graphs

- Given a ***path P*** $= (v_0, v_1, \ldots, v_k)$, with $k+1$ ***vertices*** and k ***edges***, we define ***w***(***P***) as the ***length*** (or ***weight***) of ***P***:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

e.g., $w(JFK, ORD, DFW, LAX) = 2777$, $w(JFK, MIA, DFW, LAX) = 3446$

- $d(u, v)$ denotes the ***distance*** or ***shortest path*** between ***u*** and ***v***: **minimum** weight sum of a path between ***u*** and ***v***.
  e.g.

$$\begin{aligned} d(JFK, LAX) &= w(JFK, ORD, DFW, LAX) \\ &> w(JFK, MIA, DFW, LAX) \end{aligned}$$

- If there is no path existing between u and v, then $d(u, v) = \infty$

Dijkstra's Shortest Path Algorithm

Starting from a ***source vertex s***, perform a ***BFS***-like procedure:

1. Initially:

1.1 Set $D(s) = 0$, and every other vertex $t \neq s$, $D(t) = \infty$. [***distance***]

1.2 Set $a(v) = nil$ for every vertex v. [***ancestor*** in *shortest path*]

1.3 Insert all vertices into a ***priority queue*** Q [***key***ed by D]

2. While Q is not empty, repeat the following:

2.1 Find vertex u in Q s.t. $D(u)$ is the **minimum**.

2.2 For every vertex v ***adjacent to*** u, **if**:

$v \in Q \wedge \boxed{D(u) + w(u,v) < D(v)}$, then:

- Set $D(v) = D(u) + w(u,v)$
- Set $a(v) = u$

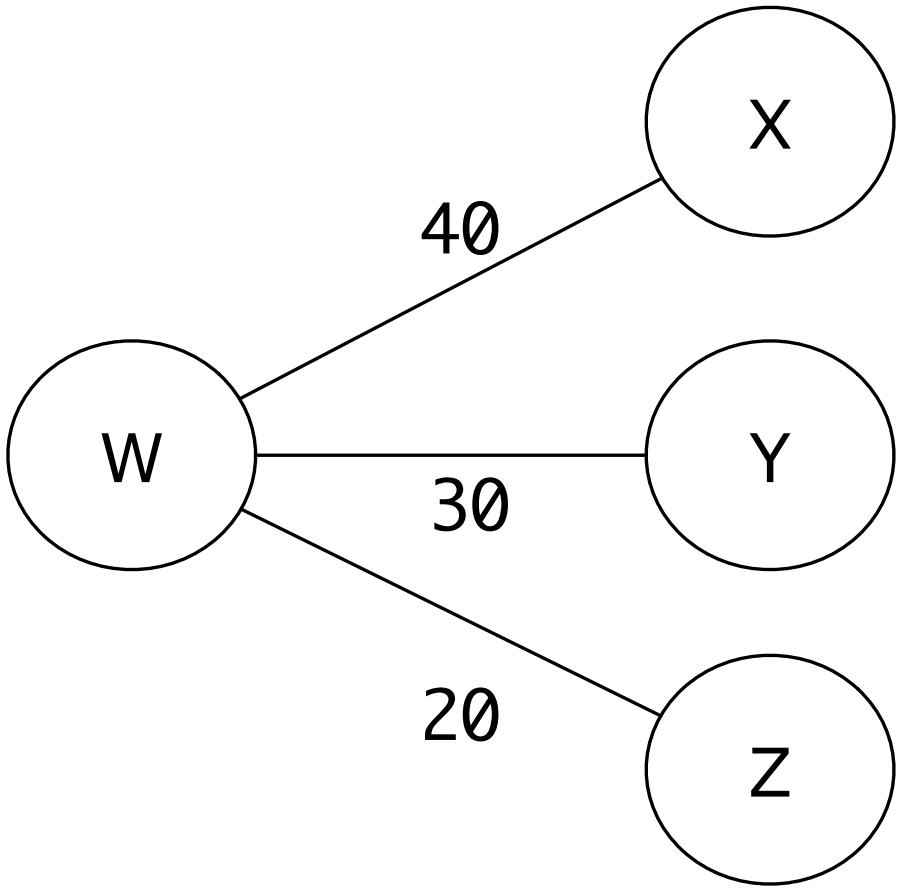
2.3 Remove vertex u from Q.

Upon completion, for every vertex t ($t \neq s$):

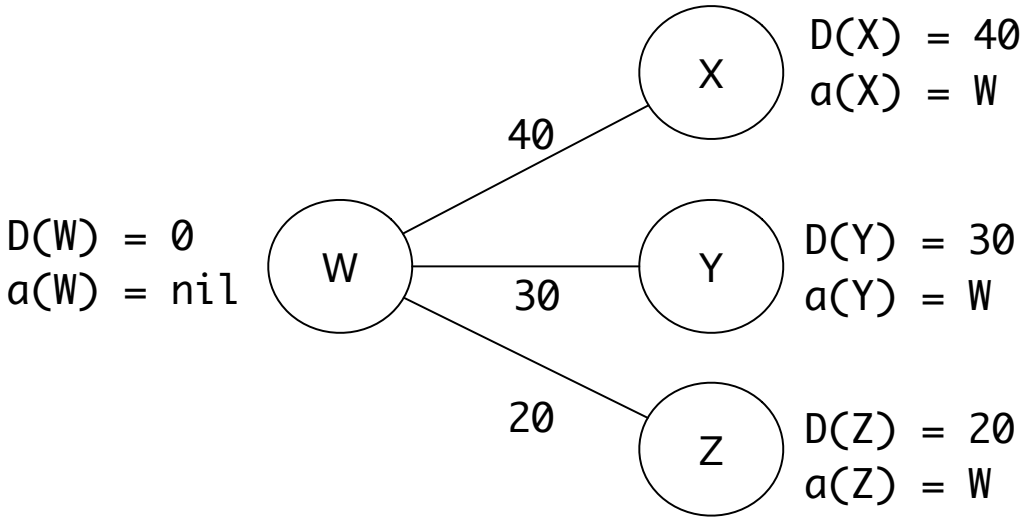
- $\boxed{D(t) = d(s,t)}$ (i.e., weight of *shortest path* from s to t).
- **Reversing** t's ***ancestor path*** $\rightarrow$ *shortest path*: $\langle s, \ldots, a(t), t \rangle$

Dijkstra's Algorithm: Example (1) Input

Perform Dijkstra's algorithm on the following graph, starting with a ***source vertex*** *W*:



Dijkstra's Algorithm: Example (1) Output



History of Q's contents:

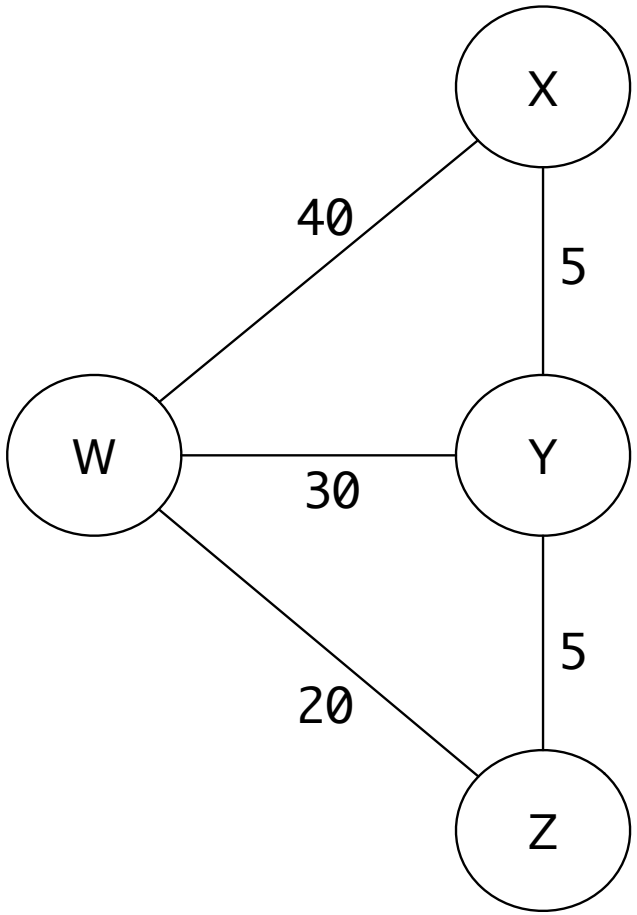
W	X	Y	Z

Shortest paths and distances:

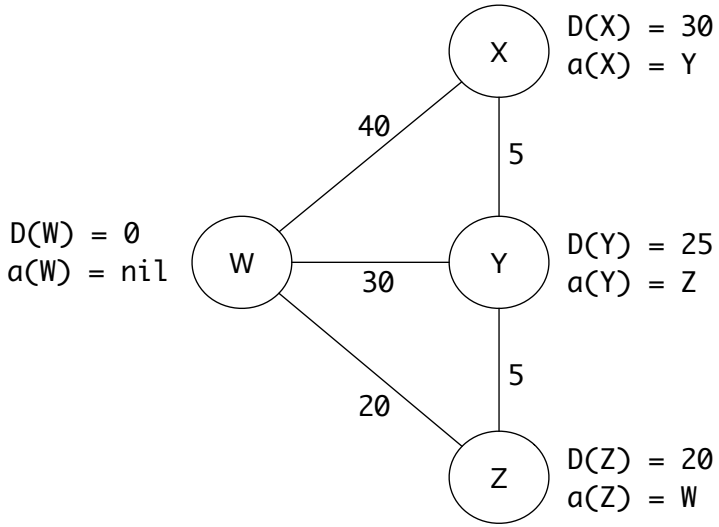
- W to X: $\langle a(X),\ X\rangle = \langle W,\ X\rangle$; $d(W, X) = D(X) = 40$
- W to Y: $\langle a(Y),\ Y\rangle = \langle W,\ Y\rangle$; $d(W, Y) = D(Y) = 30$
- W to Z: $\langle a(Z),\ Z\rangle = \langle W,\ Z\rangle$; $d(W, Z) = D(Z) = 20$

Dijkstra's Algorithm: Example (2) Input

Perform Dijkstra's algorithm on the following graph, starting with a ***source vertex*** *W*:



Dijkstra's Algorithm: Example (2) Output



History of Q's contents:

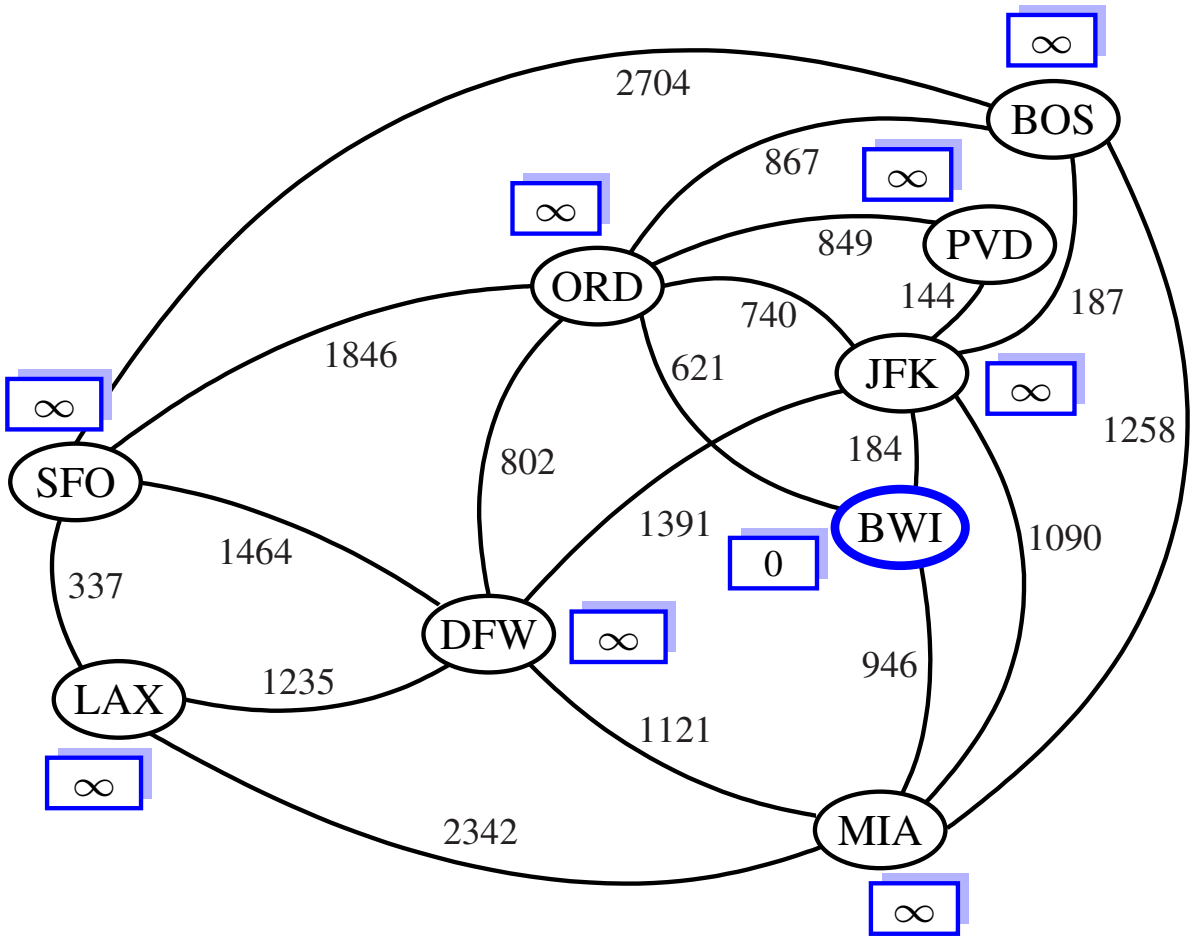
W	X	Y	Z

Shortest paths and distances:

- W to X: $\langle a(Z),\ a(Y),\ a(X),\ X\rangle = \langle W,\ Z,\ Y,\ X\rangle$; $d(W, X) = D(X) = 30$
- W to Y: $\langle a(Z),\ a(Y),\ Y\rangle = \langle W,\ Z,\ Y\rangle$; $d(W, Y) = D(Y) = 25$
- W to Z: $\langle a(Z),\ Z\rangle = \langle W,\ Z\rangle$; $d(W, Z) = D(Z) = 20$

Dijkstra's Algorithm: Exercise

Perform Dijkstra's algorithm on the following initial configuration, starting with a ***source vertex*** *BWI*:



Correctness of Loops

How do we prove that the following loop is correct?

```
{Q}
S_init
while (B) {
   S_body
}
{R}
```

In case of C/Java, B denotes the ***stay condition***.

- In C/Java, there is **not** native, syntactic support for checking the ***correctness*** of loops.
- Instead, we have to **manually** add assertions to encode:
  - **LOOP INVARIANT** [for establishing ***partial correctness***]
  - **LOOP VARIANT** [for ensuring ***termination***]

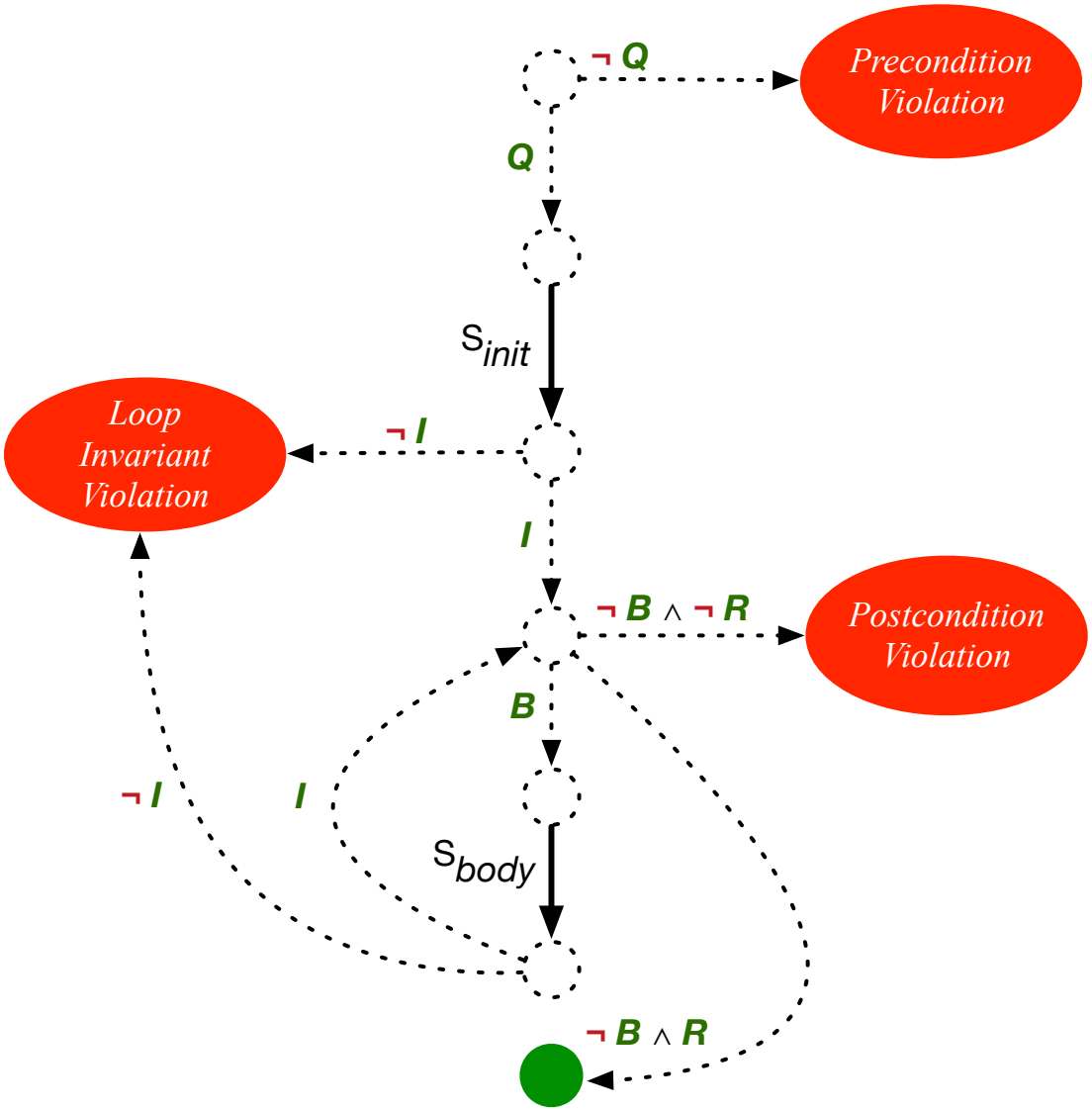
Specifying Loops

- ***Loop Invariant*** (***LI***): Boolean expression for measuring/proving ***partial*** *correctness*
  - **Established** before the very first iteration.
  - **Maintained** TRUE after each iteration.

Specifying Loops: Syntax

```
void myAlgorithm() {
  assert Q; /* Precondition */
  S_init
  assert I; /* Is LI established? */
  while( B ) {
    S_body
    assert I; /* Is LI preserved? */
  }
  assert R; /* Postcondition */
}
```

Specifying Loops: Runtime Checks (1)

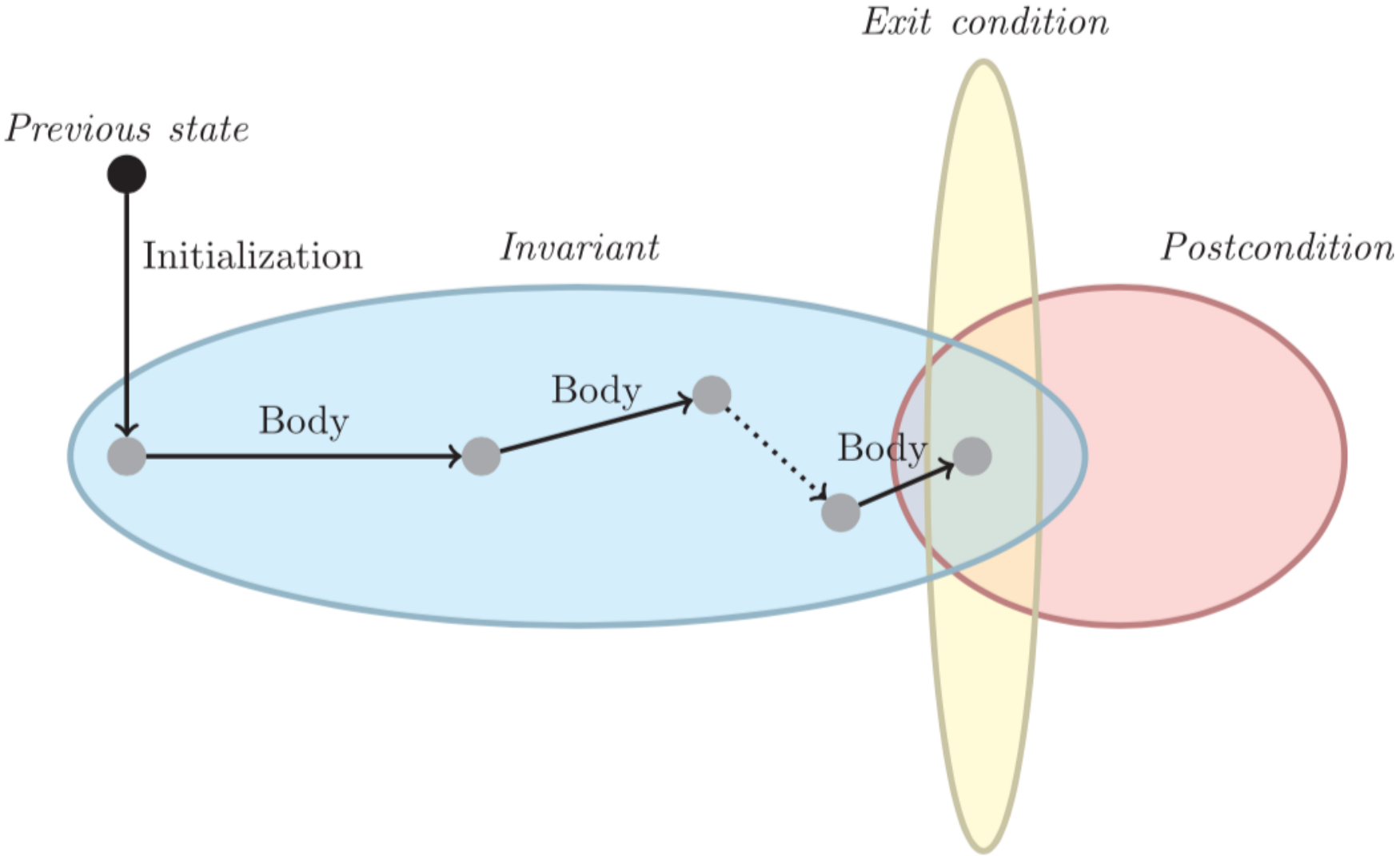


Specifying Loops: Runtime Checks (2)

```
1  void testLI() { /* Assume: integer attribute i */
2    assert i == 1; /* Precondition */
3    assert (1 <= i) && (i <= 6); /* Is LI established? */
4    while (i <= 5) {
5      i = i + 1;
6      assert (1 <= i) && (i <= 6); /* Is LI maintained? */
7    }
8    assert i == 6; /* Postcondition */
9  }
```

L1: Change to `1 <= i && i <= 5` for a ***Loop Invariant Violation***.

Specifying Loops: Visualization



Digram Source: page 5 in *Loop Invariants: Analysis, Classification, and Examples*

Dijkstra's SP Algorithm: Loop Invariant

- Recall: A ***loop invariant*** (***LI***) is a Boolean condition.
  - ***LI*** is establisehd before the 1st iteration.
  - ***LI*** is preserved **at the end of** each subsequent iteration.
- The (iterative) Dijkstra's algorithm has ***LI***:

> For every vertext u that has already been removed from the priority queue Q (i.e., u is considered visited), $D(u)$ equals the ***true*** shortest-path distance from source s to u.

- Formally:

$$\forall u \bullet u \in V \wedge u \notin Q \Rightarrow D(u) = d(s, u)$$

- Important assumption: weights are ***non-negative***.
- To relax this assumption, update **visited** nodes.

[ref: Bellman-Ford]

$\Rightarrow$ Worse running time: $O(|V|^3)$ (rather than $O(|V|^2 \cdot log\ |V|)$)

Running Time of Dijkstra's Algorithm (1)

```
1   ALGORITHM: Dijkstra-Shortest-Path
2     INPUT: Graph G = (V, E); Source Vertex s ∈ V
3     OUTPUT: For t ∈ V (t ≠ s),
4       • D(t) := d(s, t)
5       • Shortest Path: ⟨s, ..., a(a(t)), a(t), t⟩
6   PROCEDURE:
7     D(s) = 0
8     for(t ∈ (V \ {s})): D(t) := ∞
9     for(v ∈ V): a(v) := nil
10    for(v ∈ V): Q.insert(v) -- Q is a PQ keyed by D
11    while ( ¬Q.isEmpty() ):
12      u := Q.min()
13      for(v adjacent to u):
14        if(v ∈ Q ∧ D(u) + w(u, v) < D(v)):
15          D(v) := D(u) + w(u, v)
16          a(v) := u
17        else:
18          skip
19      Q.removeMin()
```

Running Time of Dijkstra's Algorithm (2)

- When implemented using a ***heap***, the ***priority queue*** Q can perform each insertion and deletion in **O(*log n*)** time.
- Given $|V| = n$ and $|E| = m$, ***time compexity*** breaks down to:
  - **L7 – L9**: initializing ***D*** and ***a*** for all vertices [$O(n)$]
  - **L10**: n insertions to Q [$O(n \cdot \log n)$]
    L11: `while` loop has n iterations (**L12** – **L19**).
  - **L12**: retrieving the root of heap n times [$O(n)$]
    Q. How many iterations for **L14 – L18**?
    A. # adjacency edges across all vertices: $\sum_{u \in V} \texttt{degree}(u) = m$
  - **L15**: ***upward bubbling*** to restore relational property of Q [$O(m \cdot \log n)$]
  - **L14,L16-18**: constant operations [$O(m)$]
  - **L19**: removing min-root of heap n times [$O(n \cdot \log n)$]
- Efficient implementation of Dijkstra Algorithm: $O((n + m) \cdot \log n)$
- G almost ***complete*** (i.e., $m = O(n^2)$) $\Rightarrow$RT is $O(n^2 \cdot \log n)$

Graphs in Java: Adjacency Matrix (1)

- Extends the ***edge list*** structure
- Each ***vertex*** *v* also stores an ***integer index*** that is used to index into a 2-dimensional ***adjacency matrix***.

  ⇒ locating an edge between two vertices takes **O(1)**

```
class AdjacencyMatrixVertex<V> extends EdgeListVertex<V> {
  private int index;
  /* getter and setter for index */
}
```

Graphs in Java: Adjacency Matrix (2)

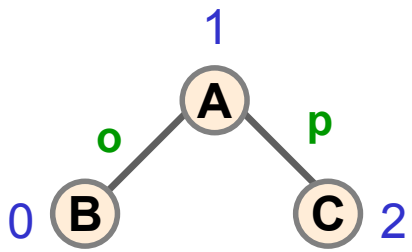
```
class AdjacencyMatrixGraph<V, E> implements Graph<V, E> {
  private DoublyLinkedList<AdjacencyMatrixVertex<V>> vertices;
  private DoublyLinkedList<EdgeListEdge<E, V>> edges;
  private boolean isDirected;

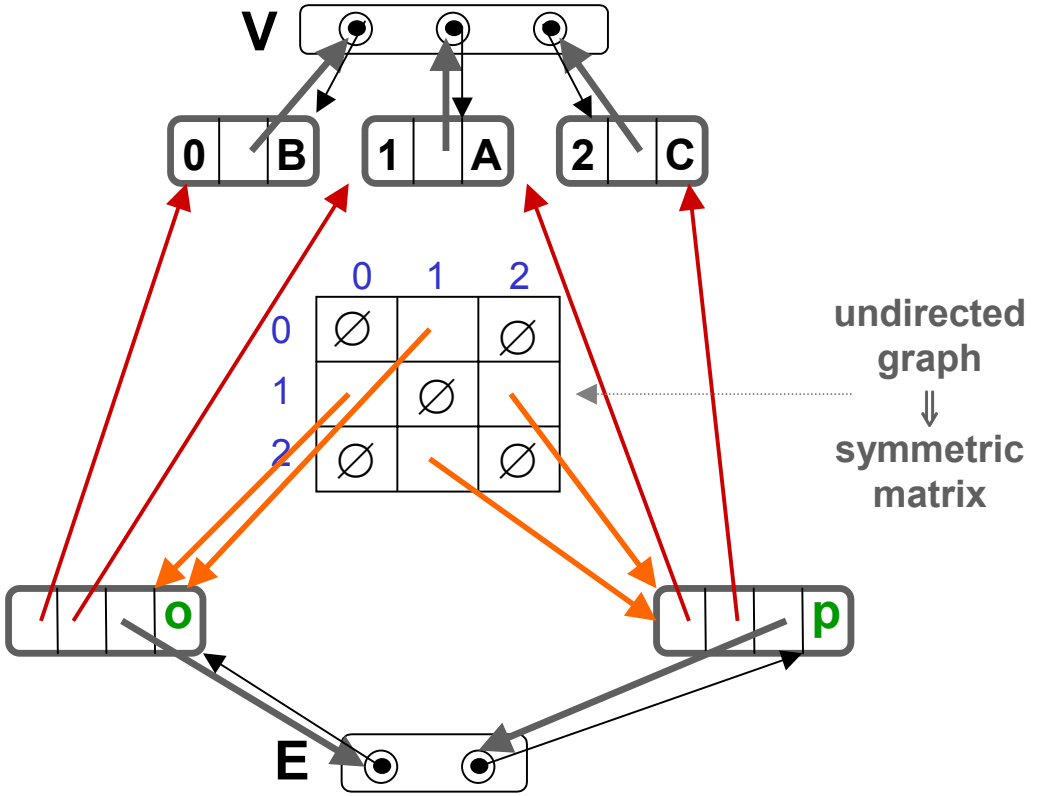
  private EdgeListEdge<E, V>[][] matrix;

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

Space Requirements? $O(|V|^2)$

Graphs in Java: Adjacency Matrix (3)





- Each ***row index*** in `matrix` represents an ***origin vertex***.
- Each ***column index*** in `matrix` represents a ***destination vertex***.
  e.g., edge (B, A): `matrix[0][1]`
- For an undirected graph, cells are **symmetric**.
  e.g., `matrix[0][1] == matrix[1][0]`

Graphs in Java: Comparing Strategies (1)

	VERTEX	EDGE	GRAPH
ADJACENCY LIST	incidentEdges	originIncidentListPos	isDirected vertices edges
		destIncidentListPos	
EDGE LIST	vertexListPosition	edgeListPosition	
ADJACENCY MATRIX			
	index		matrix

Graphs in Java: Comparing Strategies (2)

	Edge List	Adjacency List	Adjacency Matrix
numVertices() numEdges()	O(1)		
vertices()	O(n)		
edges()	O(m)		
getEdge(u, v)	O(m)	***O(min(d_u, d_v))***	***O(1)***
outDegree(v) inDegree(v)	O(m)	***O(d_v)***	O(n)
outgoingEdges(v) incomingEdges(v)	O(m)	***O(d_v)***	O(n)
insertVertex(x)	***O(1)***		***O(n^2)***
removeVertex(v)	O(m)	***O(d_v)***	***O(n^2)***
insertEdge(u, v, x)	O(1)		
removeEdge(e)			

Directed Acyclic Graph (DAG)

- ***Directed Acyclic Graph*** (***DAG***): **directed** graph with **no cycle**.
- A ***DAG*** has many applications where ***dependency*** exists between ***vertices***:
  - e.g., Prerequisites between courses of the undergrad program
  - e.g., Inheritance hierarchy among Java classes
  - e.g., Scheduling constraints between tasks
  - e.g., Dependency betwen variables in transactional updates
- In a ***DAG***, an ***edge*** (v_i, v_j) means v_i "occurs **before**" v_j
  - e.g., (*eecs2101*, *eecs3101*)
  - e.g., (`int x = 0`, `println(x)`)
- Given a ***DAG*** $G = (V, E)$, where $|V| = n$, a ***topological ordering*** of G is a sequence of n vertices

$$v_1, v_2, \ldots, v_n$$

such that

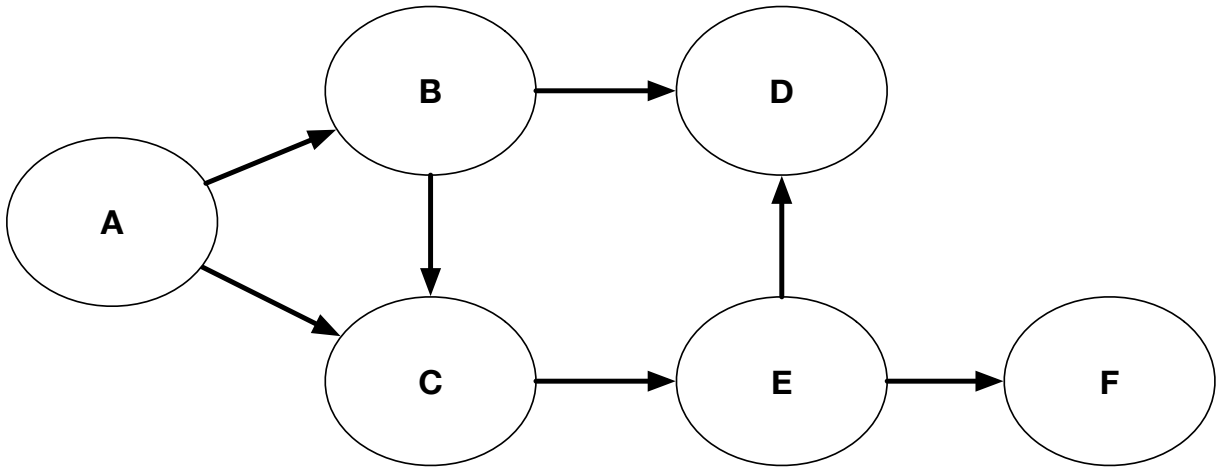
$$\forall i, j \bullet 1 \leq i, j \leq n \land (v_i, v_j) \in E \Rightarrow i < j$$

Using DFS for Topological Sort

- Given a ***DAG***, the process of computing a ***topological ordering*** of G is called performing a ***topological sort***.
- Initialize an empty ***sorted list*** of vertices.
- Repetitively perform an extended verison of **DFS**, say DFS_{topo}, until all vertices in the ***DAG*** are visited.
- Each DFS_{topo}:
  - Starts with an arbitrary unvisited vertex v
  - Returns a ***sorted list*** that corresponds to the **reverse order** in which vertices ***backtracked***.
    - When visiting v, only **push** v's ***adjacent*** vertices that are unvisited.
    - When **popping** a vertex v, add v to the front of the ***sorted list***.

    $\Rightarrow$ The ***sorted list*** contains all vertices **reachable** from v, within the current DFS run (i.e., all vertices that must "occur after" v).
  - Add the produced ***sorted list*** to the front of the list accumulated from the previous DFS_{topo}.

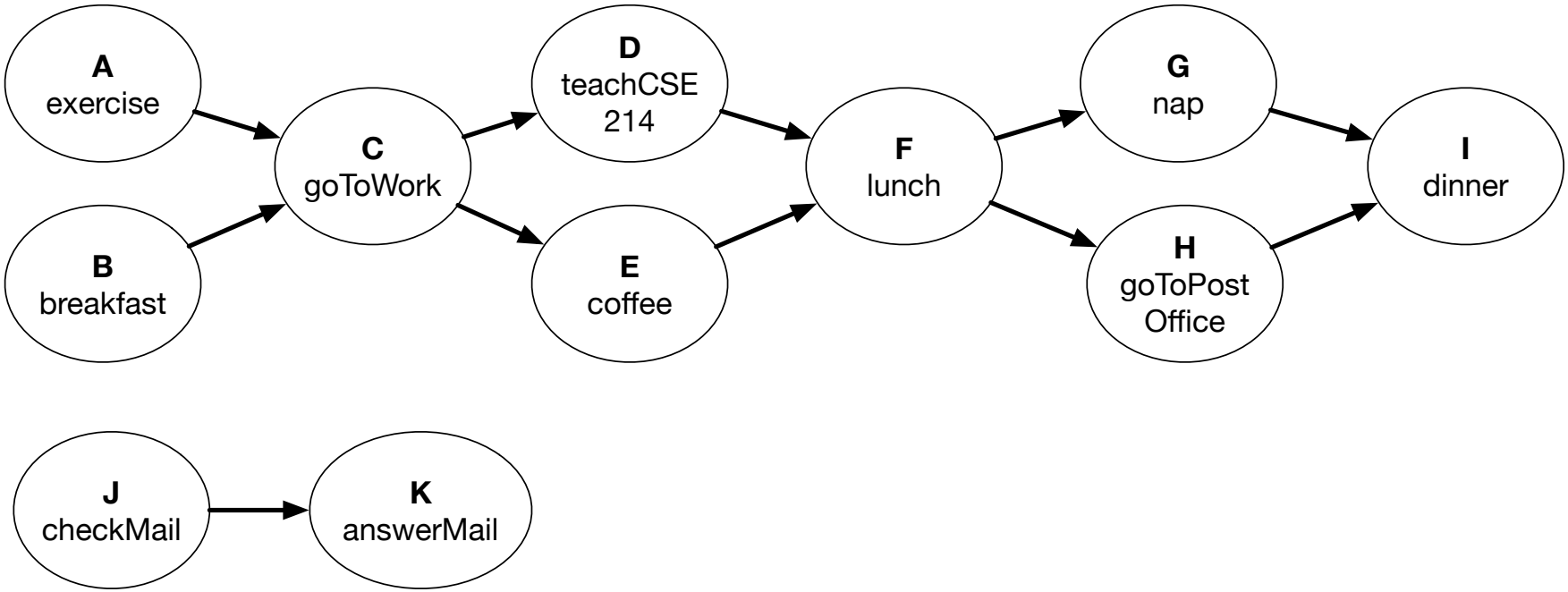
DAG: Illustration (1)



Topologically Sorted lists produced by extended DFS:

- List from $DFS_{topo}(F)$ $\langle F \rangle$
- List from $DFS_{topo}(D)$ $\langle D \rangle$
- List from $DFS_{topo}(E)$ $\langle E, D, F \rangle$ or $\langle E, F, D \rangle$
- List from $DFS_{topo}(C)$ $\langle C, E, D, F \rangle$ or $\langle C, E, F, D \rangle$
- List from $DFS_{topo}(B)$ $\langle B, C, E, D, F \rangle$ or $\langle B, C, E, F, D \rangle$
- List from $DFS_{topo}(A)$ $\langle A, B, C, E, D, F \rangle$ or $\langle A, B, C, E, F, D \rangle$

DAG: Illustration (2)



Possible ***topological orderings*** after a ***topological sort***:

- $\langle A, B, C, D, E, F, G, H, I, J, K \rangle$
- $\langle B, A, C, E, D, F, H, G, I, J, K \rangle$
- $\langle J, B, A, C, E, D, F, H, G, I, K \rangle$

. . .

DAG: Topological Sort in Java (1)

```
Iterable<Vertex<V>> topologicalSort(Graph<V, E> g) {
  ArrayList<Vertex<V>> order = new ArrayList<>();
  for(Vertex<V> v: g.vertices()) {
    if(!v.isVisited()) {
      DFStopo(g, v, order)
    }
  }
  return order;
}
```

DAG: Topological Sort in Java (2)

```
1  DFStopo(Graph<V, E> g, Vertex<V> v, ArrayList<Vertex<V>> order){
2    Stack s = new LinkedStack(); v.setVisited(); s.push(v);
3    while(!s.isEmpty()) {
4      Vertex<V> top = s.peek();
5      Iterator<Edge<E, V>> it = g.outGoingEdges(top);
6      boolean foundUnexploredEdge = false;
7      while(it.hasNext() && !foundUnexploredEdge) {
8        Edge<E, V> e = it.next();
9        Vertex<V> opposite = e.getDestination();
10       if(!opposite.isVisited()) { /* discovery edge */
11         foundUnexploredEdge = true;
12         opposite.setVisited(); s.push(opposite);
13       }
14     }
15     if(!foundUnexploredEdge) { order.addFirst(top); s.pop();}
16   }
17 }
```

Minimum Spanning Trees (MSTs): Problem

- ***Minimum Spanning Tree (MST) problem***:

  Given a **simple**, and **undirected**, **weighted** graph $G = (V, E)$, find a ***spanning tree*** T with the ***minimum*** total weight (over all spanning trees). More precisely:

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

  is the ***minimum*** among all ***spanning trees***.

- Solving the ***MST problem*** in practice:
  - e.g., **Telecom network design**: Build a network connecting all cell towers via fiber links with the minimum installation cost or distance.
  - e.g., **Chip design**: Connect all ground pins through wiring backbone with minimal wire length or delay.

MST Problem: The Greedy Method

- Intuitively, to design using the ***Greedy Method*** :
  - Build an **iterative** solution.
  - At each iteration:
    - Multiple **feasible** choices exist to keep the partial solution valid.
    - Pick the one that looks best **right now** w.r.t. a simple ***cost function***.
    - The choice made is **not** necessarily the **globally** best-looking.
    - After a choice is made, **never** undo it (i.e., **no** ***backtracking***).
- A ***cost/score function*** assigns a number:
  - to ***rank*** possible choices; and
  - to ***measure*** the **partial** solution constructed so far.

  We may either minimize the cost (e.g., smallest edge weight, smallest distance), or maximize the score (e.g., largest profit).
- A ***greedy algorithm*** builds the solution **incrementally**, always picking the ***locally-optimal choice*** (i.e., a **feasible** option with the **best** score at the moment by the ***cost function***).

MST Problem: Kruskal's Algorithm

```
1   ALGORITHM: Find-MST-Kruskal
2     INPUT: Simple, Undirected, Weighted, Connected G = (V, E)
3     OUTPUT: A minimum spanning tree T of G
4   PROCEDURE:
5     for v ∈ V: C(v) := {v} -- build |V| elementary clusters
6     Initialize a priority queue Q containing E -- keyed by weights
7     T := ∅
8     while |T| ≠ n − 1:
9       (u, v) := Q.removeMin()
10      let C(u) be the cluster containing u
11      let C(v) be the cluster containing v
12      if C(u) ≠ C(v) then
13        T := T ∪ {(u, v)}
14        Merge C(u) and C(v) into one cluster
```

MST Problem: Tracing Kruskal's (1)

Apply Kruskal's algorithm on this graph:

$V = \{A, B, C, D, E, F, G, H, G\}$

edge	weight
(A, B)	1
(A, D)	3
(B, C)	2
(B, H)	10
(C, D)	3
(C, F)	6
(D, E)	5
(E, F)	4
(E, H)	8
(F, G)	7
(G, H)	9

MST Problem: Tracing Kruskal's (2)

Iteration	Min Edge	Processing	Resulting Partition	T: MST Under Construction
Init.	—	—	$\left\{ \{A\},\{B\},\{C\},\{D\},\{E\},\{F\},\{G\},\{H\} \right\}$	$\varnothing$
1	$w(A,B)=1$	$\because C(A) \neq C(B) \therefore$ Tree Edge	$\left\{ \{A,B\},\{C\},\{D\},\{E\},\{F\},\{G\},\{H\} \right\}$	$\left\{ (A,B) \right\}$
2	$w(B,C)=2$	$\because C(B) \neq C(C) \therefore$ Tree Edge	$\left\{ \{A,B,C\},\{D\},\{E\},\{F\},\{G\},\{H\} \right\}$	$\left\{ (A,B),(B,C) \right\}$
3	$w(A,D)=3$	$\because C(A) \neq C(D) \therefore$ Tree Edge	$\left\{ \{A,B,C,D\},\{E\},\{F\},\{G\},\{H\} \right\}$	$\left\{ (A,B),(B,C),(A,D) \right\}$
4	$w(C,D)=3$	$\because C(C) = C(D) \therefore$ Internal Edge	No Change	
5	$w(E,F)=4$	$\because C(E) \neq C(F) \therefore$ Tree Edge	$\left\{ \{A,B,C,D\},\{E,F\},\{G\},\{H\} \right\}$	$\left\{ (A,B),(B,C),(A,D),(E,F) \right\}$
6	$w(D,E)=5$	$\because C(D) \neq C(E) \therefore$ Tree Edge	$\left\{ \{A,B,C,D,E,F\},\{G\},\{H\} \right\}$	$\left\{ (A,B),(B,C),(A,D),(E,F),(D,E) \right\}$
7	$w(C,F)=6$	$\because C(C) = C(F) \therefore$ Internal Edge	No Change	
8	$w(F,G)=7$	$\because C(F) \neq C(G) \therefore$ Tree Edge	$\left\{ \{A,B,C,D,E,F,G\},\{H\} \right\}$	$\left\{ (A,B),(B,C),(A,D),(E,F),(D,E),(F,G) \right\}$
9	$w(E,H)=8$	$\because C(E) \neq C(H) \therefore$ Tree Edge	$\left\{ \{A,B,C,D,E,F,G,H\} \right\}$	$\left\{ (A,B),(B,C),(A,D),(E,F),(D,E),(F,G),(E,H) \right\}$

MST Problem: From Clusters to Cuts

- ***Partition*** (of V) [V broken into one or more pieces]
  - A set P of non-empty, ***disjoint*** vertex sets whose **union** equals V.

$$\bigcup_{x \in P} x = V$$
$$\wedge \quad (\forall x_1, x_2 \bullet x_1 \in P \wedge x_2 \in P \wedge x_1 \neq \varnothing \wedge x_2 \neq \varnothing \wedge x_1 \neq x_2 \Rightarrow x_1 \cap x_2 = \varnothing)$$

- ***Cluster*** [one piece of the current partition]
  - In each iteration of executing Kruskal's algorithm (say $x, y \in V$):
    - A ***cluster*** is a set $C(x)$ that's a member of ***partition*** P (i.e., $C(x) \in P$) e.g., **L10** and **L11** in Kruskal's algorithm
    - $C(x) = C(y) \Rightarrow x$ and y in the **same** ***connected component***.
    - $C(x) \neq C(y) \Rightarrow x$ and y in **different** ***connected components***.

- ***Cut*** (of V) [V broken into two pieces]
  - A ***partition*** of V into **two** non-empty, ***disjoint*** sets:

$$P = \{S, V \setminus S\}$$

  - An edge ***crosses*** *the* ***cut*** with its endpoints in S and $V \setminus S$.

MST Problem: Cut Property of Safe Edges

Cut Property. Given:

- $G = (V, E)$ a weighted, ***connected*** graph
- Any ***cut*** $\{S, V \setminus S\}$ of the vertices
- e a ***minimum-weight edge*** among all edges that ***cross*** *this* ***cut***

$\Rightarrow$ There exists **an** ***MST*** that contains e.

We say: e is a ***safe edge*** for some ***MST***.

In Kruskal's algorithm:

- **L10**: ***Cluster*** $C(u)$ helps form a ***cut***: $\{C(u), V \setminus C(u)\}$
  - $C(u) \neq \varnothing$
  - $V \setminus C(u) \neq \varnothing$ [i.e., not in the **last** iteration]
- **L12**: $C(u) \neq C(v)$ means $C(u) \cap C(v) = \varnothing$ and $C(v) \subseteq V \setminus C(u)$.
  - Recall: Edges extracted from Q in a non-decreasing order on weights
  - ***Clusters*** only **merge**; never split.
  - Any edge (x, y) with $x \in C(u)$ and $w(x, y) < w(u, v)$ would've been processed (as a **tree** edge or as an **internal** edge), so now $x, y \in C(u)$.
    $\Rightarrow \nexists$ edge "cheaper" than (u, v) ***crossing*** *the* ***cut*** $\{C(u), V \setminus C(u)\}$

  $\therefore (u, v)$ from **L9** is a ***min-weight edge*** ***crossing*** *the* ***cut***
- By the ***Cut Property***, **L13** is justified:
  - (u, v) is a ***safe edge***: $\exists$ some ***MST*** containing $T \cup \{(u, v)\}$
  - Adding (u, v) keeps us **on the right track** to reach some ***MST***.

Index (1)

Learning Outcomes of this Lecture

Graphs: Definition

Directed vs. Undirected Edges

Self vs. Parallel Edges

Vertices

Exercise (1)

Directed vs. Undirected Graphs

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

Index (2)

Basic Properties of Graphs (1)

Basic Properties of Graphs (2)

Basic Properties of Graphs (3)

Paths and Cycles (1)

Paths and Cycles (2)

Subgraphs vs. Spanning Subgraphs

Connected Graph vs. Connected Components

Forests vs. Trees

Spanning Trees

Exercise (2)

Basic Properties of Graphs (4)

Index (3)

Graph Traversals: Definition

Graph Traversals: Applications

Depth-First Search (DFS)

DFS on Undirected Graph: Marking Vertices and Edges

DFS on Undirected Graph: Illustration (1.1)

DFS on Undirected Graph: Illustration (1.2)

DFS on Undirected Graph: Illustration (2)

DFS on a Directed Graph: Notations (1)

DFS on a Directed Graph: Notations (2)

DFS on a Directed Graph: Exercises

Index (4)

DFS: Properties

Graph Questions: Adapting DFS

Graphs in Java: DL Node and List

Graphs in Java: Vertex and Edge

Graphs in Java: Interface (1)

Graphs in Java: Interface (2)

Graphs in Java: Interface (3)

Graphs in Java: Edge List (1)

Graphs in Java: Edge List (2)

Graphs in Java: Edge List (3)

Breadth-First Search (BFS)

Index (5)

BFS in Java: Marking Vertices and Edges

Iterative BFS: Illustration (1.1)

Iterative BFS: Illustration (1.2)

BFS: Illustration (2)

BFS: Properties

Graph Questions: Adapting BFS

Graphs in Java: Adjacency List (1)

Graphs in Java: Adjacency List (2)

Graphs in Java: Adjacency List (3)

Weighted Graphs

Shortest Paths in Weighted Graphs

Index (6)

Dijkstra's Shortest Path Algorithm

Dijkstra's Algorithm: Example (1) Input

Dijkstra's Algorithm: Example (1) Output

Dijkstra's Algorithm: Example (2) Input

Dijkstra's Algorithm: Example (2) Output

Dijkstra's Algorithm: Exercise

Correctness of Loops

Specifying Loops

Specifying Loops: Syntax

Specifying Loops: Runtime Checks (1)

Specifying Loops: Runtime Checks (2)

Index (7)

Specifying Loops: Visualization

Dijkstra's SP Algorithm: Loop Invariant

Running Time of Dijkstra's Algorithm (1)

Running Time of Dijkstra's Algorithm (2)

Graphs in Java: Adjacency Matrix (1)

Graphs in Java: Adjacency Matrix (2)

Graphs in Java: Adjacency Matrix (3)

Graphs in Java: Comparing Strategies (1)

Graphs in Java: Comparing Strategies (2)

Directed Acyclic Graph (DAG)

Using DFS for Topological Sort

Index (8)

DAG: Illustration (1)

DAG: Illustration (2)

DAG: Topological Sort in Java (1)

DAG: Topological Sort in Java (2)

Minimum Spanning Trees (MSTs): Problem

MST Problem: The Greedy Method

MST Problem: Kruskal's Algorithm

MST Problem: Tracing Kruskal's (1)

MST Problem: Tracing Kruskal's (2)

MST Problem: From Clusters to Cuts

MST Problem: Cut Property of Safe Edges