Graph Terminology and Representations

Graphs

- A graph is a pair \((V, E)\), where
  - \(V\) is a set of nodes, called vertices
  - \(E\) is a collection of pairs of vertices, called edges
- Vertices and edges are positions and store elements
- Example:
  - A vertex represents an airport and stores the three-letter airport code
  - An edge represents a flight route between two airports and stores the mileage of the route

![Graph Diagram]

Edge Types

- Directed edge
  - ordered pair of vertices \((u,v)\)
  - first vertex \(u\) is the origin
  - second vertex \(v\) is the destination
  - e.g., a flight

- Undirected edge
  - unordered pair of vertices \((u,v)\)
  - e.g., a flight route

- Directed graph
  - all the edges are directed
  - e.g., route network

- Undirected graph
  - all the edges are undirected
  - e.g., flight network

Applications

- Electronic circuits
  - Printed circuit board
  - Integrated circuit

- Transportation networks
  - Highway network
  - Flight network

- Computer networks
  - Local area network
  - Internet
  - Web

- Databases
  - Entity-relationship diagram
Terminology

- **End vertices (or endpoints) of an edge**
  - U and V are the endpoints of a

- **Edges incident on a vertex**
  - a, d, and b are incident on V

- **Adjacent vertices**
  - U and V are adjacent

- **Degree of a vertex**
  - X has degree 5

- **Parallel edges**
  - h and i are parallel edges

- **Loop**
  - j is a loop

Terminology (cont.)

- **Path**
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
  - it contains at least one edge

- **Simple path**
  - path such that all its vertices and edges are distinct

- **Examples**
  - \( P_1 = (V, b, X, h, Z) \) is a simple path
  - \( P_2 = (U, c, W, e, X, g, Y, f, W, d, V) \) is a path that is not simple
Terminology (cont.)

- **Cycle**
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints

- **Simple cycle**
  - cycle such that all its vertices and edges are distinct

- **Examples**
  - $C_1 = \{(V,b,X,g,Y,f,W,c,U,a,\ldots)\}$ is a simple cycle
  - $C_2 = \{(U,c,W,e,X,g,Y,f,W,d,V,a,\ldots)\}$ is a cycle that is not simple

- Edges can be dropped if no multiple edges exist.

Properties

**Property 1**

\[ \sum_v \deg(v) = 2m \]

**Notation**

- $n$: number of vertices
- $m$: number of edges
- $\deg(v)$: degree of vertex $v$

**Proof:**

- each edge is counted twice

**Property 2**

In an undirected graph with no loops and no multiple edges

\[ m \leq n(n-1)/2 \]

**Proof:**

- each vertex has degree at most $(n - 1)$

What is the bound for a directed graph?

**Example**

- $n = 4$
- $m = 6$
- $\deg(v) = 3$
Vertices and Edges

- A **graph** is a collection of **vertices** and **edges**.
- A **Vertex** is can be an abstract unlabeled object or it can be labeled (e.g., with an integer number or an airport code) or it can store other objects.
- An **Edge** can likewise be an abstract unlabeled object or it can be labeled (e.g., a flight number, travel distance, cost), or it can also store other objects.

Relations vs Graph

- A relation R on the set A is a subset of $A \times A$.
- There is 1-to-1 correspondence between R and (directed) $G=(A, R)$.

**Example:** Let $A = \{1, 2, 3, 4\}$. Which ordered pairs are in the relation $R = \{(a, b) \mid a < b\}$?

$$R = \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$$
Representing Relations Using Digraphs

- **Example:** Display the digraph with $V = \{a, b, c, d\}$, $E = \{(a, b), (a, d), (b, b), (b, d), (c, a), (c, b), (d, b)\}$.

An edge of the form $(b, b)$ is called a **loop**.

---

Relations on a Set

- **How many different relations can we define on a set $A$ with $n$ elements?**
  - A relation on a set $A$ is a subset of $A \times A$.
  - How many elements are in $A \times A$?
  - The number of subsets that we can form out of a set with $m$ elements is $2^m$. Therefore, $2^{n^2}$ subsets can be formed out of $A \times A$.

- **Answer:** We can define $2^{n^2}$ different relations on $A$. As a result, we have that much directed graphs on $n$ points.

- **How many different undirected graphs over $n$ points?**
Properties of Relations

- **Definition:** A relation R on a set A is called **reflexive** if \((a, a) \in R\) for every element \(a \in A\).

- The graph that each node has a loop represents a reflexive relation.

- How many different loop-free directed graphs over \(n\) points?

---

Properties of Relations

**Definitions:**

- A relation R on a set A is called **symmetric** if \((b, a) \in R\) whenever \((a, b) \in R\) for all \(a, b \in A\).
  - Every undirected graph represents a symmetric relation.

- A relation R on a set A is called **antisymmetric** if \(a = b\) whenever \((a, b) \in R\) and \((b, a) \in R\).
  - \((N, \leq)\) is antisymmetric

- A relation R on a set A is called **asymmetric** if \((a, b) \in R\) implies that \((b, a) \notin R\) for all \(a, b \in A\).
  - \((N, <)\) is asymmetric

- What is the relation between “antisymmetric” and “asymmetric”?
  - \(R\) is asymmetric iff \(R\) is antisymmetric and has no loops.
Properties of Relations

- **Definition:** A relation $R$ on a set $A$ is called **transitive** if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for $a, b, c \in A$.

- Whenever there is a path that goes from $a$ to $b$, then there is an edge $(a, b)$ in the graph, then the graph represents a transitive relation.

- Are the following relation on $\{1, 2, 3\}$ transitive?
  
  $R = \{(1, 1), (1, 2), (2, 1), (3, 3)\}$

Combining Relations

- **Definition:** Let $R$ be a relation from a set $A$ to a set $B$ and $S$ a relation from $B$ to a set $C$. The **composite** of $R$ and $S$ is the relation consisting of ordered pairs $(a, c)$, where $a \in A$, $c \in C$, and for which there exists an element $b \in B$ such that $(a, b) \in R$ and $(b, c) \in S$. We denote the composite of $R$ and $S$ by $S \circ R$.

- If $A = B = C$, and $S = R$, then $R \circ R$ can be written as $R^2$.

- If $R$ is represented by a graph, then $(a, b)$ is in $R^2$ iff there is a path of length 2 from $a$ to $b$.

- In general, $(a, b)$ is in $R^k$ iff there is a path of length $k$ from $a$ to $b$. 
Combining Relations

- **Definition**: Let $R$ be a relation on the set $A$. The powers $R^k$, $k = 1, 2, 3, \ldots$, are defined inductively by
  - $R^1 = R$
  - $R^{k+1} = R^k \circ R$

- In other words: $R^k = R \circ R \circ \ldots \circ R$ (k times the letter $R$)

- The relation $R^* = R^1 \cup R^2 \cup R^3 \cup \ldots \cup R^{n-1}$, where $n$ is the number of nodes, is called the transitive closure of $R$.

- To decide if $(a, b)$ in $R^*$, we need to decide if there is a path from $a$ to $b$ in $G = (A, R)$.

**Theorem**: The relation $R$ on a set $A$ is transitive if and only if $R^k \subseteq R$ for all positive integers $k$.

- Remember the definition of transitivity:
  - **Definition**: A relation $R$ on a set $A$ is called transitive if whenever $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$ for $a, b, c \in A$.

- The composite of $R$ with itself contains exactly these pairs $(a, c)$.
  - Therefore, for a transitive relation $R$, $R \circ R$ does not contain any pairs that are not in $R$, so $R \circ R \subseteq R$.
  - Since $R \circ R$ does not introduce any pairs that are not already in $R$, it must also be true that $(R \circ R) \circ R \subseteq R$, and so on, so that $R^k \subseteq R$. 

Equivalence Relations

- **Equivalence relations** are used to relate objects that are similar in some way.

- **Definition:** A relation on a set $A$ is called an equivalence relation if it is reflexive, symmetric, and transitive.

- Two elements that are related by an equivalence relation $R$ are called **equivalent**.

- The best representation of an equivalence relation is Sets.

Adjacency List Structure

- Incidence sequence for each vertex
  - sequence of references to edge objects of incident edges
- Augmented edge objects
  - references to associated positions in incidence sequences of end vertices
Adjacency Matrix Structure

- Edge list structure
- Augmented vertex objects
  - Integer key (index) associated with vertex
- 2D-array adjacency array
  - Reference to edge object for adjacent vertices
  - Null for non-adjacent vertices
- The "old fashioned" version just has 0 for no edge and 1 for edge

Graph Representations

Option 1:

Class Node

  String: Name
  Boolean: Visited
  List<Node>: Neighbors
  List<Integer>: Costs

End Node

Option 2:

Class Node

  String: Name
  Boolean: Visited
  List<Edge>: Edges

End Node

Class Edge

  Integer: Cost
  Node: toNode
  Node: fromNode

End Edge
Lists and Iterators

Performance
(All bounds are big-oh running times, except for "Space")

<table>
<thead>
<tr>
<th></th>
<th>Edge List</th>
<th>Adjacency List</th>
<th>Adjacency Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Space</strong></td>
<td>$n + m$</td>
<td>$n + m$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>IncidentEdges($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$n$</td>
</tr>
<tr>
<td>areAdjacent ($v$, $w$)</td>
<td>$m$</td>
<td>$\min(\text{deg}(v), \text{deg}(w))$</td>
<td>1</td>
</tr>
<tr>
<td>insertVertex($o$)</td>
<td>1</td>
<td>1</td>
<td>$n^2$</td>
</tr>
<tr>
<td>insertEdge($v$, $w$, $o$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>removeVertex($v$)</td>
<td>$m$</td>
<td>$\text{deg}(v)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>removeEdge($e$)</td>
<td>1</td>
<td>$\text{deg}(v)$</td>
<td>1</td>
</tr>
</tbody>
</table>

Subgraphs
- A subgraph $S$ of a graph $G$ is a graph such that
  - The vertices of $S$ are a subset of the vertices of $G$
  - The edges of $S$ are a subset of the edges of $G$
- A spanning subgraph of $G$ is a subgraph that contains all the vertices of $G$
Application: Web Crawlers

- A fundamental kind of algorithmic operation that we might wish to perform on a graph is **traversing the edges and the vertices** of that graph.
- A **traversal** is a systematic procedure for exploring a graph by examining all of its vertices and edges.
- For example, a **web crawler**, which is the data collecting part of a search engine, must explore a graph of hypertext documents by examining its vertices, which are the documents, and its edges, which are the hyperlinks between documents.
- A traversal is efficient if it visits all the vertices and edges in linear time.

Connectivity

- A graph is connected if there is a path between every pair of vertices.
- A connected component of a graph $G$ is a maximal connected subgraph of $G$. 
Trees and Forests

- A (free) tree is an undirected graph $T$ such that:
  - $T$ is connected
  - $T$ has no cycles
  - This definition of tree is different from the one of a rooted tree
- A forest is an undirected graph without cycles
- The connected components of a forest are trees

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- Spanning trees have applications to the design of communication networks
- A spanning forest of a graph is a spanning subgraph that is a forest
# Depth-First Search

- Depth-first search (DFS) is a general technique for traversing a graph.
- A DFS traversal of a graph $G$:
  - Visits all the vertices and edges of $G$.
  - Determines whether $G$ is connected.
  - Computes the connected components of $G$.
  - Computes a spanning forest of $G$.

- DFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time.
- DFS can be further extended to solve other graph problems:
  - Find and report a path between two given vertices.
  - Find a cycle in the graph.

- Depth-first search can be done iteratively or recursively, and the results are different if a node has multiple children.

---

## Depth-First Traversal with Marking

```python
DFS_recur(Node: node):
    <Process node>
    node.Visited = True
    for each edge in node.Edges
        if (not edge.toNode.Visited) then
            DFS_recur(edge.toNode)
            edge.toNode.parent = node
        end if
    end for
end DFS_recur
```

Complexity: $O(n + m)$, $n$ and $m$ are the numbers of nodes and edges, resp.
Depth-First Traversal with Time-Stamp

```
DFS_recur(Node: node)
  <Process node>
  node.StartTime = ++time // time is global
  for each edge in Edges
    if (edge.toNode.StartTime == 0) then
      DFS_recur(edge.toNode)
      edge.toNode.parent = node
    end if
  end for
  node.FinishTime = ++time // optional
end DFS_recur
```

Color of a node: white if StartTime is undefined; gray if StartTime is defined but FinishTime is undefined; black if FinishTime is defined.

Example

- **unexplored vertex**
  - **visited vertex**
  - **unexplored edge**
  - **discovery edge**
  - **back edge**
Example (cont.)

Non-Recursive DFS

Non-Recursive DFS

```plaintext
DepthFirstTraverse(Node: start_node)

start_node.Visited = True  // Visit this node.
// Make a stack and put the start node in it.
Stack[Node]: stack;    stack.Push(start_node);
// Repeat as long as the stack isn't empty.
while not stack.IsEmpty() do
    Node node = stack.Pop() // Get the next node from the stack.
    for each edge in node.Edges // Process the node's Edges.
        // if toNode hasn't been visited...
        if (not Edge.toNode.Visited) then
            // Mark the node as visited and may set StartTime
            Edge.toNode.Visited = True
            Edge.toNode.parent = node
            stack.Push(Edge.toNode) // Push the node onto the stack.
        end if
    end for // may set FinishTime of node.
end while  // Continue processing the stack until empty.
end DepthFirstTraverse
```

3 stages of a node: not visited (white), in stack (grey), exited stack (black)
Depth-First Search

- Starting from A, give the finishing time for each vertex when the recursive DFS is used.
- Repeat the above exercise when non-recursive DFS is used.

Possible Quiz Questions:
- Starting from a, give the finishing time for each vertex when the recursive DFS is used.
- Repeat the above exercise when non-recursive DFS is used.
DFS and Maze Traversal

- The DFS algorithm is similar to a classic strategy for exploring a maze.
  - We mark each intersection, corner, and dead end (vertex) visited.
  - We mark each corridor (edge) traversed.
  - We keep track of the path back to the entrance (start vertex) by means of a rope (recursion stack).

Properties of DFS

Property 1

$\text{DFS}(G, v)$ visits all the vertices and edges in the connected component of $v$.

Property 2

The discovery (tree) edges labeled by $\text{DFS}(G, v)$ form a spanning tree of the connected component of $v$ (Recursive and Non-recursive DFS produce different trees, and different start and finish times).
The General DFS Algorithm

- Perform a DFS from each unexplored vertex, and produce a forest of DFS trees:

```
Algorithm DFS(G):
    Input: A graph G
    Output: A labeling of the vertices in each connected component of G as explored
    Initially label each vertex in G as unexplored
    for each vertex, v, in G do
        if v is unexplored then
            DFS(G, v)
```

Analysis of DFS

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge in an undirected graph is seen twice
  - once as DISCOVERY (i.e., TREE edge)
  - once as BACK
- Each edge in a directed graph is seen once
  - as TREE, BACK, CROSS, or FORWARD edges
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \text{deg}(v) = 2m$
Breadth-First Search

- Breadth-first search (BFS) is a general technique for traversing a graph
- A BFS traversal of a graph $G$
  - Visits all the vertices and edges of $G$
  - Determines whether $G$ is connected
  - Computes the connected components of $G$
  - Computes a spanning forest of $G$
- BFS on a graph with $n$ vertices and $m$ edges takes $O(n + m)$ time
- BFS can be further extended to solve other graph problems
  - Find and report a path with the minimum number of edges between two given vertices
  - Find a simple cycle, if there is one

Example

- $A$ unexplored vertex
- $A$ visited vertex
- $L_0$ unexplored edge
- $L_1$ discovery edge
- $L_1$ cross edge
Example (cont.)

Example (cont.)
Properties

Notation

\( G_s \): connected component of \( s \)

Property 1

\( \text{BFS}(G, s) \) visits all the vertices and edges of \( G_s \)

Property 2

The discovery edges labeled by \( \text{BFS}(G, s) \) form a spanning tree \( T_s \) of \( G_s \)

Property 3

For each vertex \( v \) in \( L_i \)
- The path of \( T_s \) from \( s \) to \( v \) has \( i \) edges
- Every path from \( s \) to \( v \) in \( G_s \) has at least \( i \) edges

BFS Algorithm

- The algorithm uses "levels" \( L_i \) and a mechanism for setting and getting "labels" of vertices and edges.

```
Algorithm BFS(G, s):

Input: A graph G and a vertex s of G
Output: A labeling of the edges in the connected component of s as discovery edges and cross edges

Create an empty list, \( L_0 \)
Mark s as explored and insert s into \( L_0 \)
i = 0

while \( L_i \) is not empty do
    create an empty list, \( L_{i+1} \)
    for each vertex, v, in \( L_i \) do
        for each edge, e = (v, w), incident on v in G do
            if edge e is unexplored then
                if vertex w is unexplored then
                    Label e as a discovery edge
                    Mark w as explored and insert w into \( L_{i+1} \)
                else
                    Label e as a cross edge
            end
        end
    end
    i = i + 1
end
```
Breadth-First Traversal

BreadthFirstTraverse(Node: start_node)
  start_node.Visited = True   // Visit this node.
  // Make a stack and put the start node in it.
  Queue[Node]: queue;    queue.add(start_node);
  // Repeat as long as the stack isn’t empty.
  While <queue isn’t empty>
    Node node = queue.remove() // Get the next node from the queue.
    // Process the node’s Edges.
    For each edge In node.Edges
      // if toNode hasn’t been visited…
      If (Not Edge.toNode.Visited) Then
        // Mark the node as visited and set StartTime
        Edge.toNode.Visited = True
        // Push the node onto the stack.
        stack.Push(Edge.toNode)
      End If
    End for   // Set FinishTime of node.
  Loop // Continue processing the queue until empty
End DepthFirstTraverse

3 stages of a node: not visited (white), in queue (grey), exited queue (black)

Analysis

- Setting/getting a vertex/edge label takes $O(1)$ time
- Each vertex is labeled twice
  - once as UNEXPLORED
  - once as VISITED
- Each edge is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or CROSS
- Each vertex is inserted once into a sequence $L_i$
- Method incidentEdges is called once for each vertex
- BFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
Applications

- We can use the BFS traversal algorithm, for a graph $G$, to solve the following problems in $O(n + m)$ time:
  - Compute the connected components of $G$
  - Compute a spanning forest of $G$
  - Find a simple cycle in $G$, or report that $G$ is a forest
  - Given two vertices of $G$, find a path in $G$ between them with the minimum number of edges, or report that no such path exists

DFS vs. BFS

<table>
<thead>
<tr>
<th>Applications</th>
<th>DFS</th>
<th>BFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spanning forest, connected components, paths, cycles</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Shortest paths</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>Biconnected components</td>
<td>√</td>
<td></td>
</tr>
</tbody>
</table>

DFS (Depth-First Search) vs. BFS (Breadth-First Search)

DFS Example:

1. Start at A
2. Visit B, then C, then D, then E, then F
3. Backtrack to B
4. Visit G and H

BFS Example:

1. Start at A
2. Visit B, then C, then D, then E, then F
3. Continue...

Note: DFS explores as far as possible along each branch before backtracking, while BFS explores the graph in a breadth-first manner.
DFS vs. BFS (cont.)

Back edge $(v, w)$
- $w$ is an ancestor of $v$ in the tree of discovery edges

Cross edge $(v, w)$
- $w$ is in the same level as $v$ or in the next level

Digraphs

- A **digraph** is a shorthand for directed graph whose edges are all directed
- Applications
  - one-way streets
  - flights
  - task scheduling
Digraph Properties

- A graph $G=(V,E)$ such that
  - Each edge goes in one direction:
    - Edge $(a,b)$ goes from $a$ to $b$, but not $b$ to $a$
- If $G$ is simple, $m \leq n \cdot (n - 1)$
- If we keep in-edges and out-edges in separate adjacency lists, we can perform listing of incoming edges and outgoing edges in time proportional to their size

Digraph Application

- **Scheduling**: edge $(a,b)$ means task $a$ must be completed before $b$ can be started

---

The good life

- cs4330, cs4340
- cs4350, ...

- cs1210
- cs2120
- cs2230
- cs2320
- cs3320
- cs3520
- cs3620
- cs3640
- cs3820
- cs4640
- cs4840
- cs5040
- cs5540
- The good life
Directed DFS

- We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction.
- In the directed DFS algorithm, we have four types of edges:
  - discovery (tree) edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex \( s \) determines the vertices reachable from \( s \).

Edge classification by DFS

- Tree edges
- Forward edges
- Back edges
- Cross edges

The edge classification depends on the particular DFS tree!
Edge classification by DFS

Edge classification by DFS

Edge classification by DFS

The edge classification depends on the particular DFS tree!

Edge (u,v) of G is classified as:

1. Tree edge iff u discovers v during the DFS: P[v] = u
   i.e., v.StartTime is undefined (v is white).

   If (u,v) is NOT a tree edge then it is:

2. Back edge iff u is a descendant of v in the DFS tree
   i.e., v.FinishTime is undefined (v is grey).

3. Forward edge iff u is an ancestor of v in the DFS tree
   i.e., v.FinishTime is defined (v is black) and

4. Cross edge iff u is neither an ancestor nor a descendant of v
   i.e. v.FinishTime is defined (v is black) and
   u.StartTime > v.FinishTime (v is black).
Reachability

- DFS tree rooted at $v$: vertices reachable from $v$ via directed paths

DAGs and back edges

- Can there be a **back** edge in a DFS on a Directed Acyclic Graph (DAG)?
- NO! Back edges form a cycle!
- A graph $G$ is a DAG $\iff$ there is no back edge classified by DFS($G$)
DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles.
- A topological ordering of a digraph is a numbering \( v_1, \ldots, v_n \) of the vertices such that for every edge \((v_i, v_j)\), we have \( i < j \).
- Example: in a task scheduling digraph, a topological ordering is a task sequence that satisfies the precedence constraints.

Theorem

A digraph admits a topological ordering if and only if it is a DAG.

Topological Sorting

- Number vertices, so that \((u, v)\) in \(E\) implies \(u < v\).

A typical student day:
1. Wake up
2. Study computer sci.
3. Eat
4. Nap
5. More c.s.
6. Work out
7. Play
8. Write c.s. program
9. Bake cookies
10. Sleep
11. Dream about graphs
Algorithm for Topological Sorting

- Note: This algorithm is different than the one in the book

```
Algorithm TopologicalSort(G)
    H ← G       // Temporary copy of G
    t ← 1
    while H is not empty do
        Let v be a vertex with no ingoing edges
        Label v ← t
        t ← t + 1
        Remove v from H
```

- Running time: O(n + m)

Topological Sorting Example

[Diagram of a graph showing topological sorting example]

63
Implementation with DFS

- Simulate the algorithm by using depth-first search
- \(O(n+m)\) time.

**Algorithm topologicalDFS(G, v)**

- **Input** graph \(G\) and a start vertex \(v\) of \(G\)
- **Output** labeling of the vertices of \(G\) in the connected component of \(v\)

```
setLabel(v, VISITED)
for all \(e \in \text{G.outEdges}(v)\) {
  outgoing edges
  \(w \leftarrow \text{theOtherEnd}(v, e)\)
  if getLabel(w) = UNEXPLORED {
    e is a discovery edge
    topologicalDFS(G, w)
  }
  else {
    e is a forward or cross edge
  }
  Label v with topological number \(t\)
  \(t \leftarrow t - 1\)
```

Topological number = \(n - \text{finishTime} + 1\)

---

**Topological Sorting Example**

A directed graph showing the topological order of its vertices.
Topological Sorting Example

Topological Sorting Example
Topological Sorting Example

Diagram 1:

Topological Sorting Example

Diagram 2:
Topological Sorting Example

Diagram of a directed graph with nodes labeled from 3 to 9. The graph has directed edges connecting the nodes in a way that ensures a topological order can be determined.
Topological Sorting Example

Possible Quiz Question

List the vertices of this graph in Topological Order:
Strong Connectivity

- Each vertex can reach all other vertices

Application: Networking

- A computer network can be modeled as a graph, where vertices are routers and edges are network connections between edges.
- A router can be considered **critical** if it can disconnect the network for that router to fail.
- It would be nice to identify which routers are critical.
- We can do such an identification by solving the biconnected components problem.
Strongly Connected Components

- Any directed graph can be partitioned into a unique set of strong components.

- The algorithm for finding the strong components of a directed graph $G$ uses the transpose of the graph.$$
\text{The transpose } G^T \text{ has the same set of vertices } V \text{ as graph } G, \text{ but a new edge set consisting of the edges of } G \text{ but with the opposite direction.}$$
Strongly Connected Components

- Execute the depth-first search `dfs()` for the graph `G` which creates the list `dfsList` consisting of the vertices in `G` in the reverse order of their finishing times.
- Generate the transpose graph `G^T`.
- Using the order of vertices in `dfsList`, make repeated calls to `dfs()` for vertices in `G^T`. The list returned by each call is a strongly connected component of `G`.

Graph `G` and `G^T`
Running Time of strongComponents()

- Recall that the depth-first search has running time $O(V+E)$, and the computation for $G^T$ is also $O(V+E)$. It follows that the running time for the algorithm to compute the strong components is $O(V+E)$.

dfsList: [A, B, C, E, D, G, F]

Using the order of vertices in dfsList, make successive calls to dfs() for graph $G^T$

- Vertex A: dfs(A) returns the list [A, C, B] of vertices reachable from A in $G^T$.
- Vertex E: The next unvisited vertex in dfsList is E. Calling dfs(E) returns the list [E].
- Vertex D: The next unvisited vertex in dfsList is D; dfs(D) returns the list [D, F, G] whose elements form the last strongly connected component.
Strong Connectivity Algorithm

- Pick a vertex \( v \) in \( G \)
- Perform a DFS from \( v \) in \( G \)
  - If there’s a \( w \) not visited, print “no”
- Let \( G' \) be \( G \) with edges reversed
- Perform a DFS from \( v \) in \( G' \)
  - If there’s a \( w \) not visited, print “no”
  - Else, print “yes”
- Running time: \( O(n+m) \)

Possible Quiz Question

Find the strongly connected components of this graph: