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
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 an edge
- 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
- Self-loop
  - j is a self-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

- **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**
  - $C_1 = (V, b, X, g, Y, f, W, c, U, a, ...)$ is a simple cycle
  - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, ...)$ is a cycle that is not simple
Properties

Property 1
\[ \sum_v \deg(v) = 2m \]
Proof: each edge is counted twice

Property 2
In an undirected graph with no self-loops and no multiple edges
\[ m \leq n \frac{(n - 1)}{2} \]
Proof: each vertex has degree at most \((n - 1)\)

Example
- \( n = 4 \)
- \( m = 6 \)
- \( \deg(v) = 3 \)

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

What is the bound for a directed graph?
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.
Graph Operations

- Return the number, \( n \), of vertices in \( G \).
- Return the number, \( m \), of edges in \( G \).
- Return a set or list containing all \( n \) vertices in \( G \).
- Return a set or list containing all \( m \) edges in \( G \).
- Return some vertex, \( v \), in \( G \).
- Return the degree, \( \deg(v) \), of a given vertex, \( v \), in \( G \).
- Return a set or list containing all the edges incident upon a given vertex, \( v \), in \( G \).
- Return a set or list containing all the vertices adjacent to a given vertex, \( v \), in \( G \).
- Return the two end vertices of an edge, \( e \), in \( G \); if \( e \) is directed, indicate which vertex is the origin of \( e \) and which is the destination of \( e \).
- Return whether two given vertices, \( v \) and \( w \), are adjacent in \( G \).
Graph Operations, Continued

- Indicate whether a given edge, $e$, is directed in $G$.
- Return the in-degree of $v$, $\text{inDegree}(v)$.
- Return a set or list containing all the incoming (or outgoing) edges incident upon a given vertex, $v$, in $G$.
- Return a set or list containing all the vertices adjacent to a given vertex, $v$, along incoming (or outgoing) edges in $G$.

- Insert a new directed (or undirected) edge, $e$, between two given vertices, $v$ and $w$, in $G$.
- Insert a new (isolated) vertex, $v$, in $G$.
- Remove a given edge, $e$, from $G$.
- Remove a given vertex, $v$, and all its incident edges from $G$. 

Biconnectivity
Edge List Structure

- **Vertex object**
  - element
  - reference to position in vertex sequence

- **Edge object**
  - element
  - origin vertex object
  - destination vertex object
  - reference to position in edge sequence

- **Vertex sequence**
  - sequence of vertex objects

- **Edge sequence**
  - sequence of edge objects
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 nonadjacent vertices
- The “old fashioned” version just has 0 for no edge and 1 for edge
## Performance

(All bounds are big-oh running times, except for “Space”)

- $n$ vertices, $m$ edges
- no parallel edges
- no self-loops

<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>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>
Depth-First Search
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 $G$
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.

Connected graph

Non connected graph with two connected components
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 is to graphs what Euler tour is to binary trees
DFS Algorithm from a Vertex

Algorithm DFS$(G, v)$:

**Input:** A graph $G$ and a vertex $v$ in $G$

**Output:** A labeling of the edges in the connected component of $v$ as discovery edges and back edges, and the vertices in the connected component of $v$ as explored

Label $v$ as explored

for each edge, $e$, that is incident to $v$ in $G$ do

if $e$ is unexplored then

Let $w$ be the end vertex of $e$ opposite from $v$

if $w$ is unexplored then

Label $e$ as a discovery edge

DFS$(G, w)$

else

Label $e$ as a back edge
Example

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

\( DFS(G, v) \) visits all the vertices and edges in the connected component of \( v \)

Property 2

The discovery edges labeled by \( DFS(G, v) \) form a spanning tree of the connected component of \( v \)
The General DFS Algorithm

- Perform a DFS from each unexplored vertex:

```
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 v 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 is labeled twice
  - once as UNEXPLORED
  - once as DISCOVERY or BACK
- Method incidentEdges is called once for each vertex
- DFS runs in $O(n + m)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\Sigma_v \deg(v) = 2m$
Path Finding (not in book)

- We can specialize the DFS algorithm to find a path between two given vertices $u$ and $z$ using the template method pattern.
- We call $DFS(G, u)$ with $u$ as the start vertex.
- We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
- As soon as destination vertex $z$ is encountered, we return the path as the contents of the stack.

```
Algorithm pathDFS(G, v, z)
setLabel(v, VISITED)
S.push(v)
if v = z
    return S.elements()
for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
        w ← opposite(v, e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            S.push(e)
            pathDFS(G, w, z)
            S.pop(e)
else
    setLabel(e, BACK)
S.pop(v)
```
Cycle Finding (not in book)

- We can specialize the DFS algorithm to find a simple cycle using the template method pattern.
- We use a stack $S$ to keep track of the path between the start vertex and the current vertex.
- As soon as a back edge $(v, w)$ is encountered, we return the cycle as the portion of the stack from the top to vertex $w$.

Algorithm `cycleDFS(G, v, z)`

```
setLabel(v, VISITED)
S.push(v)
for all $e \in G.incidentEdges(v)$
    if getLabel(e) = UNEXPLORED
        $w \leftarrow$ opposite(v,e)
        S.push(e)
        if getLabel(w) = UNEXPLORED
            setLabel(e, DISCOVERY)
            pathDFS(G, w, z)
        end
    end
else
    $T \leftarrow$ new empty stack
    repeat
        $o \leftarrow S.pop()$
        T.push(o)
    until $o = w$
return T.elements()
S.pop(v)
```
Breadth-First Search

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
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

$i ← i + 1$
Example

- unexplored vertex
- visited vertex
- unexplored edge
- discovery edge
- cross edge

Biconnectivity
Example (cont.)

Biconnectivity
Example (cont.)

Biconnectivity
Properties

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

Property 1
\( BFS(G, s) \) visits all the vertices and edges of \( G_s \)

Property 2
The discovery edges labeled by \( BFS(G, s) \) 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
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>

**Applications**

- **DFS**
  - Spanning forest, connected components, paths, cycles
  - Shortest paths
  - Biconnected components

- **BFS**
  - Spanning forest, connected components, paths, cycles
  - Shortest paths
  - Biconnected components

**Diagrams**

- **DFS**
  - A
  - B
  - C
  - D
  - E
  - F

- **BFS**
  - A
  - B
  - C
  - D
  - E
  - F
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
Directed Graphs
Digraphs

- A digraph is a graph whose edges are all directed
  - Short for “directed graph”
- 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

![Digraph Diagram]

- cs21
- cs22
- cs46
- cs51
- cs53
- cs52
- cs131
- cs141
- cs121
- cs151
- cs161
- cs171
- The good life

Biconnectivity
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 edges
  - back edges
  - forward edges
  - cross edges
- A directed DFS starting at a vertex $s$ determines the vertices reachable from $s$. 
The Directed DFS Algorithm

Algorithm DirectedDFS$(G, v)$:

Label $v$ as active  // Every vertex is initially unexplored
for each outgoing edge, $e$, that is incident to $v$ in $G$ do
    if $e$ is unexplored then
        Let $w$ be the destination vertex for $e$
        if $w$ is unexplored and not active then
            Label $e$ as a discovery edge
            DirectedDFS$(G, w)$
        else if $w$ is active then
            Label $e$ as a back edge
        else
            Label $e$ as a forward/cross edge
    Label $v$ as explored
Reachability

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

- Each vertex can reach all other vertices
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: \( O(n+m) \)
Strongly Connected Components

- Maximal subgraphs such that each vertex can reach all other vertices in the subgraph
- Can also be done in $O(n+m)$ time using DFS, but is more complicated (similar to biconnectivity).

\[ \{ a, c, g \} \]
\[ \{ f, d, e, b \} \]
Transitive Closure

- Given a digraph $G$, the transitive closure of $G$ is the digraph $G^*$ such that:
  - $G^*$ has the same vertices as $G$.
  - If $G$ has a directed path from $u$ to $v$ ($u \neq v$), $G^*$ has a directed edge from $u$ to $v$.

- The transitive closure provides reachability information about a digraph.
Computing the Transitive Closure

- We can perform DFS starting at each vertex
  - $O(n(n+m))$

If there's a way to get from A to B and from B to C, then there's a way to get from A to C.

Alternatively ... Use dynamic programming: The Floyd-Warshall Algorithm
Floyd-Warshall
Transitive Closure

- Idea #1: Number the vertices 1, 2, ..., n.
- Idea #2: Consider paths that use only vertices numbered 1, 2, ..., k, as intermediate vertices:

Uses only vertices numbered 1, ..., k
(add this edge if it’s not already in)

Uses only vertices numbered 1, ..., k-1

Uses only vertices numbered 1, ..., k-1
Floyd-Warshall’s Algorithm: High-Level View

- Number vertices \( v_1, \ldots, v_n \)
- Compute digraphs \( G_0, \ldots, G_n \)
  - \( G_0 = G \)
  - \( G_k \) has directed edge \((v_i, v_j)\) if \( G \) has a directed path from \( v_i \) to \( v_j \) with intermediate vertices in \( \{v_1, \ldots, v_k\} \)
- We have that \( G_n = G^* \)
- In phase \( k \), digraph \( G_k \) is computed from \( G_{k-1} \)
- Running time: \( O(n^3) \), assuming areAdjacent is \( O(1) \) (e.g., adjacency matrix)
The Floyd-Warshall Algorithm

Algorithm FloydWarshall($\tilde{G}$):

Input: A digraph $\tilde{G}$ with $n$ vertices
Output: The transitive closure $\tilde{G}^*$ of $\tilde{G}$

Let $v_1, v_2, \ldots, v_n$ be an arbitrary numbering of the vertices of $\tilde{G}$
$\tilde{G}_0 \leftarrow \tilde{G}$

for $k \leftarrow 1$ to $n$ do

$\tilde{G}_k \leftarrow \tilde{G}_{k-1}$

for $i \leftarrow 1$ to $n$, $i \neq k$ do

for $j \leftarrow 1$ to $n$, $j \neq i, k$ do

if both edges $(v_i, v_k)$ and $(v_k, v_j)$ are in $\tilde{G}_{k-1}$ then

if $\tilde{G}_k$ does not contain directed edge $(v_i, v_j)$ then

add directed edge $(v_i, v_j)$ to $\tilde{G}_k$

return $\tilde{G}_n$

- The running time is clearly $O(n^3)$.
Floyd-Warshall Example
Floyd-Warshall, Iteration 1

The picture can't be displayed.
Floyd-Warshall, Iteration 2

Biconnectivity
Floyd-Warshall, Iteration 3
Floyd-Warshall, Iteration 4

The picture can't be displayed.
Floyd-Warshall, Iteration 5
Floyd-Warshall, Iteration 6

Biconnectivity
Floyd-Warshall, Conclusion
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 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

Biconnectivity
Algorithm for Topological Sorting

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

Algorithm TopologicalSort\((G)\)

\[
H \leftarrow G \quad \text{// Temporary copy of } G
\]

\[
n \leftarrow G.\text{numVertices}()
\]

while \(H\) is not empty do

Let \(v\) be a vertex with no outgoing edges

Label \(v \leftarrow n\)

\[
n \leftarrow n - 1
\]

Remove \(v\) from \(H\)

- Running time: \(O(n + m)\)
Implementation with DFS

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

**Algorithm** \textit{topologicalDFS}(G, v)

\textbf{Input} graph $G$ and a start vertex $v$ of $G$

\textbf{Output} labeling of the vertices of $G$

in the connected component of $v$

\begin{itemize}
  \item \texttt{setLabel}(v, \texttt{VISITED})
  \item for all $e \in G$.outEdges$(v)$
    \{ outgoing edges \}
    \begin{itemize}
      \item \texttt{w} $\leftarrow$ \texttt{opposite}(v,e)
      \item if \texttt{getLabel}(w) = \texttt{UNEXPLORERED}
        \{ e is a discovery edge \}
        \begin{itemize}
          \item \texttt{topologicalDFS}(G, w)
        \end{itemize}
      \else
        \{ e is a forward or cross edge \}
        \end{itemize}
    \end{itemize}
  \end{itemize}

Label $v$ with topological number $n$

$n \leftarrow n - 1$
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example
Topological Sorting Example

The diagram illustrates a directed graph used to demonstrate the concept of topological sorting.
Topological Sorting Example
Topological Sorting Example

Biconnectivity
Topological Sorting Example
Topological Sorting Example

![Graph Illustration](image-url)
Topological Sorting Example

Diagram of a directed graph with nodes labeled from 1 to 9.
Biconnected Components

Biconnectivity
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.
Separation Edges and Vertices

- **Definitions**
  - Let $G$ be a connected graph
  - A separation edge of $G$ is an edge whose removal disconnects $G$
  - A separation vertex of $G$ is a vertex whose removal disconnects $G$

- **Applications**
  - Separation edges and vertices represent single points of failure in a network and are critical to the operation of the network

- **Example**
  - DFW, LGA and LAX are separation vertices
  - (DFW,LAX) is a separation edge
Biconnected Graph

- Equivalent definitions of a biconnected graph $G$
  - Graph $G$ has no separation edges and no separation vertices
  - For any two vertices $u$ and $v$ of $G$, there are two disjoint simple paths between $u$ and $v$ (i.e., two simple paths between $u$ and $v$ that share no other vertices or edges)
  - For any two vertices $u$ and $v$ of $G$, there is a simple cycle containing $u$ and $v$

- Example

```
ORD
PVD
MIA
```

```
HNL
SFO
LAX
ORD
LGA
DFW
MIA
```
Biconnected Components

- Biconnected component of a graph $G$
  - A maximal biconnected subgraph of $G$, or
  - A subgraph consisting of a separation edge of $G$ and its end vertices
- Interaction of biconnected components
  - An edge belongs to exactly one biconnected component
  - A nonseparation vertex belongs to exactly one biconnected component
  - A separation vertex belongs to two or more biconnected components
- Example of a graph with four biconnected components
Equivalence Classes

- Given a set $S$, a relation $R$ on $S$ is a set of ordered pairs of elements of $S$, i.e., $R$ is a subset of $S \times S$.
- An equivalence relation $R$ on $S$ satisfies the following properties:
  - Reflexive: $(x,x) \in R$
  - Symmetric: $(x,y) \in R \Rightarrow (y,x) \in R$
  - Transitive: $(x,y) \in R \land (y,z) \in R \Rightarrow (x,z) \in R$
- An equivalence relation $R$ on $S$ induces a partition of the elements of $S$ into equivalence classes.
- Example (connectivity relation among the vertices of a graph):
  - Let $V$ be the set of vertices of a graph $G$.
  - Define the relation $C = \{(v,w) \in V \times V \mid \text{such that } G \text{ has a path from } v \text{ to } w\}$.
  - Relation $C$ is an equivalence relation.
  - The equivalence classes of relation $C$ are the vertices in each connected component of graph $G$. 
Link Relation

- Edges $e$ and $f$ of connected graph $G$ are linked if
  - $e = f$, or
  - $G$ has a simple cycle containing $e$ and $f$

Theorem:
The link relation on the edges of a graph is an equivalence relation

Proof Sketch:
- The reflexive and symmetric properties follow from the definition
- For the transitive property, consider two simple cycles sharing an edge

Equivalence classes of linked edges:
\[ \{a\} \quad \{b, c, d, e, f\} \quad \{g, i, j\} \]
Link Components

- The link components of a connected graph $G$ are the equivalence classes of edges with respect to the link relation.
- A biconnected component of $G$ is the subgraph of $G$ induced by an equivalence class of linked edges.
- A separation edge is a single-element equivalence class of linked edges.
- A separation vertex has incident edges in at least two distinct equivalence classes of linked edges.
Auxiliary Graph

- Auxiliary graph $B$ for a connected graph $G$
  - Associated with a DFS traversal of $G$
  - The vertices of $B$ are the edges of $G$
  - For each back edge $e$ of $G$, $B$ has edges $(e, f_1), (e, f_2), ..., (e, f_k)$, where $f_1, f_2, ..., f_k$ are the discovery edges of $G$ that form a simple cycle with $e$
  - Its connected components correspond to the link components of $G$
Auxiliary Graph (cont.)

- In the worst case, the number of edges of the auxiliary graph is proportional to $nm$
An $O(nm)$-Time Algorithm

- Lemma: The connected components of the auxiliary graph $B$ correspond to the link components of the graph $G$ that induced $B$.
- This lemma yields the following $O(nm)$-time algorithm for computing all the link components of a graph $G$ with $n$ vertices and $m$ edges:

1. Perform a DFS traversal $T$ on $G$.
2. Compute the auxiliary graph $B$ by identifying the cycles of $G$ induced by each back edge with respect to $T$.
3. Compute the connected components of $B$, for example, by performing a DFS traversal of the auxiliary graph $B$.
4. For each connected component of $B$, output the vertices of $B$ (which are edges of $G$) as a link component of $G$. 

Biconnectivity
A Linear-Time Algorithm

**Algorithm** LinkComponents($G$):

*Input:* A connected graph $G$

*Output:* The link components of $G$

Let $F$ be an initially empty auxiliary graph. Perform a DFS traversal of $G$ starting at an arbitrary vertex $s$. Add each DFS discovery edge $f$ as a vertex in $F$ and mark $f$ “unlinked.” For each vertex $v$ of $G$, let $p(v)$ be the parent of $v$ in the DFS spanning tree. For each vertex $v$, in increasing rank order as visited in the DFS traversal do:

for each back edge $e = (u, v)$ with destination $v$ do:

Add $e$ as a vertex of the graph $F$.

// March up from $u$ to $s$ adding edges to $F$ only as necessary.

while $u \neq v$ do:

Let $f$ be the vertex in $F$ corresponding to the discovery edge $(u, p(u))$.

Add the edge $(e, f)$ to $F$.

if $f$ is marked “unlinked” then

Mark $f$ as “linked.”

$u \leftarrow p(u)$

else

$u \leftarrow v$  // shortcut to the end of the while loop

Compute the connected components of the graph $F$. 

Biconnectivity
Analysis with the Proxy Graph, F

- Proxy graph $F$ for a connected graph $G$
  - Spanning forest of the auxiliary graph $B$
  - Has $m$ vertices and $O(m)$ edges
  - Can be constructed in $O(n + m)$ time
  - Its connected components (trees) correspond to the link components of $G$

- Given a graph $G$ with $n$ vertices and $m$ edges, we can compute the following in $O(n + m)$ time:
  - The biconnected components of $G$
  - The separation vertices of $G$
  - The separation edges of $G$