Connectivity and Biconnectivity
Connected Components

**Connected Graph:** any two vertices connected by a path

connected

not connected

**Connected Component:** maximal connected subgraph of a graph
Equivalence Relations

A relation on a set S is a set R of ordered pairs of elements of S defined by some property.

Example:
- S = \{1, 2, 3, 4\}
- R = \{(i,j) \in S \times S \text{ such that } i < j\} = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}

An equivalence relation is a relation with the following properties:
- (x,x) \in R, \quad \forall \ x \in S \quad \text{(reflexive)}
- (x,y) \in R \implies (y,x) \in R \quad \text{(symmetric)}
- (x,y), (y,z) \in R \implies (x,z) \in R \quad \text{(transitive)}

The relation C on the set of vertices of a graph:
- (u,v) \in C \iff u \text{ and } v \text{ are in the same connected component}

is an equivalence relation.
**DFS on a Disconnected Graph**

After dfs(1) terminates:

\[
\begin{array}{cccccccc}
    k & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
val[k] & 1 & 4 & 0 & 2 & 0 & 0 & 3 \\
\end{array}
\]
DFS of a Disconnected Graph

- Recursive **DFS** procedure visits all vertices of a connected component.
- A **for** loop is added to visit all the graph

```plaintext
for all k from 1 to N
  if val[k] = 0
    dfs(k)
```

![Graph Diagram]
Representing Connected Components

Array comp [1..N]
comp[k] = i if vertex k is in i-th connected component

vertex k  1  2  3  4  5  6  7  8
comp[k]  1  1  2  3  2  3  2  1
New **DFS Algorithm**

**Inside DFS:**

replace

\[
id = id + 1;
\]

\[
val[k] = id;
\]

with

\[
comp[k] = id;
\]

**Outside DFS:**

\[
\text{for all } k \text{ from } 1 \text{ to } N \\
\text{if } comp[k] = 0 \\
id = id + 1; \\
dfs(k);
\]

\[
\text{for each vertex} \\
\text{if not in } comp \\
\text{new component}
\]
DFS Algorithm for Connected Components

Pseudocoded

```java
public void dfs(int k) {
    comp[k] = vertex.id;
    vertex = adj[k];

    Vertex vertex
    while (vertex != null) {
        if (val[vertex.num] == 0)
            dfs(vertex.num);
        vertex = vertex.next;
    }

    // Other code...
}
```

for all k from 1 to N
    if (comp[k] == 0)
        id = id + 1;
    dfs(k);

TIME COMPLEXITY: O (N + M)
Cutvertices

Cutvertex (separation vertex): its removal disconnects the graph

If the Chicago airport is closed, then there is no way to get from Providence to beautiful Denver, Colorado!

• Cutvertex: ORD
Biconnectivity

Biconnected graph: has no cutvertices

New flights: LGA-ATL and DFW-LAX make the graph biconnected.
Properties of Biconnected Graphs

- There are two disjoint paths between any two vertices.
- There is a simple cycle through any two vertices.

By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties.
Biconnected Components

Biconnected component (block): maximal biconnected subgraph

Biconnected components are edge-disjoint but share cutvertices.
Finding Cutvertices: Brute Force Algorithm

for each vertex v
    remove v;
    test resulting graph for connectivity;
    put back v;

Time Complexity:
• $N$ connectivity tests
• each taking time $O(N + M)$

Total time:
• $O(N^2 + NM)$
DFS Numbering

We recall that depth-first-search partitions the edges into tree edges and back edges:

- $(u, v)$ tree edge $\iff \text{val}[u] < \text{val}[v]$
- $(u, v)$ back edge $\iff \text{val}[u] > \text{val}[v]$

We shall characterize cutvertices using the DFS numbering and two properties ...
Root Property

The root of the DFS tree is a cutvertex if it has two or more outgoing tree edges.

- no cross/horizontal edges
- must retrace back up
- stays within subtree to root, must go through root to other subtree
Complicated Property

A vertex $v$ which is not the root of the DFS tree is a cutvertex if $v$ has a child $w$ such that no back edge starting in the subtree of $w$ reaches an ancestor of $v$. 

\[
\begin{align*}
\text{root} & \quad \text{root} \\
& \quad \text{root}
\end{align*}
\]
Definitions

- **low(v)**: vertex with the lowest val (i.e., “highest” in the DFS tree) reachable from v by using a directed path that uses at most one back edge

- **Min(v) = val(low(v))**
DFS Algorithm for Finding Cutvertices

1. Perform DFS on the graph
2. Test if root of DFS tree has two or more tree edges (root property)
3. For each other vertex \( v \), test if there is a tree edge \( (v,w) \) such that Min\( (w) \) ≥ val\( [v] \) (complicated property)

\( \text{Min}(v) = \text{val}(	ext{low}(v)) \) is the minimum of:

- \( \text{val}[v] \)
- minimum of Min\( (w) \) for all tree edges \( (v,w) \)
- minimum of val\( [z] \) for all back edges \( (v,z) \)

Implement this recursively and you are done!!!!
Finding the Biconnected Components

- DFS visits the vertices and edges of each biconnected component consecutively
- Use a stack to keep track of the biconnected component currently being traversed