1 Basic Depth-First Search

- Algorithm

procedure Search($G = (V, E)$)

// Assume $V = \{1, 2, \ldots, n\}$

// global array $visited[1..n]$ //

$visited[1..n] \leftarrow 0$;

for $i \leftarrow 1$ to $n$

if $visited[i] = 0$ then call $dfs(i)$

procedure $dfs(v)$

$visited[v] \leftarrow 1$;

for each node $w$ such that $(v, w) \in E$ do

if $visited[w] = 0$ then call $dfs(w)$

- Questions

  - How to implement the for-loop (i) if an adjacency matrix is used to represent the graph and (ii) if adjacency lists are used?
  - How many times is $dfs$ called in all?
  - How many times is “$if visited[:] = 0$” executed in all?
  - What’s the over-all time complexity of the command “$for$ each node $w$ such that $(v, w) \in E$”

- Time complexity

  - Using adjacency matrix: $O(n^2)$
  - Using adjacency lists: $O(|V| + |E|)$
Definitions

- Depth first tree/forest, denoted as $G_{\pi}$
- Tree edges: those edges in $G_{\pi}$
- Forward edges: those non-tree edges $(u, v)$ connecting a vertex $u$ to a descendant $v$.
- Back edges: those edges $(u, v)$ connecting a vertex $u$ to an ancestor $v$.
- Cross edges: all other edges.
- If $G$ is undirected, then there is no distinction between forward edges and back edges. Just call them back edges.
2 Depth-First Search Revisited

procedure $\text{Search}(G = (V,E))$

// Assume $V = \{1,2,\ldots,n\}$ //

$\text{time} \leftarrow 0$;

d[1..n] $\leftarrow 0$; /* $d$ stands for discovery time */

for $i \leftarrow 1$ to $n$

\hspace{1em} if $d[i] = 0$ then call $dfs(i)$

procedure $dfs(v)$

\hspace{1em} $d[v] \leftarrow time \leftarrow time + 1$;

\hspace{1em} for each node $w$ such that $(v,w) \in E$ do

\hspace{2em} if $d[w] = 0$ then call $dfs(w)$;

\hspace{1em} $f[v] \leftarrow time \leftarrow time + 1$ /* $f$ stands for finishing time */
3 Topological Sort

• Problem: given a directed acyclic graph $G = (V, E)$, obtain a linear ordering of the vertices such that for every edge $(u, v) \in E$, $u$ is ahead of $v$ in the ordering.

• Solution:
  
  – Use depth-first search, with an initially empty list $L$.
  – At the end of procedure $dfs(v)$, insert $v$ to the front of $L$.
  – $L$ gives a topological sort of the vertices.

• Observation: the list of nodes in the descending order of finishing times yields a topological sort.
4 Strongly Connected Components

- A directed graph is *strongly connected* if for every two nodes \( u \) and \( v \) there is a path from \( u \) to \( v \) and one from \( v \) to \( u \).

- Decide if a graph \( G \) is strongly connected:
  - \( G \) is strongly connected iff (i) every node is reachable from node 1 and (ii) node 1 is reachable from every node.
  - The two conditions can be checked by applying \( dfs(1) \) to \( G \) and to \( G^T \), where \( G^T \) is the graph obtained from \( G \) by reversing the edges.

- A subgraph \( G' \) of a directed graph \( G \) is said to be a *strongly connected component* of \( G \) if \( G' \) is strongly connected and is not contained in any other strongly connected subgraph.

- An interesting problem is to find all strongly connected components of a directed graph.

- Each node belongs in exactly one component. So, we identify each component by its vertices.

- The component containing \( v \) equals

\[
\{ dfs(v) \text{ on } G \} \cap \{ dfs(v) \text{ on } G^T \},
\]

where \( \{ dfs(v) \text{ on } G \} \) denotes the set of all vertices visited during \( dfs(v) \text{ on } G \).
• Ideas:

− If $C$ is a strongly connected component, define

$$f(C) = \max\{f(x) : x \in C\}.$$ 

− Let $C, C'$ be two distinct strongly connected components. If there is an edge in $G$ from $C$ to $C'$, then $f(C) > f(C')$. (In $G$, edges between two strongly connected components go from the component with higher finishing time to the component with lower finishing time.)

− Let $C, C'$ be two distinct strongly connected components. If there is an edge in $G^T$ from $C'$ to $C$, then $f(C) > f(C')$. (In $G^T$, edges between two strongly connected components go from the component with lower finishing time to the component with higher finishing time.)

• Algorithm:

1. Apply depth-first search to $G$ and compute $f[u]$ for each node.
2. Compute $G^T$.
3. Apply the basic depth-first search to $G^T$:

   $$\text{visited}[1..n] \leftarrow 0$$
   
   for each vertex $u$ in decreasing order of $f[u]$ do
   
   if $\text{visited}[u] = 0$ then call $\text{dfs}(u)$

4. The vertices on each tree in the depth-first forest of Step 3 form a strongly connected component.
5 Articulation Points and Biconnected Components

5.1 Definitions

- Let $G$ be a connected, undirected graph.

- An \textit{articulation point} of $G$ is a vertex whose removal will disconnect $G$.

- A \textit{bridge} of $G$ is an edge whose removal will disconnect $G$.

- **Definition:** A (connected) graph is \textit{biconnected} if it contains no articulation points.

- A \textit{biconnected component} of $G$ is a maximal biconnected subgraph.

- Each edge belongs to exactly one biconnected component.
5.2 Identifying All Articulation Points

- Let $G_\pi$ be any depth-first tree of $G$.

- An edge in $G$ is a back edge iff it is not in $G_\pi$.

- The root of $G_\pi$ is an articulation of $G$ iff it has at least two children.

- A non-root vertex $v$ in $G_\pi$ is an articulation point of $G$ iff $v$ has a child $w$ in $G_\pi$ such that no vertex in subtree($w$) is connected to a proper ancestor of $v$ by a back edge. (subtree($w$) denotes the subtree rooted at $w$ in $G_\pi$.)

- Define

  \[
  low[w] = \min \left\{ \frac{d[w]}{d[x]} : x \text{ is joined to some vertex in } \text{subtree}(w) \text{ by a back edge} \right\}
  \]

- A non-root vertex $v$ in $G_\pi$ is an articulation point of $G$ iff $v$ has a child $w$ such that $low[w] \geq d[v]$. 

• Note that

\[
low[v] = \min \left\{ \begin{array}{l}
d[v] \\
d[w] : w\text{ is connected to } v \text{ by a back edge} \\
low[w] : w\text{ is a child of } v
\end{array} \right. 
\]

• Computing \(low[v]\) for each vertex \(v\):

**procedure** \(Art(v, u)\)

/* visit \(v\) from \(u\) */

\(low[v] \leftarrow d[v] \leftarrow time \leftarrow time + 1;\)

for each vertex \(w \neq u\) such that \((v, w) \in E\) do

if \(d[w] = 0\) then
    call \(Art(w, v)\)
    \(low[v] \leftarrow \min\{low[v], low[w]\}\)
else
    \(low[v] \leftarrow \min\{low[v], d[w]\}\)
endif
endfor

• Initial call: \(Art(1, 0)\).
- **Problem:** Print all articulation points.

function Art(v, u)

- /* visit v from u */
- low[v] ← d[v] ← time ← time + 1;
- for each vertex w ≠ u such that (v, w) ∈ E do
  - if d[w] = 0 then
    - call Art(w, v)
    - low[v] ← min{low[v], low[w]}
  - if (d[v] = 1 and d[w] ≠ 2) then
    - print v is an articulation point
  - if (d[v] ≠ 1 and low[w] ≥ d[v]) then
    - print v is an articulation point
  - else
    - low[v] ← min{low[v], d[w]}
  - endif
  endfor
endfor


• **Problem:** Identify all biconnected components.

```plaintext
procedure Art(v, u)
    /* visit v from u */
    low[v] ← d[v] ← time ← time + 1;
    for each vertex w ≠ u such that (v, w) ∈ E do
        if d[w] < d[v] then add (v, w) to Stack
        if d[w] = 0 then
            call Art(w, v)
            low[v] ← min{low[v], low[w]}
        if low[w] ≥ d[v] then
            Pop off all edges from Stack till edge (v, w)
            // these edges form a biconnected component //
        else
            low[v] ← min{low[v], d[w]}
        endif
    endfor
```