Graph Algorithms

Plan:
- DFS
- Topological Sorting
- Classification of Edges
- Biconnected Components

Graphs Traversal
Visiting all vertices in a systematic order.

```plaintext
for all v in V do visited[v] = false
for all v in V do if !visited[v] traversal(v)
traversal(v) {
    visited[v] = true
    for all w in adj(v)
        do if !visited[w] traversal(w)
}
```

O(V + E)

Graphs Traversals
- Depth-First Search (DFS)
- Breadth-First Search (BFS)

DFS uses a stack for backtracking.
BFS uses a queue for bookkeeping

Properties of DFS

Property 1
- DFS visits all the vertices in the connected component

Property 2
- The discovery edges labeled by DFS form a spanning tree of the connected component

Applications of DFS
- Determine the connected components of a graph
- Find cycles in a graph
- Determine if a graph is bipartite.
- Topologically sort in a directed graph
- Find the biconnected components
Topological Sorting

Find an ordering of the vertices such that all edges go forward in the ordering.

It's easy to see that such an ordering exists. Find a vertex with zero in-degree. Print it, delete it from the graph, and repeat.

Complexity?

PQ wrt in-degrees. O(E log V)

Topological Sorting with DFS

DFS (v) {
    visited[v] = true
    for all w in adj(v)
        do if !visited[w]
            DFS (w);
    print(v);
}

Do DFS; Reverse the order;

Complexity?

O(E + V)

Classification of Edges with DFS

Tree edge
Back edge
Forward edge
Cross edge

Classification of Edges

Tree edges - are edges in the DFS
Forward edges - edges (u,v) connecting u to a descendant v in a depth-first tree
Back edges - edges (u,v) connecting u to an ancestor v in a depth-first tree
Cross edges - all other edges

DAG

Theorem.
A directed graph is acyclic iff a DFS yields no back edges.

Proof.
=>) by contrapositive
If there is a back edge, the graph is surely cyclic.
Theorem.
A directed graph is acyclic iff a DFS yields no back edges.

Proof.
$\Rightarrow$ Suppose there is a cycle.
Let $v$ be the first vertex discovered in the cycle. Let $(u, v)$ be the preceding edge in this cycle. When we push $v$ on the stack, no any vertices on the cycle were discovered yet. Thus, vertex $u$ becomes a descendent of $v$ in DFS. Therefore, $(u, v)$ is a back edge.

for all $v \in V$ do $\text{num}[v] = 0$, $\text{stack}[v] = \text{false}$
for all $v \in V$ do if $\text{num}[v] = 0$ DFS($v$)

$k = 0$;

DFS($v$) {
    $k++$, $\text{num}[v] = k$, $\text{stack}[v] = \text{true}$
    for all $w \in \text{adj}(v)$ do
        if $\text{num}[w] = 0$ DFS($w$) 
            tree edge
        else if $\text{num}[w] > \text{num}[v]$ 
            forward edge
        else if $\text{stack}[w]$ 
            back edge
        else 
            cross edge
    $\text{stack}[v] = \text{false}$
}

Biconnectivity

In many applications it's not enough to know that a graph is connected, but "how well" it's connected.

Articulation points

A vertex is an articulation point if its removal (with edges) disconnect a graph.

A connected graph is biconnected if it has no articulation points.

If a graph is not biconnected, we define the biconnected components

Biconnected Components

If a graph is not biconnected, we define the biconnected components

Biconnected graphs are of great interest in communication and transportation networks

Find articulation points

Fred Hacker's algorithm:
Delete a vertex
Run DFS to see if a graph is connected
Choose a new vertex. Repeat.

Complexity: $O(V (V+E))$
**Biconnected Component Algorithm**

- It is based on a DFS
- We assume that $G$ is undirected and connected.
- We cannot distinguish between forward and back edges
- Also there are no cross edges (!)

**Find articulation point:**

an observation

If for some child, there is no back edge going to an ancestor of $u$, then $u$ is an articulation point.

We need to keep a track of back edges!

We keep a track of back edge that goes higher in the tree.

**Find articulation point:**

next observation

What about the root?

Can it be an articulation point?

DFS root must have two or more children

**Biconnected Component Algorithm**

- Run DFS
- When we reach a dead end, we will back up. On the way up, we will discover back edges. They will tell us how far in the tree we could have gone.
- These back edges indicate a cycle in the graph. All nodes in a cycle must be in the same component.

**Bookkeeping**

- For each vertex we will store two indexes. One is the counter of nodes we have visited so far $dfs[v]$. Second - the back index $low[v]$.
- **Definition.**

  $low[v]$ is the DFS number of the lowest numbered vertex $x$ (i.e. highest in the tree) such that there is a back edge from some descendent of $v$ to $x$.

**How to compute $low[v]$?**

- Tree edge $(u, v)$

  $$low[u] = \min( low[u], dfs[v] )$$

  Vertices $u$ and $v$ are in the same cycle.

- Back edge $(u, v)$

  $$low[u] = \min( low[u], dfs[v] )$$

  If the edge goes to a lower $dfs$ value then the previous back edge, make this the new low.
How to test for articulation point?

Using low[u] value we can test whether u is an articulation point.

If for some child, there is no back edge going to an ancestor of u, then u is an articulation point.

If there was a back edge from child v, than low[v] < dfs[u].

It follows, u is an articulation point iff it has a child v such that low[v] >= dfs[u].

The Algorithm

\[\text{low}(A) = \text{dfs}(B)\]

Remove bicomponent G/AB

All edges are on a stack

Vertex labels
dfs/low

Store edges on a stack as you run DFS

Verteex labels

dfs/low

Theorem : Let \(G = (V, E)\) be a connected, undirected graph and \(S\) be a depth-first tree of \(G\). Vertex \(x\) is an articulation point of \(G\) if and only if one of the following is true:

(1) \(x\) is the root of \(S\) and \(x\) has two or more children in \(S\).

(2) \(x\) is not the root and for some child \(s\) of \(x\), there is no back edge between any descendant of \(s\) (including \(s\) itself) and a proper ancestor of \(x\).

Proof:

\(\Rightarrow\) If \(x\) is an articulation vertex, then removing it will disconnect child \(s\) from the parent of \(x\).

\(\Leftarrow\) If there is no such \(s\), then \(x\) is not articulation point. To see this, suppose \(v_0\) is the parent and \(v_1, \ldots, v_k\) are all children. By our assumption, there exists a path from \(v\) to \(v_0\). They are in the same connected components. Removing \(x\), won't disconnect the graph.