

Depth First Search

Preamble. In addition to Breadth First Search (BFS), Depth first search (DFS) is another common graph search strategy. It turns out that this search strategy works well with some recursive algorithms for some graph problems. As an example, we will use it solve the articulation point problem.

Algorithm. Contrary to BFS, DFS recursively traces a path as far as possible before returning from the recursion to explore other unvisited neighbors. For solving the articulation point problem, we keep a global variable time and a variable $d[v]$ for each vertex v to record the time at which we first visit v . Note that this is a simplified version of the DFS algorithm we gave in class (since we are only calculating starting times and not finishing times).

DFS(G)

1. time=0;
2. For each vertex u , $\text{pred}[u] = -1$ and $\text{flag}[u] := \text{false}$.
3. For each vertex u , if $\text{flag}[u] = \text{false}$, then DFSVisit(u).

DFSVisit(u)

1. $\text{flag}[u] := \text{true}$.
2. time = time + 1 and $d[u] := \text{time}$.
3. For each neighbor v of u , if ($\text{flag}[v] = \text{false}$), then $\text{pred}[v] := u$ and DFSVisit(v).

If the graph G is connected, every vertex is marked after the vertex u is visited in the for-loop in step 3.

Running time. Let T_u denote the time that we spend at a vertex u , *excluding* the waiting time for any recursive call at the neighbors of u . Then, the running time is

$$O(n) + \sum_u T_u.$$

We have

$$\begin{aligned} \sum_u T_u &\leq \sum_u O(\text{outdeg}(v) + 1) \\ &= O(n + m). \end{aligned}$$

Hence, DFS takes $O(n + m)$ time.

Recall that m is the number of edges and n the number of vertices.

DFS tree. As in the case of BFS tree, the paths traversed by DFS form a DFS tree rooted at the source vertex. We call the edges in a DFS tree the *tree edges*. The edges of G not in the DFS tree are called *back edges*. Recall that if (u, w) is a back edge from u , then w must be on the path from u to the root.

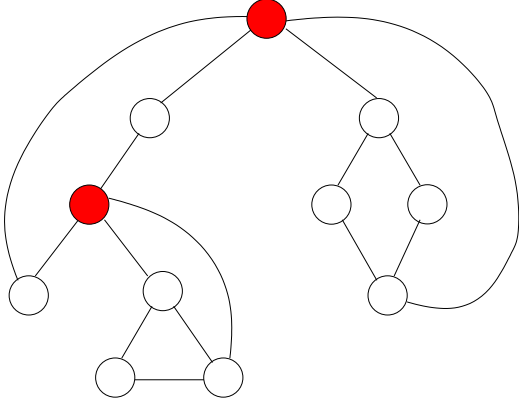
Even if we fix the source vertex, a DFS tree is not unique because its structure depends on the order in which we recursively visit the neighbors of vertices. A useful property is that the start time $d[v]$ increases strictly down a path in a DFS tree.

In particular, note that this implies that if u is a descendent of v and (u, w) is a back edge, then if $d[w] > d[v]$, w is on the path between u and v and if $d[w] < d[v]$, w is on the path between v and the root.

Articulation point. An *articulation point* of G is a vertex whose removal disconnects G . The red vertices in the example below are articulation points. How can we find articulation points quickly?

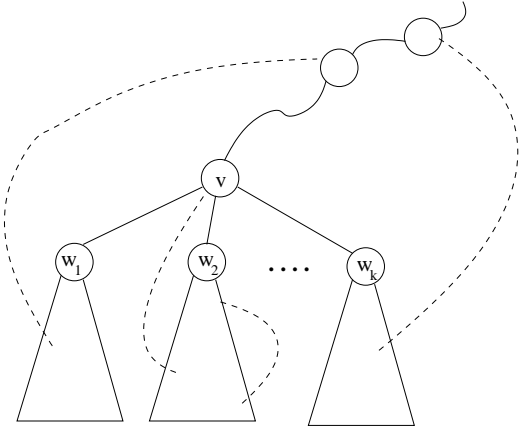
A brute-force approach is to remove each vertex from G and test run a graph search to check the connectivity of the remaining subgraph. This takes $O(n^2 + nm)$ time. We can do better by exploiting the following properties of a DFS tree.

- A leaf of a DFS tree is not an articulation point.
- The root is an articulation point iff it has two or more children.
- An internal vertex v in a DFS tree is an articulation point iff there exists a subtree T rooted



at a child of v such that no back edge connects a node in T to a proper ancestor of v .

Note that if u is a descendant of T , then u has a back edge (u, w) connecting to a proper ancestor of v iff $d[w] < d[v]$.



This means that internal vertex v is an articulation point iff it has a subtree T rooted at a child of v such that no back edge connects T to a vertex w such that $d[w] < d[v]$. This motivates the following definition.

$\text{low}[v]$ is the minimum of $d[v]$ and $d[w]$ over all back edges $\{u, w\}$ such that $u = v$ or u is a proper descendant of v .

We can rewrite this definition recursively.

$\text{low}[v]$ is the minimum of $d[v]$, $d[w]$ over all back edges $\{v, w\}$, and $\text{low}[u]$ over all children u of v .

The recursive formulation can be translated into a DFS-like algorithm to compute $\text{low}[v]$ for each vertex v . Then, we can use $\text{low}[v]$ to identify the articulation points.

$\text{Art}(v)$

1. $\text{flag}[v] := \text{true}$, $\text{time} := \text{time} + 1$, $d[v] := \text{time}$, $\text{low}[v] := d[v]$.
2. For each neighbor w of v ,
 - (a) If $\text{flag}[w] = \text{false}$, then
 - i. $\text{pred}[w] := v$.
 - ii. $\text{Art}(w)$.
 - iii. If $\text{pred}[v] = -1$ and w is v 's second child, output v .
 - iv. Otherwise, if $\text{low}[w] \geq d[v]$, output v .
 - v. $\text{low}[v] := \min(\text{low}[v], \text{low}[w])$.
 - (b) Otherwise, if $w \neq \text{pred}[v]$, then $\text{low}[v] := \min(\text{low}[v], d[w])$.

The overall running time is still $O(n + m)$ because only $O(1)$ extra instructions are added into the DFS pseudocode and each takes $O(1)$ time.

Note: This document was written by M. J. Golin, revised from an original by S.W. Cheng, for COMP3711H, HKUST.