CS 341: Algorithms

Lecture 11: Depth-first search

Éric Schost

based on lecture notes by many other CS341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2024

Depth-first search

Going depth-first

The idea:

- travel as deep as possible, as long as you can
- when you can't go further, backtrack.

DFS implementations are based on stacks, either **implicitly** (recursion) or **explicitly** (as with queues for BFS).

Recursive algorithm

DFS(G)
$G = (V, E)$: a graph with n vertices, given by adjacency lists
1. let visited be an array of size n, with all entries set to **false**
2. **for all** v in V
3. **if** visited$[v]$ is **false**
4. **explore**(v)

explore(v)
1. visited$[v]$ = **true**
2. **for all** w neighbour of v **do**
3. **if** visited$[w]$ = **false**
4. **explore**(w)

Remark: can add parent array as in BFS

Basic properties

Claim (“white path lemma”)

When we start exploring v, any w that can be connected to v by a path of **unvisited** vertices will be visited before **explore**(v) is finished.

Proof. Same as for **BFS**(s).

Claim

If w is visited during **explore**(v), there is a path $\boldsymbol{v} \rightsquigarrow \boldsymbol{w}$.

Proof. Same as for **BFS**(s).

Consequences

Shortest paths: no

Runtime: still $\boldsymbol{O(n+m)}$

Connected components:

- let $v_1, \ldots, v_k$ be the indices from which we enter **explore** in **DFS**
- then for all j, **explore**(v_j) visits exactly the connected component of v_j
- so **DFS** gives a partition of G into rooted trees $T_1, \ldots, T_k$ **(DFS forest)** (no common vertex, no connecting edge)

Ancestors and descendants

Definition. Suppose the DFS forest is $T_1, \ldots, T_k$ and let u, v be two vertices

- u is an **ancestor** of v iff they are on the same T_i and u is on the path root $\rightsquigarrow v$
- v is a **descendant** of u iff u is an ancestor of v
- $u = v$ is OK

Claim

All edges in G connect a vertex to one of its descendants or ancestors.

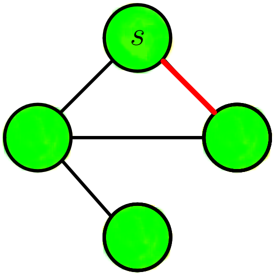
Proof. Let $\{v, w\}$ be an edge, and suppose we explore from v first.

Then when we explore from v, (v, w) is an unvisited path between v and w, so w will become a descendant of v (white path lemma)

Back edges

Definition.

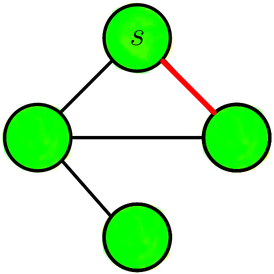
- a **back edge** is an edge in G connecting an ancestor to a descendant, which is **not** a tree edge.



Back edges

Definition.

- a **back edge** is an edge in G connecting an ancestor to a descendant, which is **not** a tree edge.



Observation

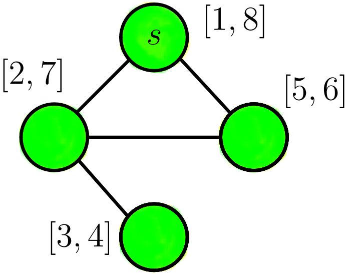
All edges are either **tree edges** or **back edges** (previous slide).

Start and finish times

Set a global variable t to 1 initially, create two arrays **start** and **finish**, and change **explore**:

```
explore(v)
    visited[v] = true
    start[v] = t
    t++
    for all w neighbour of v do
        if visited[w] = false
            explore(w)
    finish[v] = t
    t++
```

Example



Observation

time intervals are either contained in one another, or disjoint

Proof: if u starts before v, then

- either u finishes before v starts (disjoint intervals)
- or u is still on the program stack when v starts, then v finishes before u does (inclusion)

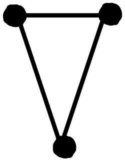
Cut vertices

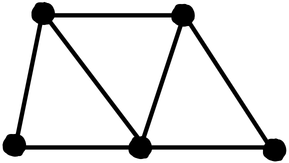
Biconnectivity

Definition: $G = (V, E)$ **biconnected** if

- G is connected
- G stays connected if we remove any vertex (and all edges that contain it)

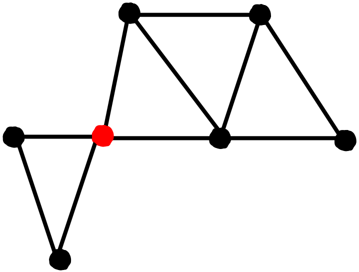
Two biconnected graphs:





Cut vertices

Definition: for G connected, a vertex v in G is a **cut vertex** if removing v (and all edges that contain it) makes G disconnected. Also called **articulation point**.

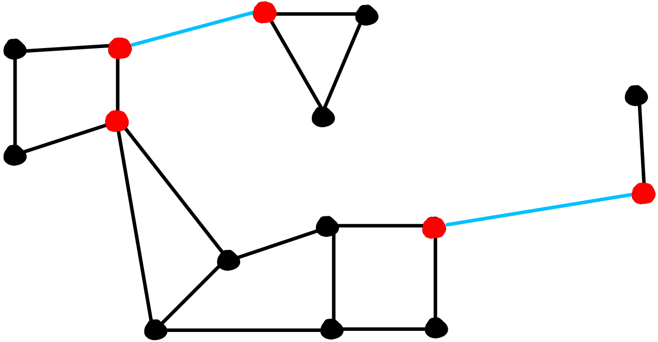


biconnected $\iff$ no cut vertex

Aside: the shape of a connected undirected graph

Call **biconnected component** a biconnected subgraph that is not contained in a larger one (two **edges** are in the same biconnected component iff there is a cycle that contains them)

Then G can be seen as a tree of alternating **biconnected components** and **cut vertices**

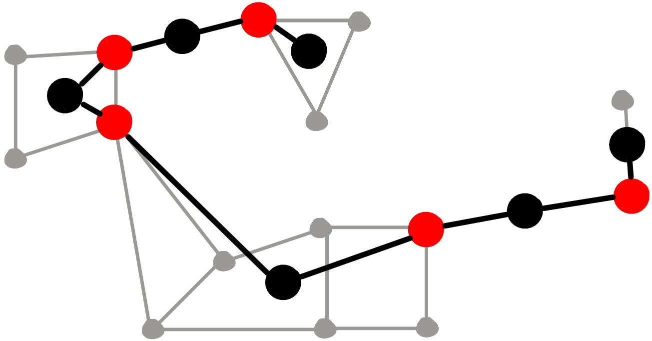


Remark: blue edges are **cut edges (bridges):** removing them makes the graph disconnected

Aside: the shape of a connected undirected graph

Call **biconnected component** a biconnected subgraph that is not contained in a larger one (two **edges** are in the same biconnected component iff there is a cycle that contains them)

Then G can be seen as a tree of alternating **biconnected components** and **cut vertices**



Finding the cut vertices (G connected)

Setup: we start from a **rooted DFS tree** T, knowing parent and level.

Warm-up

The root s is a cut vertex if and only if **it has more than one child**.

Proof.

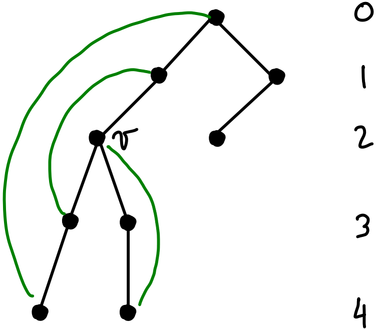
- if s has one child, removing s leaves T connected. So s not a cut vertex.
- suppose s has subtrees $S_1, \ldots, S_k$, $k > 1$.

  Key property: no edge connecting S_i to S_j for $i \neq j$. So removing s creates k connected components.

Finding the cut vertices (G connected)

Definition: for a vertex v, let

- $a(v) = \min\{\mathsf{level}[w],\ \{v, w\} \text{ edge}\}$
- $m(v) = \min\{a(w),\ w \text{ descendant of } v\}$ (v is a descendant of v)



$a(v) = 1$

$m(v) = 0$

Using the values $m(v)$

Claim

For any v (except the root), v is a cut vertex if and only if **it has a child w with $m(w) \geq \mathsf{level}[v]$**.

Proof

- Take a child w of v, let T_w be the subtree at w. Let also T_v be the subtree at v.
- If $m(w) < \mathsf{level}[v]$, then there is an edge from T_w to a vertex above v. After removing v, T_w remains connected to the root.
- If $m(w) \geq \mathsf{level}[v]$, then **all edges originating from T_w end in T_v**.

  Proof: any edge originating from a vertex x in T_w ends at a level at least $\mathsf{level}[v]$, and connects x to one of its ancestors or descendants (key property)

  So after removing v, T_w is disconnected from the root (which is still here)

Runtime

Observation:

- if v has children $\boldsymbol{w_1, \ldots, w_k}$, then $\boldsymbol{m(v) = \min\{a(v), m(w_1), \ldots, m(w_k)\}}$

Consequence:

- DFS tree in $\boldsymbol{O(m)}$
- computing $a(v)$ is $\boldsymbol{O(d_v)}$ — d_v = degree of v
- knowing all $m(w_1), \ldots, m(w_k)$, we get $m(v)$ in $\boldsymbol{O(d_v)}$
- testing the cut-vertex condition at v is $\boldsymbol{O(d_v)}$
- total $\boldsymbol{O(m)}$

Exercise

- write the pseudo-code
- find the bridges