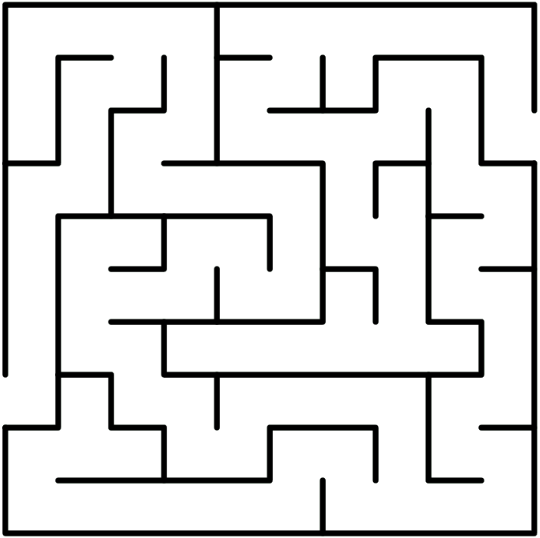
CS 341: Algorithms
Lec 06: Depth First Search

Armin Jamshidpey   Collin Roberts

Based on lecture notes by Éric Schost

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

Winter 2024

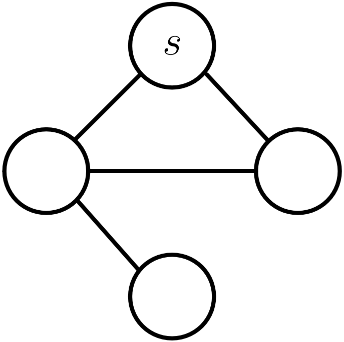


Going depth-first

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: a graph with n vertices, given by adjacency lists

```
let visited be an array of size n, with all entries set to false
for all v in G
    if visited[v] is false
        explore(v)
```

explore(v)

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

Remark: can add parent array as in BFS

The white path lemma

Claim ("white path lemma")

When we start exploring a vertex v, any w that can be connected to v by an **unvisited** path will be visited during **explore**(v).

Proof. Let $\boldsymbol{v_0 = v, \ldots, v_k = w}$ be a path $v \rightsquigarrow w$, with $v_1, \ldots, v_k$ not visited. We prove all v_i's are visited before **explore**(v) is finished.

True for $i = 0$. Suppose true for $i < k$. When we visit v_i, **explore**(v) is not finished, and v_{i+1} is one of its neighbours.

- **if** visited$[v_{i+1}]$ **is true when we reach Step 3**
  OK: it means we visited it
- **else, we will visit it just now**
  OK: it will be done before **explore**(v) is finished

In any case, by induction assumption, it happens during **explore**(v).

Another basic property

Claim

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

Proof. Same as for BFS.

Consequences

Previous properties: after we call **explore** at $v_1, \ldots, v_k$ in **DFS**, we have visited exactly the connected components containing $v_1, \ldots, v_k$

Shortest paths: no

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

Trees, forest, ancestors and descendants

Previous observation:

- DFS(G) gives a partition of G into vertex-disjoint rooted trees $T_1, \ldots, T_k$ (DFS forest)

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

- u is an **ancestor** of v if they are on the same T_i and u is on the path root $\rightsquigarrow v$
- equivalent: v is a **descendant** of u

Key property

Claim

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

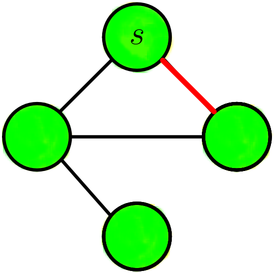
Proof. Let $\{v, w\}$ be an edge, and suppose we visit v first.

Then when we visit 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.



Observation

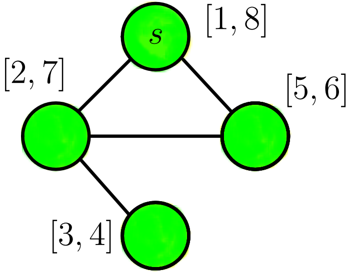
All edges are either **tree edges** or **back edges** (key property).

Start and finish times

Set a 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:

- these intervals are either contained in one another, or disjoint

Observation:

- if start$[u] <$ start$[v]$, then either **finish$[u] <$ start$[v]$** or **finish$[v] <$ finish$[u]$**.

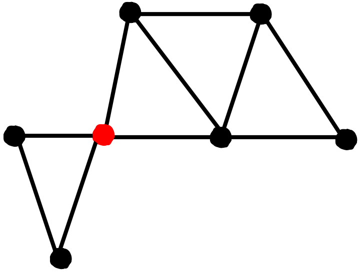
Proof: if start$[v] <$ finish$[u]$, we push v on the stack while u is still there, so we pop v before we pop u.

Cut vertices

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 points



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.

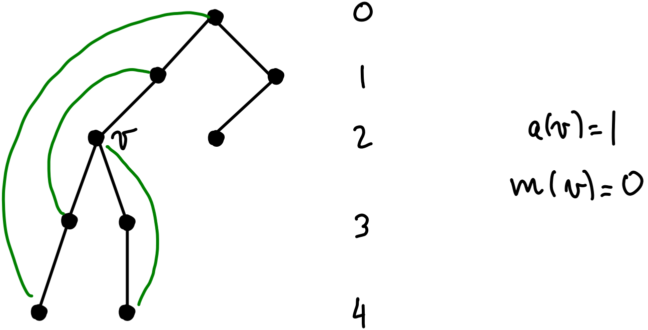
- if s has one child, removing s leaves T connected. So s not a cut vertex.
- suppose s has subtrees $S_1, \ldots, S_k$, $k > 1$.

  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

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



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)

Computing the values $m(v)$

Observation:

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

Conclusion:

- computing $a(v)$ is $\boldsymbol{O(d_v)}$ $\qquad$ d_v = degree of v
- knowing all $m(w_1), \ldots, m(w_k)$, we get $m(v)$ in $\boldsymbol{O(d_v)}$
- so all values $m(v)$ can be computed in $\boldsymbol{O(m)}$
  (remember $O(n + m) = O(m)$ when G connected)
- testing the cut-vertex condition at v is $\boldsymbol{O(d_v)}$
- testing all v is $\boldsymbol{O(m)}$

Exercise

write the pseudo-code