CS 341: ALGORITHMS (S18) — LECTURE 12
DEPTH-FIRST SEARCH

ERIC BLAIS

In the last lecture, we saw how the breadth-first search algorithm can solve the Graph Exploration problem (as well as a number of other similar problems, with appropriate slight modifications to the base BFS algorithm). Today, we examine a different algorithm for solving the same Graph Exploration problem: depth-first search.

1. Depth-first search

The breadth-first search algorithm can be described as a careful, thorough method for exploring the graph (which explores all the vertices at distance d form the source before moving on to explore any vertices at distance $d+1$). The depth-first search algorithm, by contrast, corresponds to an impatient and impulsive method for exploring a graph: whenever it discovers a new vertex, the depth-first search seeks to explore at least one of its incident edges. This can still result in a provably correct algorithm for solving the Graph Exploration problem; it just explores the vertices in a different order than the BFS does.

We can implement the depth-first search algorithm is by modifying the BFS algorithm to use a stack instead of a queue. It's also possible to implement depth-first search with a simple recursive algorithm. For this algorithm, we assume that visited is a global array of n Boolean values, and we put in placeholder functions PreVisit and PostVisit that we will define later.

Algorithm 1: Explore$(G=(V,E),v,p)$

```
visited[v] ← True;
PreVisit(v, p);
for each w ∈ Adj(G, v) do
    if not visited[w] then Explore(G, w, v);
PostVisit(v, p);
```

Algorithm 2: DFS$(G=(V,E),s)$

```
for each v ∈ V do
    visited[v] ← False;
Explore(G, s, ∅);
```

To solve the Graph Exploration problem, we simply need to implement the PreVisit or the PostVisit function to add v to a list of vertices in the same connected component as s and output that list at the end. The correctness of the resulting algorithm follows from the following argument.

Theorem 12.1. *The DFS algorithm calls* Explore *once on every vertex that is in the same connected component as s, and on no other vertex of V.*

Proof. That Explore is called at most once on every vertex in V follows from the fact that we only call it on a vertex when its status is visited$[v]$ = `False` and that the status is updated to `True` as soon as we call Explore on that vertex.

The vertices that are not in the same connected component as s in G are never explored because the DFS exploration follows only the edges of the graph.

And to show that DFS explores *all* the vertices in the same connected component as s, assume on the contrary that it misses some vertex u that is in that connected component. Since s and u are in the same connected component, we can find a simple path $s, v_1, v_2, \ldots, v_k, u$ from s to u in G. Let $z = v_i$ be the last vertex along the path that is visited by the DFS algorithm, and let $w = v_{i+1}$ be the first vertex along the path that is not visited. (Such a pair of vertices must exist if s was visited and u was not). But then we obtain a contradiction, since the Explore call on z examines all the neighbours of z and explores those vertices that have not yet been visited, so it would call Explore on w. □

For the time complexity of the algorithm, we see that Explore is called on at most n vertices, and the for loop is executed $O(m)$ times since each edge is visited at most twice. The total time complexity of the algorithm is therefore $O(n + m)$, the same as the BFS algorithm.

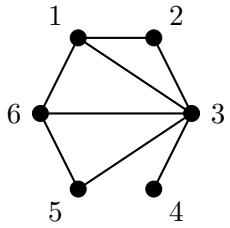
2. Spanning tree

Just as we can easily modify the BFS algorithm to generate a spanning tree of a connected graph, we can also do this with the DFS algorithm. In this case, we simply need to let $T = (V, E')$ be a global tree. Initially, the set of edges of the tree is taken to be $E' = \emptyset$, then we implement PreVisit to add the edge between v and its parent p in E'.

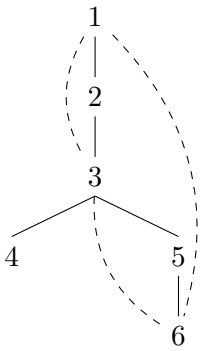
Algorithm 3: PreVisit(v, p)

if $p \neq \emptyset$ **then** $E' \leftarrow E' \cup \{(v, p)\}$;

Let's look at an example. When we run the resulting DFS algorithm on the graph



we obtain the following tree (assuming the neighbours of a given vertex are visited in order).



The extra vertex of the graph that is not included in the spanning tree is drawn with a dotted line in this example.

What is particularly useful about the spanning tree obtained by the DFS algorithm is that it captures some important structural information about the original connected graph G. Let us call the edges contained in the spanning tree defined by the DFS algorithm *tree edges*, and let the remaining edges of G be called *non-tree edges*. Furthermore, when a vertex v is in the subtree rooted at w in the spanning tree, we will say that v is a *descendant* of w and that w is an *ancestor* of v.

Lemma 12.2. *Every non-tree edge in a connected graph G connects an ancestor to one of its descendants.*

Proof. Consider any non-tree edge $(u, v) \in E$. Say that u was discovered first in the DFS of G. Then every node that is discovered while following a path from u to other unexplored vertices ends as a descendant of u; v must be one of those vertices because otherwise it would be explored directly from u (and (u, v) would be a tree edge) before we complete the exploration of u. □

We can get even more structural information from the DFS tree by not only building the tree itself but also keeping track of the times at which (or the order in which) each node is first discovered and when it is completely explored. We can do this easily with a `clock` counter that we initially set to 0 and use in the following implementations of PreVisit and PostVisit.

Algorithm 4: PreVisit(v, p)

$pre[v] \leftarrow$ `clock`;
`clock` $\leftarrow$ `clock` $+ 1$;

Algorithm 5: PostVisit(v, p)

$post[v] \leftarrow$ `clock`;
`clock` $\leftarrow$ `clock` $+ 1$;

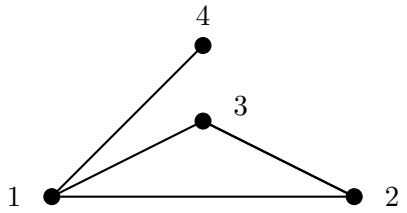
The theorem we established about the structure of non-tree edges can also be rephrased in terms of the intervals defined by these timings.

Theorem 12.3. *For any two vertices $u, v \in V$ in a connected graph $G = (V, E)$, either the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are disjoint, or one is contained within the other one.*

We could continue to work on developing more structural results that we get with the DFS tree, but let's stop here and see how the DFS tree and timing information can be used to determine just *how* well connected a graph is.

3. Finding cut vertices

A graph is connected, as we already saw, if there is a path between any two vertices in the graph. Some graphs are connected, but "barely so", in the sense that if we remove one of the vertices of the graph, the resulting subgraph is no longer connected. For example, in the graph



if we remove vertex 1 then we are left with a graph that is no longer connected. Vertex 1, in this case, is known as a *cut vertex*.

Definition 12.4. A vertex $v \in V$ is a *cut vertex* in a connected graph $G = (V, E)$ if the subgraph $G' = (V \setminus \{v\}, E')$ with $E' = \{(u, w) \in E : u, w \neq v\}$ is not connected. A graph that has no cut vertex is called *2-connected* (or sometimes *2-vertex-connected*).

These definitions immediately suggest the problem of determining if a graph is 2-connected or not.

Definition 12.5 (Testing 2-connectedness)**.** Given a connected graph $G = (V, E)$, determine if it is 2-connected or not. (Equivalently: determine if it has a cut vertex.)

We can use the DFS tree of a graph to solve this problem. As a first step, we can easily test if the root of that tree is a cut vertex.

Lemma 12.6. *The root v of the DFS tree of a connected graph $G = (V, E)$ is a cut vertex in G if and only if it has at least 2 children.*

Proof. If the root v has at most 1 child, then removing the node v leaves a tree (or an empty graph), so that G' is still connected.

If the root v has at least 2 children, then Lemma 12.2 guarantees that there can be no edge connecting a vertex u from the left subtree to a vertex w in the right subtree, so that any path between a vertex in one subtree to the other must pass through v. Therefore, v is a cut vertex. □

There is also a similar lemma for the other vertices of the DFS tree.

Lemma 12.7. *The non-root vertex v of the DFS tree T of a connected graph $G = (V, E)$ is a cut vertex in G if and only if it has a child u such that the subtree of T rooted at u has no non-tree edge going from one of its vertices to an ancestor of T.*

Proof. If the condition is satisfied, then removing v from G isolates the subtree rooted at u from the rest of the graph, so the resulting subgraph is not connected.

In the other direction, if v is a cut vertex of G then removing it must isolate some vertices from the rest of the graph, which means that one of its subtrees must be disconnected from the rest of the tree and, in particular, has no non-tree edge going to any ancestor of v. □

So the problem of determining if a graph has a cut vertex can be reduced to the problem of determining if a vertex satisfies either of the above conditions. The condition for the root of the tree is easily checked. And for the other nodes, we can determine if the condition is satisfied by using the $pre[v]$ values to keep track of the earliest ancestor connected by an edge from the current subtree. This is implemented with slight changes to the EXPLORE function.

Algorithm 6: EXPLORE'$(G = (V, E), v, p)$

```
visited[v] ← True;
PreVisit(v, p);
low[v] ← pre[v];
for each w ∈ Adj(G, v) do
    if not visited[w] then
        Explore'(G, w, v);
        low[v] ← min{low[v], low[w]};
    else
        low[v] ← min{low[v], pre[w]};
if (p ≠ ∅ and low[v] ≥ pre[v]) or (p = ∅ and |Adj(G, v)| ≥ 2) then
    Print v is a cut vertex;
PostVisit(v, p);
```