CS 341 - Algorithms, Spring 2021. University of Waterloo

Lecture 6: Depth First Search

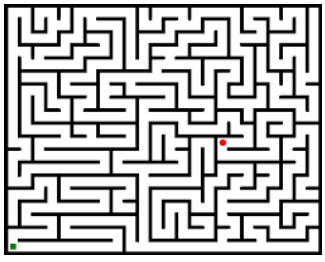
Depth first search is another basic search method in graphs, and this will be useful in identifying more refined connectivity structures as we will see today and next time.

Motivating Example

Last time we imagined that we would like to search for a person in a social network, and BFS is a very natural strategy (asking friends, then friends of friends, and so on).

There are other situations that using depth first search is more natural.

Imagine that we are in a maze searching for the exit.



We could model this problem as a s-t connectivity problem in graphs.

Each square of the maze is a vertex, and two vertices have an edge if and only if the two squares are reachable in one step.

Then, finding a path from our current position to the exit is equivalent to finding a path between two specified vertices in a graph (or determine that none exists).

How would you search for a path in the maze?

There are no friends to ask, and it doesn't look efficient anymore to explore all vertices with distance one, then distance two and so on (as we have to move back and forth).

Assuming we have a chalk and can make marks on the ground. Then it is more natural to keep going on one path bravely until we hit a dead end, and make some marks on the way and also on the way back so that we won't come back to this dead end again, and only explore yet unexplored places.

This is essentially depth first search (DFS).

Depth First Search

As for BFS, we define DFS by an algorithm. DFS is most naturally defined as a recursive algorithm.

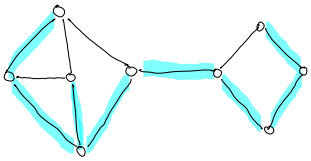
DFS Algorithm

Input: an undirected graph $G=(V,E)$, a vertex $s \in V$.

Output: all vertices reachable from s.

[Main program] visited[v] = false $\forall v \in V$, visited[s] = true. explore(s).

```
explore (u)        // recursive function explore.
    for each neighbor v of u
        if visited[v] = false
            visited[v] = true .   explore(v).
```



Time Complexity

The analysis of the time complexity is similar to that in BFS.

For each vertex u, the recursive function explore(u) is called at most once.

When explore(u) is called, the for loop is executed at most $deg(u)$ times.

Thus the total time complexity is $O(n + \sum_{v \in V} deg(v)) = O(n+m)$ word operations.

Stack and Queue

There is a way to write DFS non-recursively.

The idea, not surprisingly, is to use a stack, as recursive programs are implemented using stacks in our computers

The resulting program using stack is syntatically very similar to that of BFS.

So, one can think about the two fundamental search methods correspond to two fundamental data structures.

Try to write it out or see [kT] for the solution.

BFS and DFS

The basic lemma about graph connectivity still holds for DFS.

Lemma There is a path from s to t if and only if visited[t] = true at the end.

The proof is the same as in BFS and is left as an exercise.

The lemma shows that DFS can also be used to check s-t connectivity, to find the connected component containing s, and to check graph connectivity, all in $O(m+n)$ time.

And we can also find all connected components in $O(m+n)$ time using DFS (exercise).

The main difference from BFS is that DFS cannot be used to compute the shortest path distances, and this is the main feature of BFS.

But as we shall see, DFS can be used to solve some interesting problems that BFS cannot do.

DFS Tree

As for BFS, we can construct a DFS tree to trace out the path from s.

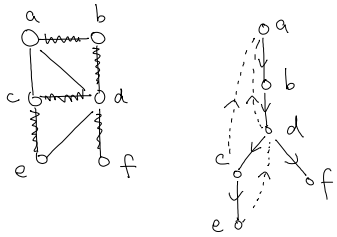
Again, when a vertex v is first visited when we explore vertex u,

we say vertex u is the parent of vertex v.

By the same argument as in BFS, these edges (v, parent[v])

form a tree, and we can use them to find a path to s.

We call this a DFS tree of the graph.



Note that a graph could have many different DFS trees depending on the order of exploring the neighbors of vertices. The same can be said for BFS trees.

Definitions / Terminology for DFS trees

- The starting vertex s is regarded as the root of the DFS tree.
- A vertex u is called the parent of a vertex v if the edge uv is in the DFS tree, and u is closer to the root than v is to the root.
- A vertex u is called an ancestor of a vertex v if u is closer to the root than v, and u is on the path from v to the root.

  In this situation, we also say v is a descendant of vertex u.
- A non-tree edge uv is called a **back edge** if either u is an ancestor or descendant of v.

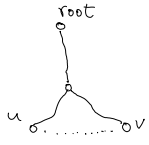
  It is called a back edge because this edge from the descendant to the ancestor.

In the above example, b is an ancestor of e and f, but c is neither an ancestor nor descendant of f.

The following is a simple but important property that we will use.

Property (back edges) In an undirected graph, all non-tree edges are back edges.

Proof Suppose by contradiction that there is an edge between u and v but u and v are not an ancestor-descendant pair.



WLOG assume that u is visited before v.

Then, since $uv \in E$, v will be explored before u is finished, and thus u will be an ancestor of v, a contradiction. □

Starting Time and Finishing Time

We record the time when a vertex is first visited and the time when its exploring is finished. These information will be very useful in design and analysis of algorithms.

To be precise, we include the pseudocode in the following.

[Main program] visited[v] = false $\forall u \in V$. time = 1. visited[s] = true. explore(s).

explore(u) // recursive function explore.

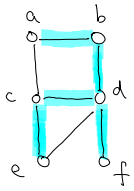
start[u] = time. time ← time + 1.

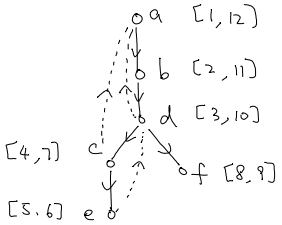
for each neighbor v of u

if visited[v] = false

visited[v] = true. explore(v).

finish[u] = time. time ← time + 1.





Property (parenthesis) The intervals [start(u), finish(u)] and [start(v), finish(v)] for two vertices u and v are either disjoint or one is contained in another. The latter case happens precisely when u, v are an ancestor-descendant pair.

Cut Vertices and Cut Edges

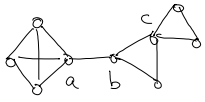
Suppose an undirected graph is connected.

We would like to identify vertices and edges that are critical in the graph connectedness.

A vertex v is a cut vertex (aka an articulation point, or a separating vertex) if G-v is not connected, i.e. removal of v and its incident edges disconnects the graph.

An edge e is a cut edge (aka a bridge) if G-e is not connected.

In the example, vertices a, b, c are cut vertices and edge ab is a cut edge.

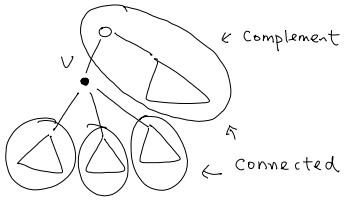


Observations and Ideas

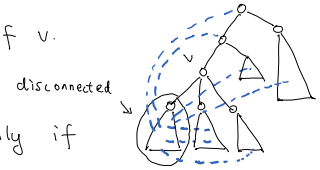
The idea is to use a DFS tree to identify all cut vertices and cut edges.

Consider a vertex v which is not the root. We would like to determine whether v is a cut vertex.

When we look at the DFS tree, all the subtrees below v are connected, as well as the complement of the subtree at v (see the picture).



The main observation is the property that all the non-tree edges are back edges (proved above), and so the only way for a subtree below v to be connected outside is to have edges going to an ancestor of v.



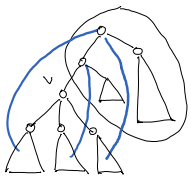
Claim A subtree T_i below v is a connected component in G-v if and only if there are no edges with one endpoint in T_i and another endpoint in a (strict) ancestor of v.

Proof ⇐) By the back edge property, all the non-tree edges are back edges, so there are no edges going to another subtree below v nor edges going to another subtree of the root.
So, if there are no such edges going to a (strict) ancestor of v, then T_i must be a component in $G-v$.
⇒) On the other hand, if such edges exist, then T_i is connected to the complement even after v is removed, and so T_i won't be a connected component in $G-v$. □

The same argument applies to each subtree below v gives the following characterization of a cut vertex.

Lemma For a non-root vertex v in a DFS tree, v is a cut vertex if and only if there is a subtree below v with no edges going to a (strict) ancestor of v.

Proof ⇒) If every subtree below v has some edges going to an ancestor of v, then every subtree is connected to the complement. See picture.
So, $G-v$ is connected and thus v is not a cut vertex.



⇐) If some subtree T_i below v has no edges going to an ancestor of v, then T_i will be a connected component in $G-v$ by the previous claim. and thus v is a cut vertex. □

It remains to consider the root vertex of the DFS tree. The proof is left as an exercise.

Lemma For the root vertex v of a DFS tree, v is a cut vertex if and only if v has at least two children.

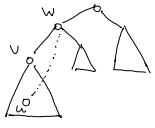
With these lemmas, we know how to determine if a vertex is a cut vertex by looking at a DFS tree

---

Algorithm

We are ready to use the above lemmas to design a $O(n+m)$ time algorithm to report all cut vertices.
To have an efficient implementation, the idea is to process the vertices of a DFS following a bottom up ordering. and keep track of how "far up" the back edges of a subtree can go (i.e. how close to the root).
By the lemma, for a non-root vertex v, v is not a cut vertex if and only if all subtrees below v have an edge that goes above v.

What would be a good parameter to keep track of how far up we can go?
The starting time would be a good measure, because an ancestor always has an earlier / smaller starting time than its descendants.
(We could also do it in other ways, e.g. by recording the distance to the root instead.)



Let us define a value low[v] for each vertex on the DFS tree.

$$low[v] := \min \left\{ start[v],\ \min \{ start[w] \mid uw \text{ is a back edge with } u \text{ being a descendant of } v \text{ or } u = v. \} \right\}$$

Informally, low[v] records how far up we can go from the subtree rooted at v.

We will be done if we can prove the following two things:

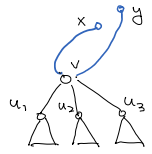
① We can compute low[v] for all $v \in V$ in $O(n+m)$ time.

② We can identify all cut vertices in $O(n+m)$ time using the low array.

For ①, we compute the low values from the leaves of the DFS tree to the root of the DFS tree.

The base case is when v is a left. Then we can compute low[v] by considering all the edges incident on v and taking the minimum of the starting time of the other endpoint. This takes $O(\deg(v))$ time.

By induction, suppose the low values of all children of v are computed.



Then, to compute low[v], we just need to take the minimum of the low value of its children, as well as the start time for all back edges involving v. This takes $O(\deg(v))$ time.

In the example given in the picture, $low[v] = \min \{ low[u_1], low[u_2], low[u_3], start[x], start[y] \}$.

It should be clear that low[v] is computed correctly, assuming the low values of all its children are correct, and so the correctness can be established by induction.

By this bottom-up ordering, every vertex on the tree is only processed once, and thus the total time complexity is $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

For ②, to check whether a non-root vertex v is a cut vertex, we just need to check whether $low[u_i] < start[v]$ for all children u_i of v.

If so, then v is not a cut vertex, as all subtrees below v have a back edge going above v.

Otherwise, if $low[u_i] \geq start[v]$, then the subtree rooted at u_i will be a connected component in G-v, and thus v is a cut vertex. These arguments are all covered in the first lemma.

The root vertex is handled using the other lemma.

This completes the description of a linear time algorithm to identify all cut vertices given the low array.

Exercise: Extend the algorithm to identify all the cut edges.

References: [DPV 3.2]. Cut vertices and cut edges are from the exercises of [DPV].