CS 341 - Algorithms, Winter 2016. University of Waterloo

Lecture 6: Depth first search

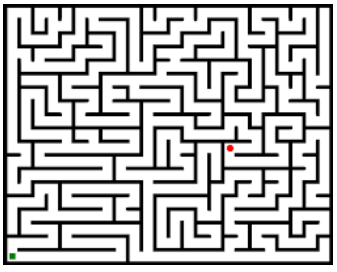
Depth first search is another basic search method in graphs, and this will be useful in in identifying more refined connectivity structures as we will see today and next lecture.

Motivating example

Last time we imagine that we are searching for a person in a social network, and breadth first search is a natural strategy (asking friends, 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 the graph (or determine 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, and make some marks on the way and also on the way back so that we won't go into the dead end again, and only explore yet unexplored places.

This is essentially depth first search (DFS).

Depth first search

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

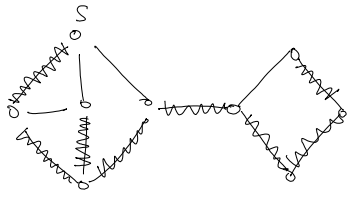
DFS algorithm

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

Output: all vertices reachable from s

```
(main program)   visited[v] = false  ∀v∈V.   // global variable array
                 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 of BFS.

Each vertex u is called the recursive function explore(u) once.

When explore(u) is executed, the for loop is executed deg(u) times.

The total time complexity is thus $O(n + \sum_{v \in V} deg(v)) = O(n+m)$.

Remark: There is a way to write DFS non-recursively. The idea (not surprisingly) is to use a stack (since recursive programs are executed using a stack in our computer). The resulting program using stack is syntatically very similar to that of BFS. So, one can think of the two basic search methods correspond to two basic data structures, queues and stacks.

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

DFS tree

The basic claim about graph connectivity still holds for DFS.

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

The proof is the same and we leave it as an exercise.

The claim 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.

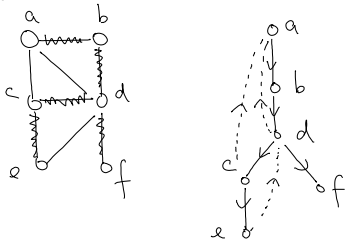
And we can also find all connected components in $O(n+m)$ time.

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

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

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

Again, when a vertex v is first visited when we are exploring vertex u, we say vertex u is the parent of vertex v.



By the same argument as in BFS, these edges from v to parent(v) form a tree, and we can use them to find a path to the starting vertex.

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.

Definitions for DFS trees

- The starting vertex 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.
- 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 call v a descendent of vertex u.
- A non-tree-edge uv is called a **back edge** if either u is an ancestor of v or u is a descendent of v. It is called a back edge because this edge was explored from the descendant to the ancestor.

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

Useful information and properties

There are some basic information and properties about DFS that are very useful in the design and analysis of algorithms.

Starting time and finishing time

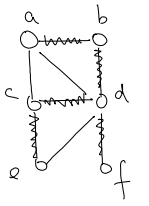
We keep track of the time when a vertex is first visited and finished exploring.

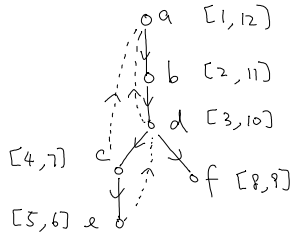
To be precise, we include the pseudocode in the following.

```
(main program)   visited[v] = false  ∀ v∈V .     // global variable array
                 time = 1 .                       // the new variable keeping track the time
                 visited[s] = true .  explore(s)

explore(u)       // recursive function explore.
    start[u] = time .    time ← time + 1          // start is an array to store starting time
    for each neighbor v of u
        if  visited[v] = false
            visited[v] = true .  explore(v).
    finish[u] = time .   time ← time + 1          // finish is an array to store finishing time
```

Side remark: There are different names for these variables, e.g. birth-death, pre-post, start-end, etc.

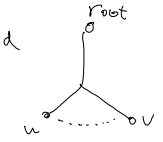




Property 1 (parenthesis) The intervals [start(u), finish(u)] and [start(v), finish(v)] for u and v are either disjoint or one is contained in another. The latter case happens precisely when u, v are an ancestor-descendent pair.

Property 2 (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 and u and v are not an ancestor-descendant pair. Say u has an earlier starting time. Then v will be visited before u is finished, and u will be the parent of v, contradicting that they are not an ancestor-descendant pair. □



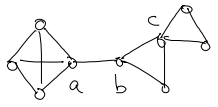
Cut vertices and cut edges

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 (or 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 (or a bridge) if $G-e$ is not connected (i.e. removal of e disconnects the graph).

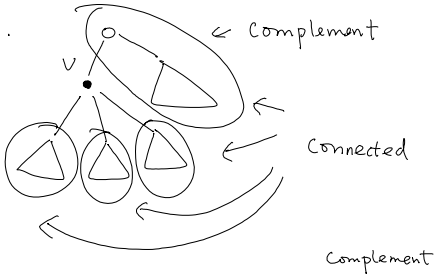


In the example, vertices a,b,c are cut vertices and edge ab is a cut edge.

Observations

We would like to show that DFS can be used to identify all cut vertices and cut edges.

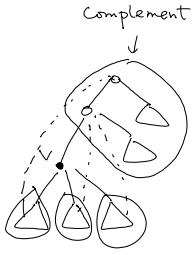
Consider a vertex v which is not the root of the DFS tree.



Then, when we look at the DFS tree, all the subtrees below v are connected, as well as the complement of the subtree at v (called it complement; see the picture).

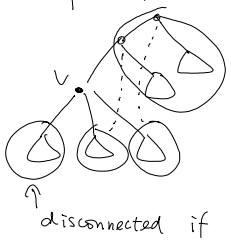
We know from property 2 that all the non-tree edges are back edges.

If there is an edge from a subtree below v to an ancestor of v, then that subtree is connected to the complement, even in the absence of v.



So, if all subtrees below v have such an edge going to some ancestor of v, then even when v is deleted, the remaining graph is connected, and v is not a cut vertex.

On the other hand, if some subtree doesn't have an edge going to some ancestor of v, then that subtree must be disconnected when v is removed, because all non-tree edges must be back edges by property 2, so there is no other way for that subtree to connect to outside.



To summarize, we have the following conclusion.

Claim 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 an ancestor of v.

It remains to consider the root vertex.

Claim For the root vertex v in a DFS tree, v is a cut vertex if and only if

v has at least two children.

The proof is left as an exercise (again using property 2).

With these claims, we characterize how to determine a vertex is a cut vertex by looking at a DFS tree.

Algorithm

Now, we use the above observations to design a $O(n+m)$ time algorithm to report all cut vertices.

The idea is to look at a DFS tree from bottom to up, and keep track of how far up the back edges of a subtree can go.

In particular, for a non-root vertex v, all subtrees below v have an edge go above v if and only if v is not a cut vertex.

What would be a good parameter to keep track of how far up we can go?

The starting time would be a good measure, as a vertex on a path to the root has a earlier (or smaller) starting time if and only if it is closer to the root.

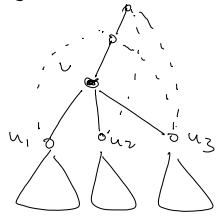
Let us define a value $low[v]$ for each vertex:

$$low[v] := \min \begin{cases} start[v] \\ start[w] & \text{where } uw \text{ is a back-edge with } u \text{ a descendent of } v \text{ or } u=v. \end{cases}$$

It remains to prove two things. One is that the low array can be computed in $O(n+m)$ time, and another is that we can identify all cut vertices in $O(m+n)$ using the low array.

We compute the low values from leaves to root.

Suppose the low values of all the children are computed.



In the picture, suppose all $low[u_1]$, $low[u_2]$, $low[u_3]$ are computed.

Then, to compute $low[v]$, we just need to take the minimum of these three values, as well as the start values for all the back edges involving v.

By this bottom up order, all vertices would only be processed once, and thus the total time complexity is $O(n + \sum_{v \in V} deg(v)) = O(n+m)$.

To check whether v is a cut vertex, we just need to check whether $low[u_i] < start[v]$,

$low[u_2] < start[v]$, and $low[u_3] < start[v]$.

This contains all the main ideas to do the programming problem.

Question: Can you extend the algorithm to identify all the cut edges?

References: [DPV 3.2]; cut vertices and cut edges are in the exercises of [DPV].