CIS 575. Introduction to Algorithm Analysis
Material for April 17, 2024

Depth-First Search

The topic of this note is part of what is covered in *Cormen*'s Section 20.3.

1 Depth-First Search

Some data structures are easier to handle than others:

Trees are good and convenient in that they can be processed and manipulated using a recursive top-down approach.

Graphs may in general be harder to handle in a systematic way since they lack a clear structure.

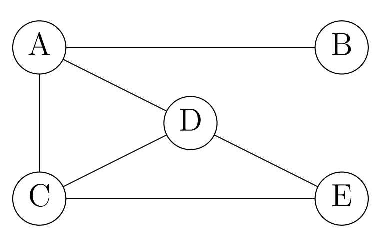
The **good news** is that **a graph may be viewed as a tree**. This may seem too good to be true, and we do indeed need to add a qualification:

> A graph may be viewed as a tree, **together with some extra edges**.

We shall now show in detail how to do so, in this note for **un**directed graphs (as this is the simplest case), and in the next note for **directed** graphs (which is quite similar but a bit more involved).

Given an undirected graph $G = (V, E)$, Figure 1 presents an algorithm for selecting from E, by Depth First Search, an acyclic set of edges T that "spans" the graph. As the algorithm executes, the **color** of each node changes from white (before it was *discovered*) to gray (while it is being processed) to black (after it is *finished*). The algorithm also records for each node u the time $d[u]$ of its discovery, and the time $f[u]$ when it finishes.

We shall illustrate the algorithm on the graph to the right, and shall choose to process nodes in alphabetical order. We thus first call DFS on node A; this will cause the following actions:



1. A is colored gray; then the edge from A to B is included in T and DFS is called on B
2. B is colored gray; since A is not white, no edge from B will be explored and the call to DFS on B finishes as B is colored black

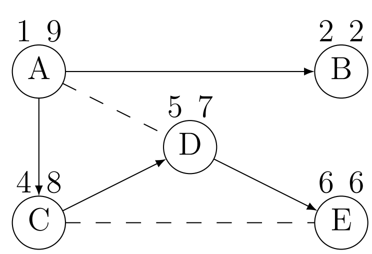
```
DepthFirstSearch(V, E)
        color all nodes in V white; T ← empty
        while V has still white nodes
           pick white node u; DFS(u)

DFS(u)   // u is white
        color[u] ← gray
        d[u] ← current_time
        foreach w where E contains an edge from u to w
           if color[w] = white
              T ← T ∪ {u → w}
              DFS(w)
        f[u] ← current_time
        color[u] ← black
```

Figure 1: The Depth-First Search Algorithm.

3. back at A, we go on to explore its next neighbor and include the edge from A to C in T and call DFS on C
4. C is colored gray; since A is not white, we go on to explore D and include the edge from C to D in in T and call DFS on D
5. D is colored gray; since neither A nor C is white, we go on to explore E and include the edge from D to E in T and call DFS on E
6. E is colored gray; since neither C nor D is white, no edge from E will be explored and the call to DFS on E finishes as E is colored black
7. back at D, there are no further edges to explore, so the call to DFS on D finishes as D is colored black
8. back at C, we must examine the edge to E but that edge will not be explored as E is not white; thus the call to DFS on C finishes as C is colored black
9. back at A, we must examine the edge to D but as D is not white we are done and can color A black.

Now there are no more white nodes left, so the initial **while** loop will terminate after just one iteration, having produced T given on the right where we use dashed lines for the graph edges *not* included in T, and where we write the discovery/finish times at the top left/right of a node.



Spanning Tree(s). If the given graph is **connected**, as for our example, the **while** loop will execute once only, and T will be a tree that spans the graph.

> The shape of the tree will depend on the start node. In our example, the choice of A yielded a tree where the root has two children; choosing B would have yielded a tree where the root has only one child.

If the given graph is **not connected**, the **while** loop will execute several times, and T will not be a spanning tree but rather a "spanning forest". But in the following we shall often implicitly assume that the graph G is connected and hence T is indeed a spanning tree.

Tree Traversal. We observe that

- the **discovery** times correspond to a **pre**-order traversal of the tree T, that is, we start with the root and then recursively explore the branches;
- the **finish** times correspond to a **post**-order traversal of the tree T, that is, we recursively explore the branches and finish with the root.

Ancestry. We observe that for two nodes, their discovery & finish times can relate in two different ways:

1. It may be that one node has a finish time that is earlier that the discovery time of the other node. In that case, neither is an ancestor (in the tree T) of the other.
2. It may be that one node u was discovered after another node w but before w finishes, but then u will finish before w finishes (due to the calls to DFS forming a stack). In that case, u will be a descendant (in the tree T) of w (and thus w an ancestor of u).

In our example, node B is finished before either of the nodes C, D or E have been discovered; this shows that in the tree T, B is not an ancestor of these nodes, nor a descendant (but rather B is a sibling, an uncle, etc). And the discovery time of E is later than the discovery time of C but the finish time of E is earlier than the finish time of C; this shows that in the tree T, C is an ancestor of E.

We have seen that for a tree produced by the depth-first search algorithm, valuable ancestry information can be immediately retrieved from the discovery & finish times, rather than through a potentially expensive search up and down in the tree.

Edge Classification Assume that the graph G has an edge e between u and w that is *not* a **tree edge**, that is, *not* included in T. If say u was discovered before w (the other case is symmetric) then u would have to examine w but since e was not included in T it must be the case that then w already had been discovered. Thus w was discovered before u is finished (but after u was discovered) which (cf. the previous paragraph) shows that u is an ancestor of w in the tree T.

A non-tree edge between a node and an ancestor is called a **back edge**. We have just argued that *all* edges that were not selected for the tree will be back edges.

In our example, we had two back edges: between D and A, and between E and C. Both connect a node to its grandparent in T. It is obviously impossible for a back edge to connect a node to its parent in T (as such an edge would be a tree edge), but one can easily imagine situations where a back edge connects a node to its great-grandparent, etc.