(M. Dietzfelbinger, 2021-03-14)

3.2 Depth-First-Search in undirected graphs

Given a graph $G = (V, E)$, we want to "explore" the graph, meaning that we want to visit all nodes and all edges and collect information about G (e.g., if it is connected, which means that one can get from every node to every other node along a path).

3.2.1 Exploring from a node

The basic idea is to start at one start node and then hop from node to node along edges. When a node v has been reached, "going forward" from v has priority over looking around at all neighbors of v. So if the process sits at v and sees a new node w along an edge (v, w), the next step is going to w. One backs up from a node v only if no new node can be seen along any edge from v. In this case one goes back to the node from which v was reached.

(In the book this is explained as a systematic way to explore the reachable part of a maze.)

For organization, nodes have a tag that can have values "new" or "visited". Initially, all nodes are marked "new". We use a recursive procedure **explore**(v). Its purpose is to organize the actions at a node v, which must be "new". First, mark v as "visited", maybe do something with respect to v ("previsit"). Then go forward, if possible (do a recursive call **explore**(w) for a "new" neighbor w of v); after returning from a recursive call check whether it is still possible to go forward from v in a different direction, and do this, if possible. When all edges out of v have been checked, possibly do something else with respect to v ("postvisit"). Then finish the call explore(v), and return to the node from which v was first reached.

In pseudocode, this looks as follows:

Algorithm 1: (Exploring from a node v in graphs)

```
procedure explore(v): // goes forward into G from "new" node v
Input: node v, which must be "new"
  // Situation: graph G, some nodes may be "visited", the rest is "new"
  mark v as "visited";
  previsit(v);    // some action at v for first visit
  for each edge (v,w) do    // in the order of the adjacency list of v
    if w is "new" then explore(w)
  postvisit(v); // some action at v for last visit
```

Example: Fig. 3.4 in the book.

We note some properties.

Claim 1: If all nodes of G are "unvisited", a call explore(v) finds all nodes u that can be reached from v along a path

$$v = v_0, v_1, \ldots, v_t = u, \text{ with } (v_{i-1}, v_i) \in E \text{ for } 1 \leq i \leq t.$$

(All these nodes u comprise the "connected component of v".)

Proof. Indirect. Suppose for a contradiction there is a path $v = v_0, v_1, \ldots, v_t = u$ in G, but explore(u) is not called. Let $1 \leq i \leq t$ be minimal such that explore(v_i) is not called. Then during the execution of explore(v_{i-1}) edge (v_{i-1}, v_i) is inspected, v_i is "unvisited", hence explore(v_i) is called, contradiction. □

Claim 2: In the general situation, where some nodes are marked "visited" and the others are "new", a call explore(v) finds all nodes u that can be reached from v along a path of "new" nodes (a "white path").

Proof. Practically the same as of Claim 1. Only consider a path $v = v_0, v_1, \ldots, v_t = u$ of "new" nodes. □

Note that Claim 2 explains better what explore(v) does than what is said in Figure 3.3 in the book.

Visiting a node costs time $O(1)$ plus the time for checking its edges (plus any recursive calls, but this is charged to the other nodes). Each edge is looked at twice, once in each direction. So the total running time of explore(v) is $O(1)$ for each node looked at and $O(1)$ for each edge looked at. Altogether this is $O(n_v + m_v)$, where n_v is the number of nodes that can be reached from v and m_v is the number of edges that can be reached from v.

We consider all edges (u, v) with the property that explore(v) is called directly out of explore(u). Such edges are called *tree edges*. The root is v_0, the node for which the first call explore(v_0) is carried out. The tree edges form a *tree*, which means that each node is reachable from v_0 along a directed path of tree edges and each node excepting v_0 has exactly one ingoing (directed) tree edge.

Example: Figure 3.4 in the book.

3.2.2 Exploring the whole graph

An outer loop makes sure that every node and every edge is visited.

Algorithm 2: (Depth First Search [DFS] in undirected graphs)

DFS(G): // visits every node and every edge
  mark all $v \in V$ as "new"
  for each $v \in V$ **do if** v is "new" **then** explore(v)

At the first glance this looks silly since we first mark all nodes as "new" and then test them for being "new". However, the calls explore(v) cause other recursive calls and

may flip the label of other nodes to "visited".

It is clear that by DFS(G) all nodes and all edges of G are visited. The running time is $O(n+m)$, since every node and every edge causes cost $O(1)$.

Looking at the tree edges, one obtains a set of DFS trees, which is called the *DFS forest*.

Example: Figure 3.6 in the book.

3.2.3 Connectivity

We say that u and v are *connected* in $G=(V,E)$ (notation: $u \sim v$) if there is a path $u=v_0,v_1,\ldots,v_t=v$ in G that connects u and v. The binary relation $\sim$ is an equivalence relation (it is reflexive, symmetric, and transitive). The equivalence classes are called the *connected component*s of G. Graph G is called *connected* if it has only one connected component.

Using one call DFS(G) we can test whether G is connected (this is the case if there is exactly one call explore(v) in the outer loop), and we can calculate the connected components of an arbitrary graph. For this, every node gets an attribute *comp* (its component number) and we introduce a counter *current_comp*, initialized with 0. Procedure DFS is modified as follows:

Algorithm 3: Label connected components

```
DFS(G):
  current_comp := 0;
  mark all v ∈ V as "new"
  for each v ∈ V do if v is "new" then current_comp++; explore(v)
```

Further, the previsit(v) procedure has the line

$comp[v] := current\_comp$

in its body. When DFS(G) is finished, every node is labeled with the number of its connected component.

3.2.4 Numbering visiting events

We want to keep track of the order in which exploration of nodes starts and ends. For this, we introduce a new counter *clock*, initialized to 0 at the beginning of DFS(G). Whenever exploring a node v is started, we increase *clock* and record the current value in pre[v] (the *preorder number* or *DFS number* of v). Whenever exploring a node v ends, we increase *clock* and record the current value in post[v] (the *postorder number* or *finishing number* of v).

This can easily be done by including the initialization $clock := 0$ at the beginning of DFS(G) and the line

$clock$++; pre[v] := $clock$

in previsit(v) and the line

$clock$++; post[v] := $clock$

in postvisit(v).

Example: Fig. 3.6 in the book.

Nesting property: Of intervals $[\text{pre}[u], \text{post}[u]]$ and $[\text{pre}[v], \text{post}[v]]$ with $\text{pre}[u] < \text{pre}[v]$ either one is included in the other:

$$\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{post}[u] \qquad (\text{ in the book: } [_u\ [_v\ ]_v\ ]_u\ )$$

or they are disjoint:

$$\text{pre}[u] < \text{post}[u] < \text{pre}[v] < \text{post}[v] \qquad (\text{ in the book: } [_u\ ]_u\ [_v\ ]_v\ ).$$

The reason is that if explore(v) is called at a point in time when explore(u) has started but not yet finished then explore(v) must finish before explore(u) can finish. We'll come back to the nesting property later in the context of directed graphs.

3.3 DFS in directed graphs

We want to apply DFS to a directed graph $G = (V, E)$. We use almost the same procedures as in the undirected case. (Procedure explore(v) inspects the edges going out from v, not the incoming edges.) We do not activate the labeling with component numbers, though. We get node labelings with preorder numbers and postorder numbers.

With each node we associate a *status*, which changes during the run of the DFS procedure. We say

- v is "new" before explore(v) has started (i.e., pre[v] is undefined),
  color code: white;
- v is "active" when explore(v) has started but not yet finished (i.e., pre[v] is defined, but post[v] is undefined);
  color code: red;
- v is "finished" when explore(v) has finished (i.e., both pre[v] and post[v] are defined)
  color code: grey.

(There is no need for an extra status tag.)

Tree edges (T): An edge $(u, v) \in E$ is a *tree edge* if explore(v) is called directly out of explore(u). As in the undirected case, only considering the tree edges, the digraph is split into trees, which form the *DFS forest* resulting from the DFS.

Note that for a tree edge (u, v) we have

$$\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{post}[u], \text{ or } [_u\ [_v\ ]_v\ ]_u\ .$$

A *root* of one of the DFS trees is a node that has no ingoing tree edge. It is clear that v is a root if and only if explore(v) is called in the outer loop in DFS(G). It is not hard to see that v is a root if and only if v cannot be reached along a directed path from any node u that is looked at before v in this outer loop.

Other edge types. (See picture on page 88 of the book.)

With DFS we can classify the non-tree edges of G into three further categories: forward edges (F), back edges (B), and cross edges (C), which are defined in the following.

Forward edges (F): An edge (u, v) is a *forward edge* if there is a path

$$u = v_0, v_1, \ldots, v_t = v$$

of *tree edges* (v_{i-1}, v_i), with $t \geq 2$. Thus, forward edges jump forward along a path of tree edges of length at least 2. It follows immediately from the same relation for tree edges that we have

$$\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{post}[u], \text{ or } [_u\ [_v\ ]_v\ ]_u\ ,$$

for a forward edge (u, v).

Back edges (B): An edge (u, v) is a *back edge* if there is a path

$$v = v_0, v_1, \ldots, v_t = u$$

of *tree edges* (v_{i-1}, v_i), with $t \geq 0$. (Note that $u = v$ is allowed here!) Thus, back edges jump backwards along a path of tree edges (which may have length 0 or 1 or larger). Conversely to forward edges, we have

$$\text{pre}[v] \leq \text{pre}[u] < \text{post}[u] \leq \text{post}[v], \text{ or } [_v\ [_u\ ]_u\ ]_v\ ,$$

for a back edge (u, v).

Cross edges (C): An edge (u, v) is a *cross edge* if it goes from u to some node v in a different branch of the tree or into another tree altogether. The condition on the numbers is

$$\text{pre}[v] < \text{post}[v] < \text{pre}[u] < \text{post}[u], \text{ or } [_v\ ]_v\ [_u\ ]_u\ ,$$

for a cross edge (u, v).

One may ask why there are no edges (u, v) going from a node u to a node in the forest that starts only after u is finished, meaning $\text{pre}[u] < \text{post}[u] < \text{pre}[v] < \text{post}[v]$. Well, in the course of explore(u) we will look along this edge, will find that v is new and start explore(v), and (u, v) will become a tree edge, so this situation is impossible.

Using the fact that intervals $[\text{pre}[u], \text{post}[u]]$ and $[\text{pre}[v], \text{post}[v]]$ are either nested or disjoint we note that the possibilities listed cover all possible cases. This means that the relations between pre and post numbers of u and v determine what the type of the edge (u, v) is. This leads to the picture in the box on page 89 in the book.

We now discuss how to recognize the four edge types in the course of running the algorithm instead of waiting till the end and looking at the numbers.

Edge (u, v) is a **tree edge (T)** if v is new when the edge is inspected.

Edge (u, v) is a **back edge (B)** if $\text{pre}[v] \leq \text{pre}[u] < \text{post}[u] \leq \text{post}[v]$. It is inspected during interval $[\text{pre}[u], \text{post}[u]]$, and this is at a time when v is active.

Edge (u, v) is a **cross edge (C)** if $\text{pre}[v] < \text{post}[v] < \text{pre}[u] < \text{post}[u]$. Thus, when we inspect this edge during explore(u), we'll see that v is finished and that $\text{pre}[v] < \text{pre}[u]$.

Edge (u, v) is a **forward edge (F)** if $\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{post}[u]$, and it is not a tree edge. Thus, when we inspect this edge during explore(u), we'll see that v is active or finished. It is impossible that v is active, since between times pre[v] and post[v] the procedure explore(u) is suspended, and the algorithm cannot look along edge (u, v). If inspecting edge (u, v) takes place between pre[u] and pre[v], node v is discovered to be new, so (u, v) becomes a tree edge, which is impossible. The only possibility is that v is finished when (u, v) is inspected and that $\text{pre}[u] < \text{pre}[v]$.

Algorithm 4 collects everything together.

We continue with some useful observations about the behavior of DFS, which are not in the book.

Claim 3: If all nodes of G are "unvisited", a call explore(v) finds all nodes u that can be reached from v along a directed path

$$v = v_0, v_1, \ldots, v_t = u, \text{ with } (v_{i-1}, v_i) \in E \text{ for } 1 \leq i \leq t.$$

The proof is exactly the same as the proof of Claim 1. Note, however, that in directed graphs there is no notion of a "connected component".

Claim 4: **("White path theorem").** In the general situation (some nodes may be "active" and some may be "finished", the others are "new") a call explore(v) finds a node u if and only if u can be reached from v along a path of "new" nodes (a "white path") at the time when the call starts. (In the DFS tree that contains v these nodes sit in the subtree below v.)

Proof: "⇒": Assume u is found during explore(v). Then there must be a nested sequence

Algorithm 4: (Depth First Search in a digraph)

explore(v):
  pre[v] $\leftarrow$ count++
  for all edges (v, w) do
    if w is new (no pre number): (v, w) is T-edge; explore(w)
    elseif w is active (has pre number but no post number): (v, w) is B-edge;
    else // w is finished (has post number) :
      if pre[w] < pre[v]: (v, w) is C-edge
      if pre[w] > pre[v]: (v, w) is F-edge
  post[v] $\leftarrow$ count++

DFS(G):
count $\leftarrow$ 1
mark all $v \in V$ as new (all pre and post numbers set to undefined)
for each $v \in V$ do if v is new (no pre number) then explore(v)

explore(v_0), explore(v_1), ..., explore(v_t),

with $v_0 = v$ and $v_t = u$, where explore(v_i) is called directly from explore(v_{i-1}), for $1 \leq i \leq t$. But then $v = v_0, v_1, \ldots, v_t = u$ is a path of tree edges, and $v_0, \ldots, v_t$ must all be new when explore(v) is called.
"$\Leftarrow$": : Assume $v = v_0, v_1, \ldots, v_t = u$ is a white path (without node repetition), when explore(v) is called. Suppose for a contradiction that explore(u) is not called. Let i be minimal so that explore(v_i) is not called, $1 \leq i \leq t$. During the execution of explore(v_{i-1}) the edge (v_{i-1}, v_i) is inspected, and v_i is found to be new. So explore(v_i) is called, a contradiction. □

3.3.1 Directed acyclic graphs

A path $v_0, v_1, \ldots, v_t$ (meaning that (v_{i-1}, v_i) is an edge for $1 \leq i \leq t$) is called a *cycle* if $v_0 = v_t$, it is called a *simple cycle* if in addition $v_0, v_1, \ldots, v_{t-1}$ are different. A graph has a cycle if and only if it has a simple cycle. Digraphs that do not have a cycle are a very important subclass of directed graphs, they are called

Directed **A**cyclic **G**raphs, or **DAG**s.

(An example is given in Fig. 3.8 in the book.) Given a digraph G, we wish to decide if it has a cycle. Here's the simple method: Do DFS on G and check if there is a back edge or not.

Acyclicity property: G has a cycle if and only if DFS finds a back edge.

Proof: It is easy to see that if there is a B edge (u, v), then there is a cycle: By definition, there must be a path $v = v_0, v_1, \ldots, v_t = u$ of tree edges, and together with edge $(u, v) = (v_t, v_0)$ we have the cycle $v_0, v_1, \ldots, v_t, v_0$.

For the other direction we present two proofs. One is in the book, and one uses post numbers.

Proof from the book: Assume $v_0, \ldots, v_t$ with $v_t = v_0$ is a simple cycle. Watch the calls to procedure explore. Let v_i be the first node on the cycle for which explore(v_i) is called. At this point in time the path

$$v_i, v_{i+1}, \ldots, v_t = v_0, v_1, \ldots, v_{i-1}$$

consists of new ("white") nodes. By Claim 4 (White Path Theorem) node v_{i-1} will become a descendant of v_i in the DFS tree, so there is a path of tree edges from v_i to v_{i-1}. By definition, edge (v_{i-1}, v_i) is a back edge.

Alternative proof via post numbers: We first note the following:

Claim 5: If (u, v) is a tree edge, a forward edge, or a cross edge, we have post$[v] <$ post$[u]$. (Look at the cases considered above.)

Now assume that there are no back edges. We show that then there is no cycle in G. By Claim 5 it is the case that along all edges in G (there are only T-, F-, and C-edges!) the post numbers decrease. Obviously, then there cannot be cycles. □

Thus, if we run DFS on a DAG, we don't have back edges, and along all edges the post numbers decrease. We can use this for constructing an interesting ordering of the nodes in G.

Definition. A *topological ordering* or a *linear ordering* of a DAG G is an ordering $v_1, \ldots, v_n$ of the nodes such that all edges run "from left to right", meaning that $(v_i, v_j) \in E$ implies $i < j$.

We can easily obtain a topological ordering for a DAG G: Just run DFS and arrange the nodes in order of decreasing post numbers.

Note that if $v_1, \ldots, v_n$ is a linear ordering then v_1 cannot have ingoing edges (so v_1 is a *source*) and v_n cannot have outgoing edges (so v_n is a *sink*). We obtain the last property on page 90 in the book: Every DAG has at least one source and at least one sink.