§1. Further Applications of Graph Traversal

This is a replacement section, with the correct algorithm for strong components!

In the following, assume $G = (V, E)$ is a digraph with $V = \{1, 2, \ldots, n\}$. Let $per[1..n]$ be an integer array that represents a permutation of V in the sense that $V = \{per[1], per[2], \ldots, per[n]\}$. This array can also be interpreted in other ways (e.g., a ranking of the vertices).

Topological Sort. One motivation is the so-called[1] PERT graphs: in their simplest form, these are DAG's where vertices represent activities. An edge $u{-}v \in E$ means that activity u must be performed before activity v. By transitivity, if there is a path from u to v, then u must be performed before v. A topological sort of such a graph amounts to a feasible order of execution of all these activities.

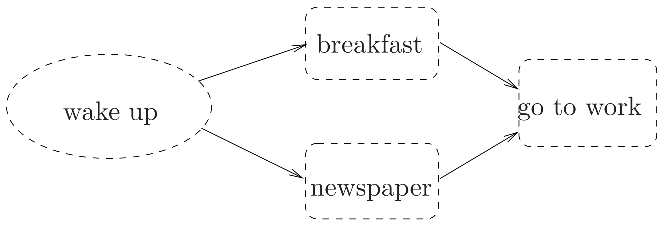

Figure 1: PERT graph

Let

$$(v_1, v_2, \ldots, v_n) \tag{1}$$

be a listing of the vertices in V. We call it a **topological sort** if every edge has the form $v_i{-}v_j$ where $i < j$. In other words, each edge points to the right, no edge points to the left. REMARK: if $(v_1, \ldots, v_n)$ is a topological sort, then $(v_n, v_{n-1}, \ldots, v_1)$ is called a **reverse topological sort**.

If an edges $u{-}v$ is intepreted as saying "activity u must precede activity v", then a topological sort give us one valid way for doing these activities (do activities $v_1, v_2, \ldots$ in this order).

Let us say that vertex v_i has **rank** i in the topological sort (1). Hence, we may represent this topological sort by a rank attribute array $Rank[1, \ldots, n]$, where $Rank[v_i] = i$ for all $v_i \in V$.

E.g., $(v_1, \ldots, v_n) = (v_3, v_1, v_2, v_4)$ in (1). The corresponding rank attribute array is $Rank[v_1, v_2, v_3, v_4] = [2, 3, 1, 4]$.

We use the DFS algorithm and the DFS Driver to compute the rank attribute array. First, we must initialize the $Rank$ array using the global initialization shell:

$$GLOBAL\_INIT(G) \equiv (\mathsf{for}\ v = 1\ \mathsf{to}\ n, Rank[v] \leftarrow -1).$$

Indeed, we need not use a separate color array: we simply interpret the $Rank$ of -1 as `unseen`. The idea is to use DFS(v) to assign a rank to v: but before we could assign a rank to v, we must (recursively) assign a larger rank to the vertices reachable from v. To do this, we use a global counter R that is initialized to n. Each time a vertex is to receive a rank, we use the current value of R, and then decrement R. So by the

[1]PERT stands for "Program Evaluation and Review Technique", a project management technique that was developed for the U.S. Navy's Polaris project (a submarine-launched ballistic missile program) in the 1950's. The graphs here are also called networks. PERT is closely related to the CriticalPath Method (CPM) developed around the same time.

time v receives its rank, all those vertices reachable from v would have received a larger rank. This idea can be implemented by programming the postvisit shell as follows:

$$POSTVISIT(v) \equiv (Rank[v] \leftarrow R; R \leftarrow R-1).$$

It is easy to prove the correctness of this procedure, provided the input graph is a DAG. But what can go wrong in this code if the input is not a DAG?

REMARKS: Note that the rank function is just as the order of v according to `lastTime`$[v]$. In our strong component algorithm below, we prefer to compute the inverse of Rank, i.e., an array $Per[1..n]$ such that $Per[i] = v$ iff $Rank[v] = i$. The topological sort (1) is then equal to $(Per[1], Per[2], \ldots, Per[n])$. We leave it as an easy exercise to modify the above code to computer Per directly.

Robust Topological Sort. Suppose we want a more robust algorithm that will detect an error in case the input is not a DAG. We need the following fact: *G is cyclic iff there exists a back edge in every DFS traversal*. This was shown in the previous section. To detect back edges, when we need two modifications. The previous solution is implicitly a 2-color scheme ($Rank[v] = -1$ if v is `unseen`, and otherwise v is `seen`). Now, we need to a 3-color scheme where

$$Rank[v] \begin{cases} = -1 & \text{if } v \text{ is } \texttt{unseen}, \\ = 0 & \text{if } v \text{ is } \texttt{seen}, \\ > 0 & \text{if } v \text{ is } \texttt{done}. \end{cases}$$

To implement this, we just need to program the shell for visiting a vertex:

$$VISIT(v, u) \equiv (Rank[v] \leftarrow 0.)$$

The second modification is to check for back edges. This can be done during previsits to a vertex v from u:

$$PREVISIT(v, u) \equiv (\mathsf{if}\ (Rank[v] = 0)\ \mathsf{then}\ ThrowException(\texttt{"Cycle detected"}))$$

Strong Components. Computing the components of digraphs is somewhat more subtle than the corresponding problem for bigraphs. In fact, at least three distinct algorithms for this problem are known. Here, we will develop the version based on "reverse graph search".

Let $G = (V, E)$ be a digraph where $V = \{1, \ldots, n\}$. For clarity, we also write "v_i" for $i \in V$. Let $Per[1..n]$ be an array that represents some permutation of the vertices, so $V = \{Per[1], Per[2], \ldots, Per[n]\}$. Let $DFS(i)$ denote the DFS algorithm starting from vertex i. Consider the following method to visit every vertex in G:

```
STRONG_COMPONENT_DRIVER($G, per$)
    INPUT: Digraph $G$ and permutation $Per[1..n]$.
    OUTPUT: A set of DFS Trees.
▷ Initialization
1.      For $i = 1, \ldots, n$, $color[i]$ =unseen.
▷ Main Loop
2.      For $i = 1, \ldots, n$,
3.          If ($color[Per[i]]$ =unseen)
4.              $DFS_1(Per[i])$    ◁ Outputs a DFS Tree
```

This program is the usual DFS Driver program, except that we use $Per[i]$ to determine the choice of the next vertex to visit, and it calls DFS_1, a variant of DFS. We assume that $DFS_1(i)$ will (1) change the color of every vertex that it visits, from `unseen` to `seen`, and (2) output the DFS tree rooted at i. If Per is correctly chosen, we want each DFS tree that is output to correspond to a strong component of G.

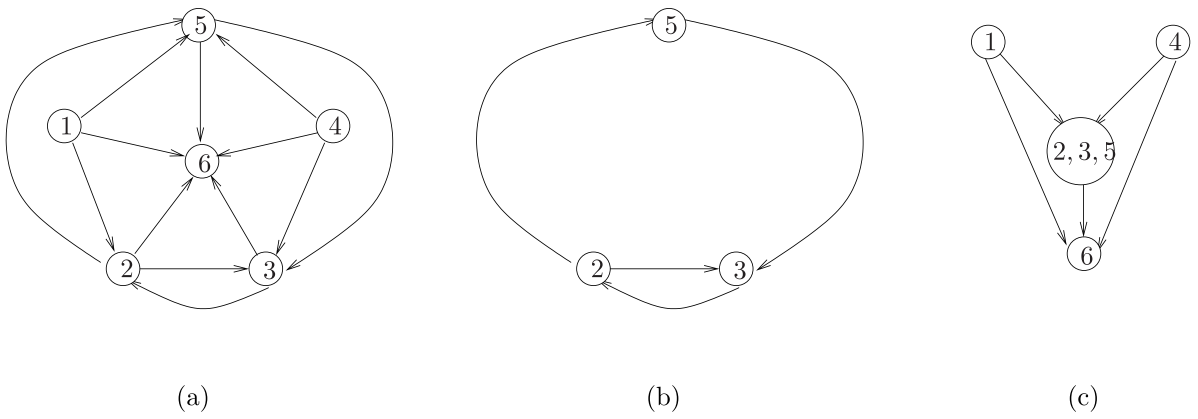

Figure 2: A digraph and its reduced graph.

First, let us see how the above subroutine will perform on the digraph G_6 in Figure 2(a). Let us also assume that the permutation is

$$\begin{aligned} Per[1,2,3,4,5,6] &= [6,3,5,2,1,4] \\ &= [v_6, v_3, v_5, v_2, v_1, v_4]. \end{aligned} \tag{2}$$

The output of Strong_Component_Driver will be the DFS trees for on the following sets of vertices (in this order):

$$C_1 = \{v_6\}, \quad C_2 = \{v_3, v_2, v_5\}, \quad C_3 = \{v_1\}, \quad C_4 = \{v_4\}.$$

Since these are the four strong components of G_6, the algorithm is correct. It is not not hard to see that there always exist "good permutations" for which the output is correct. Here is the formal definition of what this means:

A permutation $Per[1..n]$ is **good** if, for any two strong components C, C' of G, if there is a path from C to C', then the *first vertex of C' is listed before the first vertex of C'*.

It is easy to see that our Strong Component Driver will give the correct output iff the given permutation is good. But how do we get good permutations? Roughly speaking, they correspond to weak forms of "reverse topological sort" of G. There are two problems: topological sorting of G is not really meaningful when G is not a DAG. Second, good permutations requires some knowledge of the strong components which is what we want to compute in the first place! Nevertheless, let us go ahead and run the topological sort algorithm (not the robust version) on G. We may assume that the algorithm returns an array $Per[1..n]$ (the inverse of the $Rank[1..n]$). The next lemma shows that $Per[1..n]$ almost has the properties we want. For any set $C \subseteq V$, we first define

$$Rank[C] = \min\{i : Per[i] \in C\} = \min\{Rank[v] : v \in C\}$$

Lemma 1. *Let C, C' be two distinct strong components of G.*
(a) If $u_0 \in C$ is the first vertex in C that is seen, then $Rank[u_0] = Rank[C]$.
(b) If there is path from C to C' in the reduced graph of G, then $Rank[C] < Rank[C']$.

Proof. (a) By the Unseen Path Lemma, every node $v \in C$ will be a descendent of u_0 in the DFS tree. Hence, $Rank[u_0] \leq Rank[v]$, and the result follows since $Rank[C] = \min\{Rank[v] : v \in C\}$.
(b) Let u_0 be the first vertex in $C \cup C'$ which is seen. There are two possibilities: (1) Suppose $u_0 \in C$. By

part (a), $Rank[C] = Rank[u_0]$. Since there is a path from C to C', an application of the Unseen Path Lemma says that every vertex in C' will be descendents of u_0. Let u_1 be the first vertex of C' that is seen. Since u_1 is a descendent of u_0, $Rank[u_0] < Rank[u_1]$. By part(a), $Rank[u_1] = Rank[C']$. Thus $Rank[C] < Rank[C']$.
(2) Suppose $u_0 \in C'$. Since there is no path from u_0 to C, we would have assigned a rank to u_0 before any node in C is seen. Thus, $Rank[C_0] < Rank[u_0]$. But $Rank[u_0] = Rank[C']$. **Q.E.D.**

This lemma implies that, in the reverse "topological sort" ordering,

$$[Per[n], Per[n-1], \ldots, Per[1]] \tag{3}$$

if there is path from C to C', then the *last* vertex of C' in this list appears *before* the *last* vertex of C in this list. So this is not quite good.

We use another insight: consider the reverse graph G^{rev} (i.e., $u{-}v$ is an edge of G iff $v{-}u$ is an edge of G^{rev}). It is easy to see that C is a strong component of G^{rev} iff C is a strong component of G. However, there is a path from C to C' in G^{rev} iff there is a path from C' to C in G.

LEMMA 2. *If $Per[1..n]$ is the result of running topological sort on G^{rev} then Per is a good permutation for G.*

Proof. Let C, C' be two components of G and there is a path from C to C' in G. Then there is a path from C' to C in the reverse graph. According to the above, the last vertex of C is listed before the last vertex of C' in (3). That means that the first vertex of C is listed after the first vertex of C' in the listing $[Per[1], Per[2], \ldots, Per[n]]$. This is good. **Q.E.D.**

We now have the complete algorithm:

```
STRONG_COMPONENT_ALGORITHM($G$)
    INPUT: Digraph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$.
    OUTPUT: A list of strong components of $G$.
1.      Compute the reverse graph $G^{rev}$.
2.      Call topological sort on $G^{rev}$.
            This returns a permutation array $Per[1..n]$.
3.      Call STRONG_COMPONENT_DRIVER($G, Per$)
```

Remarks. Tarjan [4] was the first to give a linear time algorithm for strong components. R. Kosaraju and M. Sharir independently discovered the reverse graph search method described here. The reverse graph search is conceptually elegant. But since it requires two passes over the graph input, it is slower in practice than the direct method of Tarjan. Yet a third method was discovered by Gabow in 1999. For further discussion of this problem, including history, we refer to Sedgewick [3].

References

[1] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* North Holland, New York, 1976.

[2] T. H. Corman, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press and McGraw-Hill Book Company, Cambridge, Massachusetts and New York, second edition, 2001.

[3] R. Sedgewick. *Algorithms in C: Part 5, Graph Algorithms.* Addison-Wesley, Boston, MA, 3rd edition edition, 2002.

[4] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Computing*, 1(2), 1972.