Note to readers:
Please ignore these sidenotes; they're just hints to myself for preparing the index, and they're often flaky!

KNUTH

THE ART OF COMPUTER PROGRAMMING

VOLUME 4 PRE-FASCICLE 12A

COMPONENTS AND TRAVERSAL
(ridiculously preliminary draft)

DONALD E. KNUTH *Stanford University*

ADDISON–WESLEY

February 8, 2023

Internet
Stanford GraphBase
MMIX

Internet page http://www-cs-faculty.stanford.edu/~knuth/taocp.html contains current information about this book and related books.

See also http://www-cs-faculty.stanford.edu/~knuth/sgb.html for information about *The Stanford GraphBase*, including downloadable software for dealing with the graphs used in many of the examples in Chapter 7.

See also http://www-cs-faculty.stanford.edu/~knuth/mmixware.html for downloadable software to simulate the MMIX computer.

Zeroth printing (revision -86), 08 February 2023

BARRY
Internet

PREFACE

But that is not my point.
I have totally forgotten my point.
— DAVE BARRY (2012)

THIS BOOKLET contains draft material that I'm circulating to experts in the field, in hopes that they can help remove its most egregious errors before too many other people see it. I am also, however, posting it on the Internet for courageous and/or random readers who don't mind the risk of reading a few pages that have not yet reached a very mature state. *Beware:* This material has not yet been proofread as thoroughly as the manuscripts of Volumes 1, 2, 3, and 4A were at the time of their first printings. And alas, those carefully-checked volumes were subsequently found to contain thousands of mistakes.

Given this caveat, I hope that my errors this time will not be so numerous and/or obtrusive that you will be discouraged from reading the material carefully. I did try to make the text both interesting and authoritative, as far as it goes. But the field is vast; I cannot hope to have surrounded it enough to corral it completely. So I beg you to let me know about any deficiencies that you discover.

To put the material in context, this portion of fascicle 12 previews parts of Section 7.4.1 of Chapter 7 of *The Art of Computer Programming*, entitled "Components and traversal." The first part, Section 7.4.1.1, is about "Union-find algorithms," and it's barely started. The next part, Section 7.4.1.2, is about "Depth-first search"; I've drafted quite a bit more about that, especially with respect to Tarjan's algorithm for strong components. And there will be at least one more part. But I haven't had time to write much more of 7.4.1 yet — not even this preface!

At present I've only got a few small scraps of copy that I'm occasionally putting into my computer, on days when possibly relevant material occurs to me. Thus almost everything you see here is more or less a place-holder for better things that hopefully will come later. Some day, however, I hope that I'll no longer have to apologize for what is now just a bunch of crumbs.

* * *

The explosion of research in computer science since the 1970s has meant that I cannot hope to be aware of all the important ideas in this field. I've tried my best to get the story right, yet I fear that in many respects I'm woefully ignorant. So I beg expert readers to steer me in appropriate directions.

Please look, for example, at the exercises that I've classed as research problems (rated with difficulty level 46 or higher), namely exercises 38, . . . ; I've also implicitly mentioned or posed additional unsolved questions in the answers to exercises 61, ... Are those problems still open? Please inform me if you know of a solution to any of these intriguing questions. And of course if no solution is known today but you do make progress on any of them in the future, I hope you'll let me know.

I urgently need your help also with respect to some exercises that I made up as I was preparing this material. I certainly don't like to receive credit for things that have already been published by others, and most of these results are quite natural "fruits" that were just waiting to be "plucked." Therefore please tell me if you know who deserves to be credited, with respect to the ideas found in exercises 25–27, 30–32, 37, 51, 56, 80, 84, 85, 92, 93, 95, ... Furthermore I've credited exercises . . . to unpublished work of . . . . Have any of those results ever appeared in print, to your knowledge?

Janson
Stappers
Tarjan
Knuth
HAUPTMAN

* * *

Special thanks are due to Svante Janson, Filip Stappers, Bob Tarjan, and . . . for their detailed comments on my early attempts at exposition, as well as to numerous other correspondents who have contributed crucial corrections.

* * *

I happily offer a "finder's fee" of $2.56 for each error in this draft when it is first reported to me, whether that error be typographical, technical, or historical. The same reward holds for items that I forgot to put in the index. And valuable suggestions for improvements to the text are worth 32¢ each. (Furthermore, if you find a better solution to an exercise, I'll actually do my best to give you immortal glory, by publishing your name in the eventual book:–)

Cross references to yet-unwritten material sometimes appear as '00'; this impossible value is a placeholder for the actual numbers to be supplied later.

Happy reading!

Stanford, California D. E. K.
99 Umbruary 2021

For all such items, my procedure is the same:
I write them down — and then write them up.
— DON HAUPTMAN (2016)

SYLVESTER
PETERSEN
Euler
EVEN
reachable
equivalence relation
connected components

7.4. GRAPH ALGORITHMS

Casting about, as I lay awake in bed one night, to discover some means of conveying an intelligible conception of the objects of modern algebra to a mixed society, mainly composed of physicists, chemists and biologists, . . . to which I stood engaged to give some account of my recent researches in this subject of my predilection, . . . I was agreeably surprised to find, of a sudden, distinctly pictured on my mental retina a chemico-graphical image . . . every bond or connecting line in the graph passes through two atoms.

— J. J. SYLVESTER, *American Journal of Mathematics* (1878)

One can give the problem a geometric form, in which x_1, x_2, . . . , x_n are represented by arbitrary points in the plane, while each factor $x_m - x_p$ is portrayed by an arbitrary connecting line between x_m and x_p. . . . English authors have introduced the name graph *for figures like this; I shall retain that name.*

— JULIUS PETERSEN, *Acta Mathematica* (1891)

Although algorithmic graph theory was started by Euler, if not earlier, its development in the last ten years has been dramatic and revolutionary.

— SHIMON EVEN, *Graph Algorithms* (1979)

GRAPHS HAVE tremendous variety, and can be viewed in many different ways. Therefore we turn now to the study of algorithms that discover structural properties of a given graph. We'll see how to take advantage of various special attributes that a graph might have. For example, many graphs can be decomposed into parts that are either independent or fit together nicely in a simple way that allows problems to be solved more quickly.

7.4.1. Components and traversal

In previous chapters we've already seen many instances of important techniques that have been developed to work with graphs efficiently inside a computer. Now it's time to examine those basic methods in more detail.

7.4.1.1. Union-find algorithms. Let's begin with connectedness, the simplest structural concept of all. A graph is *connected* if there's a path between any two of its vertices. Thus, if we divide its vertices into two nonempty parts, there's always at least one edge that connects them by having one vertex in each part; every path from one part to the other contains such an edge.

We say that vertex w is *reachable* from vertex v in graph G if $w = v$ or there's a path in G from v to w. Reachability is clearly an equivalence relation; and the equivalence classes of this relation are the *connected components* of G.

Long ago, in Algorithm 2.3.3E, we saw that it's not difficult to determine whether or not two elements of a set are equivalent to each other as the result of a given collection of individual equivalences. Such individual equivalences define the edges of a graph; so Algorithm 2.3.3E can be regarded as an algorithm to determine the connected components of a given graph.

There are, in fact, several interesting ways to improve that algorithm.

(Now I'll go into the story of "union-find" algorithms. That will be fun — it's one of my favorite chapters in the early development of computer science!)

EXERCISES

1. [*05*] Is the one-vertex graph K_1 connected?

99. [*00*] this is a temporary dummy exercise

999. [*M00*] this is a temporary exercise (for dummies)

In every affair, consider what precedes and what follows, before acting.
— EPICTETUS, *Discourses* III.xv (C. A.D. 110)

And in the lowest deep a lower deep
Still threatning to devour me opens wide.
— JOHN MILTON, *Paradise Lost* IV (1667)

EPICTETUS
MILTON
depth-first search–worm
bipartite
SGB format
stack
sentinel

7.4.1.2. Depth-first search. Another fundamental problem is to visit every vertex and every arc of a graph. It's analogous to the problem of traversing a forest; see, for example, the "worm" in 7.2.1.6–(3). But graphs are trickier because they can contain cycles. We saw a simple example related to depth-first search near the beginning of Chapter 7: Algorithm 7B tests whether or not a given graph is bipartite, by assigning one of two colors to every vertex unless that task turns out to be impossible.

In the previous section we dealt with situations where we learn one edge of a graph at a time, occasionally making a query to see if the graph-so-far has a particular property. We didn't actually need to remember every edge of that graph.

But now our outlook changes. We're given a prespecified digraph, and we want to walk through it. An undirected graph or multigraph is the special case of a digraph in which every edge is represented by two arcs, one in each direction.

We will assume that the digraph g is represented in SGB format, as discussed just before Algorithm 7B. That is, there are $n = \mathtt{N}(g)$ vertices v, each having fields $\mathtt{NAME}(v)$ and $\mathtt{ARCS}(v)$. Those vertices are $v_0 + k$ for $0 \le k < n$, where $v_0 = \mathtt{VERTICES}(g)$. There are $m = \mathtt{M}(g)$ arcs, where the arcs from v begin at $\mathtt{ARCS}(v)$. Each arc a has fields $\mathtt{TIP}(a)$ and $\mathtt{NEXT}(a)$. If the arcs from v are $v \longrightarrow w_1$, $v \longrightarrow w_2$, $\ldots$, $v \longrightarrow w_k$, we have $w_1 = \mathtt{TIP}(a_1)$, $w_2 = \mathtt{TIP}(a_2)$, $\ldots$, and $w_k = \mathtt{TIP}(a_k)$, where $a_1 = \mathtt{ARCS}(v)$, $a_2 = \mathtt{NEXT}(a_1)$, $\ldots$, $a_k = \mathtt{NEXT}(a_{k-1})$, and $\mathtt{NEXT}(a_k) = \Lambda$. (If v has no successors, $k = 0$ and $\mathtt{ARCS}(v) = \Lambda$.)

A straightforward way to walk through such a digraph is to use a stack to hold the vertices that we've seen but whose arcs we haven't yet seen. The only complication is that we don't want to put a vertex on that stack more than once.

A neat trick allows us to achieve our goal efficiently by letting the stack links themselves tell us which vertices we've seen:

Algorithm Q (*Quick digraph search*). Given a digraph g in SGB format, this algorithm marks every vertex and visits every arc of g. It uses a utility field in each vertex, called MARK, and an auxiliary stack pointed to by s.

Q1. [Initialize.] Set $\mathtt{MARK}(w) \leftarrow \Lambda$ for $\mathtt{VERTICES}(g) \le w < \mathtt{SENT}$, until eventually $w = \mathtt{SENT}$, where $\mathtt{SENT} = \mathtt{VERTICES}(g) + \mathtt{N}(g)$ is a "sentinel."

Q2. [Done?] (Every vertex $v \ge w$ has now been marked, as well as all vertices reachable from v.) Terminate if $w = \mathtt{VERTICES}(g)$. Otherwise set $w \leftarrow w - 1$, and repeat this step if $\mathtt{MARK}(w) \ne \Lambda$.

Q3. [Begin to explore from v.] Set $v \leftarrow w$, $\mathtt{MARK}(v) \leftarrow s \leftarrow \mathtt{SENT}$, $a \leftarrow \mathtt{ARCS}(v)$.

Q4. [Visit the next arc from v.] Go to Q6 if $a = \Lambda$. Otherwise set $u \leftarrow \mathtt{TIP}(a)$ and $a \leftarrow \mathtt{NEXT}(a)$.

Q5. [Mark and stack u, unless it's already marked.] If $\texttt{MARK}(u) = \Lambda$, set $\texttt{MARK}(u) \leftarrow s$ and $s \leftarrow u$. Return to Q4.

Q6. [Pop the stack.] Go back to Q2 if $s = \texttt{SENT}$. Otherwise set $v \leftarrow s$, $s \leftarrow \texttt{MARK}(v)$, $a \leftarrow \texttt{ARCS}(v)$, and return to Q4. ▌

Consider, for example, the digraph on vertices $\{0, 1, \ldots, 9\}$ whose 21 arcs are based on the first 42 digits of π:

$$\begin{array}{l} 3 \longrightarrow 1,\ 4 \longrightarrow 1,\ 5 \longrightarrow 9,\ 2 \longrightarrow 6,\ 5 \longrightarrow 3,\ 5 \longrightarrow 8,\ 9 \longrightarrow 7,\\ 9 \longrightarrow 3,\ 2 \longrightarrow 3,\ 8 \longrightarrow 4,\ 6 \longrightarrow 2,\ 6 \longrightarrow 4,\ 3 \longrightarrow 3,\ 8 \longrightarrow 3,\\ 2 \longrightarrow 7,\ 9 \longrightarrow 5,\ 0 \longrightarrow 2,\ 8 \longrightarrow 8,\ 4 \longrightarrow 1,\ 9 \longrightarrow 7,\ 1 \longrightarrow 6. \end{array} \tag{1}$$

Algorithm Q visits these arcs by seeing respectively

$$\begin{array}{l} 9 \longrightarrow 7,\ 3,\ 5,\ (7);\\ 5 \longrightarrow (9),\ (3),\ 8;\\ 8 \longrightarrow 4,\ (3),\ (8);\\ 4 \longrightarrow 1,\ (1); \end{array} \tag{2}$$

and so on, where the parentheses in '(7)' indicate that 7 was not stacked because it had been seen before. (See exercise 1.)

Algorithm Q is refreshingly quick. In fact, exercise 5 shows that it accesses memory fewer than $3m + 5n$ times, when the digraph has m arcs and n vertices. But unfortunately the order in which it traverses g is not the best, from the standpoint of graph algorithms in general. There's another way to do the traversal, called *depth-first search*, which turns out to be far more useful, because it can be extended to help solve a wide variety of more difficult problems on graphs.

Depth-first search is superficially similar to Algorithm Q, yet it's significantly better in many applications, because it implicitly constructs an *oriented spanning forest* of the given digraph. Furthermore, it visits the vertices in a recursive order that corresponds to preorder on that forest. (By contrast, the order of visitation by Algorithm Q doesn't have a clean recursive characterization.)

One good way to visualize depth-first traversal is to imagine that we're playing an adventure game in which we're exploring a cave. The vertices of the digraph are the rooms of the cave, and the arcs are one-way passages from one room to another. We haven't been given a map of the cave; our goal is to pry into every nook and cranny, systematically feeling our way.

Algorithm Q "cheats" when it says $v \leftarrow s$ in step Q6: Cave games rarely allow us to teleport ourselves from one room of a cave to another. A genuine depth-first search moves only through legitimate passageways, except when it enters or leaves the cave, although we do allow it to retrace its steps in the one-way corridors. (In a real game we can't unfall off a cliff.)

Let us therefore go depth-first without lowering our standards.

Algorithm D (*Depth-first search*). Given a digraph g in SGB format, this algorithm traverses g and uses a subset of g's arcs to construct an oriented spanning forest. (The arcs in this forest lead *away* from the roots.) Every

π
oriented spanning forest
spanning forest
preorder
adventure game
game
cave

vertex is assumed to contain utility fields called PARENT, ARC, PRE, and POST. After termination, PARENT(v) will be v's parent in the forest, or the sentinel value SENT if v is a root; PRE(v) and POST(v) will be the indices of v in preorder and postorder of the forest. We use integer variables p and q.

postorder
preorder
postorder

D1. [Initialize.] Set PARENT$(w) \leftarrow \Lambda$ for VERTICES$(g) \le w <$ SENT, until eventually $w =$ SENT, where SENT $=$ VERTICES$(g) +$ N(g). Also set $p \leftarrow q \leftarrow 0$.

D2. [Done?] (Every vertex $v \ge w$ is now in the explored forest, as well as all vertices reachable from v.) Terminate if $w =$ VERTICES(g). Otherwise set $w \leftarrow w - 1$, and repeat this step if PARENT$(w) \ne \Lambda$.

D3. [Begin to explore from v.] Set $v \leftarrow w$, PARENT$(v) \leftarrow$ SENT, $p \leftarrow p + 1$, PRE$(v) \leftarrow p$, and $a \leftarrow$ ARCS(v). (Vertex v is pth in preorder of the forest.)

D4. [Done with v?] If $a = \Lambda$, set $q \leftarrow q + 1$, POST$(v) \leftarrow q$, $v \leftarrow$ PARENT(v), and go to D8. (Vertex v is qth in postorder of the forest.)

D5. [Visit the next arc, $v \longrightarrow u$.] Set $u \leftarrow$ TIP(a) and $a \leftarrow$ NEXT(a).

D6. [If u isn't new, continue at v.] If PARENT$(u) \ne \Lambda$, return to D4.

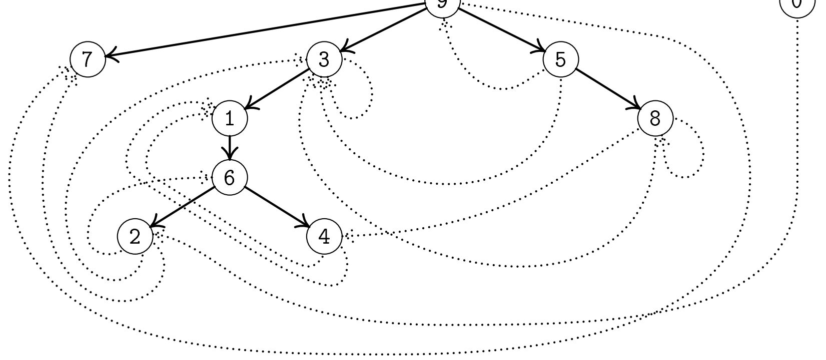
D7. [Move to u.] Set PARENT$(u) \leftarrow v$, ARC$(v) \leftarrow a$, $v \leftarrow u$, $p \leftarrow p + 1$, PRE$(v) \leftarrow p$, $a \leftarrow$ ARCS(v), and go to D4.

D8. [Done with tree?] If $v =$ SENT, go back to D2. Otherwise set $a \leftarrow$ ARC(v) and go to D4. ▮

Algorithm D essentially mimics the behavior of a well-organized spelunker. There are three main cases, depending on whether the current position v moves to a newly discovered child (step D7), moves back to a parent (step D8), or stays put (goes from D6 to D4). Step D4 is the main switch: When presented with digraph (1), arc a in that step takes the successive values

$$\begin{aligned}&9 \longrightarrow 7;\ \Lambda\ (\text{end }7);\ 9 \longrightarrow 3;\ 3 \longrightarrow 1;\ 1 \longrightarrow 6;\ 6 \longrightarrow 2;\ 2 \longrightarrow 6;\ 2 \longrightarrow 3;\ 2 \longrightarrow 7;\\ &\Lambda\ (\text{end }2);\ 6 \longrightarrow 4;\ 4 \longrightarrow 1;\ 4 \longrightarrow 1;\ \Lambda\ (\text{end }4);\ \Lambda\ (\text{end }6);\ \Lambda\ (\text{end }1);\end{aligned} \qquad (3)$$

and so on. Eventually the oriented forest



(4)

is constructed from the arcs $v \longrightarrow u$ when step D6 discovers a new vertex u.

The arcs $v \longrightarrow u$ that Algorithm D encounters when vertex u has already been seen do not form part of the oriented forest. But they're still in the digraph g, so they are shown as dotted arcs in (4). Nontree arcs come in four flavors:

- "Back arcs" go from a vertex to one of its proper ancestors.
- "Forward arcs" go from a vertex to one of its proper descendants.
- "Loops" go from a vertex to itself.
- "Cross arcs" go from a vertex to a nonancestor, nondescendant.

Can you classify all 13 of the dotted arcs in (4)? (See exercise 13.)

The nontree arcs can in fact be easily classified by looking at `PRE` and `POST`:

Theorem D. *After a depth-first search, the nontree arc* $v \longrightarrow u$ *is*

a back arc if and only if $\mathtt{PRE}(v) > \mathtt{PRE}(u)$ *and* $\mathtt{POST}(v) < \mathtt{POST}(u)$*;*
a forward arc if and only if $\mathtt{PRE}(v) < \mathtt{PRE}(u)$ *and* $\mathtt{POST}(v) > \mathtt{POST}(u)$*;*
a cross arc if and only if $\mathtt{PRE}(v) > \mathtt{PRE}(u)$ *and* $\mathtt{POST}(v) > \mathtt{POST}(u)$*;*

the case $\mathtt{PRE}(v) < \mathtt{PRE}(u)$ *and* $\mathtt{POST}(v) < \mathtt{POST}(u)$ *is impossible.*

Proof. Every vertex v begins "unseen". Then $\mathtt{PRE}(v)$ is set, and v becomes "active" while Algorithm D is visiting every unseen vertex reachable from v. Finally, after all of v's arcs have been examined, v has been "explored" and $\mathtt{POST}(v)$ is set. (See exercise 7.)

Exercise 2.3.2–20 shows that ancestor-descendant relations are characterized by preorder and postorder. Exercise 18 completes the proof. ▌

The data structure provided by Algorithm D, which chooses some of the arcs of g to be the tree arcs of an oriented spanning forest, is called the *depth-first jungle* of g. Notice that the jungle is not an inherent property of the graph g itself; isomorphic graphs can have drastically different jungles, because the order in which arcs from a given vertex are examined has a significant effect. The order in which vertices are internally numbered and surveyed in step D2 is also important; for example, the first vertex to be explored is always a root. (See exercise 22.) But we will see that the choice of jungle makes little or no difference to the efficiency of more elaborate algorithms that make use of this structure.

Strong components. In any digraph g, the relation "u is reachable from v and v is reachable from u" is clearly an equivalence relation. The equivalence classes of this relation are called the *strong components* of g, and a digraph with only one strong component is called *strongly connected.*

Robert Tarjan discovered that the strong components of any given digraph can actually be discovered "on the fly" while we're doing a depth-first search,* by making only modest changes to Algorithm D.

* Author's note: Over the years I've often been asked, "What is your favorite algorithm?" Immediately I think to myself, "Uff, how I hate to be asked to choose favorites." But then I realize that, in this particular case, the answer is quite clear: Tarjan's algorithm for strong components is, without doubt, the algorithm that I love best. When I learned of this elegant procedure in 1972, I understood for the first time that data structures can also be "deep."

Back arcs
ancestors
Forward arcs
descendants
Loops
Cross arcs
unseen
active
reachable
explored
depth-first jungle
jungle
strong components–
reachable
equivalence relation
strongly connected
Tarjan
Knuth
algorithm, favorite

In divers Liquors the little surfaces of the component particles are smooth and slippery.

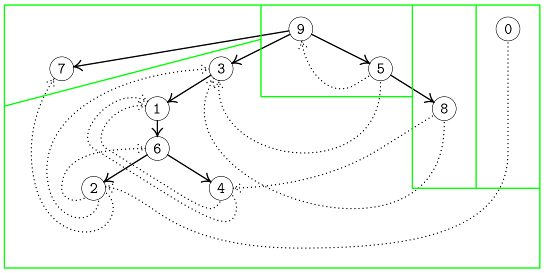
— ROBERT BOYLE, *Certain Physiological Essays* (1661)

It seems like a circular situation;
in order to compute the lowpoint we need to identify the components,
and in order to find the components we need the lowpoint.
However, Tarjan found a way out.

— SHIMON EVEN, *Graph Algorithms* (1979)

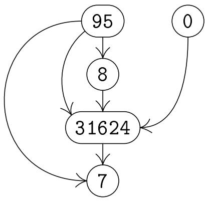
supervertices
acyclic
BOYLE
circular situation
Tarjan
EVEN
oriented cycle
shrink
directed acyclic graph
sink

Let's try to see why that's possible, by looking for example at the jungle (4) of the digraph (1) and examining its strong components. Here is (4) again, but the forward arcs $9 \longrightarrow 7$ and the loops $3 \longrightarrow 3$, $8 \longrightarrow 8$ are not shown:



(5)

(Forward arcs and loops have no effect on strong components, because they can be deleted without affecting reachability.) This jungle has been partitioned into five regions, containing "supervertices" $\{7, 31624, 95, 8, 0\}$; and it's easy to see that the vertices within each region are strongly connected. For example, each of the five vertices $\{3, 1, 6, 2, 4\}$ is reachable from the other four. Furthermore, these regions are in fact the strong components of (1)! For if we shrink the supervertices to single points, the resulting digraph



(6)

is acyclic. We can't go from one supervertex to another and back again.

In general, there's a conceptually simple way to find the strong components of *any* digraph: The vertices of an oriented cycle are strongly connected; so they're equivalent, and we can shrink them to a point. Let's do that repeatedly until the digraph no longer has any oriented cycles. The remaining supervertices are the (shrunken) strong components. In (1), for example, we could shrink $4 \longrightarrow 1 \longrightarrow 6 \longrightarrow 4$ into 416; then $5 \longrightarrow 9 \longrightarrow 5$ into 59; then $2 \longrightarrow 3 \longrightarrow 416 \longrightarrow 2$ into 23416; voilà, we've got (6).

Notice now that every nonempty directed acyclic graph has at least one "sink" vertex, a vertex with out-degree 0. Therefore, by the shrinking mechanism just described, *every nonempty digraph has at least one "sink" strong component*, a strong component without arcs to any other strong components.

Let S be a sink strong component of the digraph g. Since there's no escape from S, we can remove any arcs of g that lead to a vertex of S; after all, arcs into a "black hole" have no influence on g's strong components. Consequently *the strong components of g are S together with the strong components of $g \setminus S$.*

Good — there's a decent chance that we might discover an efficient method, based on the notion of repeatedly identifying and removing sink components until all components have been found. Depth-first search has the convenient property that it learns about g's vertices and arcs one by one; we essentially want to extend it so that it can readily recognize (and remove) sink components.

Recall that all vertices are initially unseen when depth-first search begins. Then they become active, one by one, in preorder of the depth-first forest. Furthermore, a vertex remains active until the moment when we know that we've seen all of its outgoing arcs. In other words, every vertex goes through three stages as we perform depth-first search: It's unseen and inactive, then seen and active, then seen and inactive. We're going to extend that process by also removing vertices from the graph and calling them "settled," as soon as we know that they're part of a sink component with respect to unsettled vertices.

Thus every vertex will go through *four* stages as we search for strong components: First it's unseen, inactive, unsettled; then it's seen, active, unsettled; then it's seen, inactive, unsettled; finally it's seen, inactive, settled. We know from Algorithm D how to handle the first transitions. The only thing missing is an appropriate way to settle an unsettled vertex.

Let U be the set of all currently seen but unsettled vertices. We can list them

$$u_1,\ u_2,\ \ldots,\ u_t, \quad \text{where} \quad \texttt{PRE}(u_1) < \texttt{PRE}(u_2) < \cdots < \texttt{PRE}(u_t), \tag{7}$$

according to the order in which we've seen them. Some of those vertices might be active, while others might be inactive. If the active ones are $u_{j_1}, u_{j_2}, \ldots, u_{j_a}$, with $j_1 < j_2 < \cdots < j_a$, the depth-first strategy of Algorithm D tells us that $\texttt{PARENT}(u_{j_1}) = \texttt{SENT}$, $\texttt{PARENT}(u_{j_2}) = u_{j_1}, \ldots, \texttt{PARENT}(u_{j_a}) = u_{j_{a-1}}$.

We've already seen every arc that goes out from the inactive vertices of U. For all we know, however, there may still be lots and lots of yet-unseen arcs, leading from any or all of the active vertices to any vertices whatsoever.

A nice structure does turn out to be present:

Lemma S. *Let $\hat{g}$ be the digraph whose vertices U are listed in (7), and whose arcs are those seen so far from U into U during a depth-first search for the strong components of g. Assume that $t \geq 1$. Also let $S_1, \ldots, S_k$ be the strong components of $\hat{g}$. Then $\hat{g}$ contains $k-1$ arcs*

$$u'_1 \longrightarrow u''_2,\ u'_2 \longrightarrow u''_3,\ \ldots,\ u'_{k-1} \longrightarrow u''_k, \quad \textit{with } u'_j, u''_j \in S_j, \tag{8}$$

that link those components into a path. All other arcs of $\hat{g}$ lie within the individual strong components S_j, which form consecutive intervals of the list (7). The first element of each S_j, called its "leader," is always an active vertex.

unseen
active
seen
settled
sink component
leader

Proof. The lemma holds initially, when $\hat{g}$ is a single isolated active vertex, because an isolated vertex u_1 is a strong component S_1 all by itself. We shall prove that the lemma remains true as the computation proceeds.

Let $u = u_{j_a}$ be the active vertex of largest preorder rank. Then $u \in S_k$, and the algorithm looks for the next unseen arc from u.

If there is no such arc, u becomes inactive. The lemma remains true if u isn't the first element of S_k. Otherwise, however, S_k has no remaining active vertices, so the algorithm knows that S_k is a sink component; vertices $\{u_{j_a}, \ldots, u_t\}$ become settled, and the lemma remains true with $t \leftarrow j_a - 1$, $k \leftarrow k - 1$.

Suppose the next arc from u is $u \longrightarrow v$. We ignore it if v is already settled, or if $v \in S_k$. On the other hand if $v \in S_j$ for $j < k$, we combine $S_j \cup \cdots \cup S_k$ into a single strong component and set $k \leftarrow j$; the lemma remains true.

Otherwise v is an unseen vertex, which becomes seen and active. The lemma remains true with $u_{t+1} \leftarrow v$, $u'_k \leftarrow u$, $u''_{k+1} \leftarrow v$, $t \leftarrow t+1$, $k \leftarrow k+1$. ▮

isolated vertex
Tarjan
LOW
downpath
tree arc
matures
nontree arc
leader

Corollary S. *Suppose digraph g has s strong components $\{C_1, \ldots, C_s\}$, and let v_j be the vertex of C_j that is earliest in preorder of g's depth-first forest. Order the components so that* `POST`$(v_1) < \cdots <$ `POST`(v_s). *Then the vertices of C_j are*

$$\operatorname{desc}(v_j) \setminus \bigl(\operatorname{desc}(v_1) \cup \cdots \cup \operatorname{desc}(v_{j-1})\bigr), \qquad \textit{for } 1 \le j \le s, \tag{9}$$

where $\operatorname{desc}(v)$ *denotes the descendants of v in the forest.* ▮

(Hence the strong components of g are the *connected* components of the digraph obtained by deleting the arcs to $v_1, \ldots, v_s$ from g's depth-first spanning forest.)

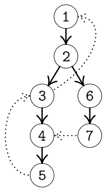
The structure guaranteed by Lemma S almost gives us the efficient algorithm that we want. But one ingredient is still missing, namely a quick way to deal with the boundaries between the current candidates S_j for strong components. Tarjan discovered that an extra field called `LOW` in each vertex is able to do what we need.

The definition of `LOW` is somewhat tricky and technical: When v is seen but unsettled, `LOW`(v) is the smallest preorder rank of an unsettled vertex w for which we've seen a "downpath" from v to w, where a downpath is an oriented path

$$v = v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_r = w, \qquad \text{for some } r \ge 0, \tag{10}$$

such that $v_j \longrightarrow v_{j+1}$ "matured" before $v_{j-1} \longrightarrow v_j$ for $0 < j < r$. A tree arc $v \longrightarrow u$ matures when u becomes inactive; a nontree arc matures when we first see it.

Consider, for example, the depth-first search that leads to the jungle shown here. At the time vertex 5 has just become seen, we have `LOW(1)` $= 1, \ldots,$ `LOW(5)` $= 5$. Then the arc $5 \longrightarrow 3$ is a downpath with $r = 1$, so `LOW(5)` $\leftarrow 3$. After 5 becomes inactive, $4 \longrightarrow 5$ matures; hence `LOW(4)` $\leftarrow 3$ because of the downpath $4 \longrightarrow 5 \longrightarrow 3$ with $r = 2$. Soon `LOW(3)` $\leftarrow 1$; and so on (see exercise 45). The definition ensures that at most one `LOW` value changes at every step of the depth-first process.



Theorem T. *When vertex v becomes inactive during a depth-first search, v is the leader of the current sink component if and only if* `LOW`$(v) =$ `PRE`(v).

Proof. (This fact is Tarjan's "secret sauce.") See exercise 50. ▮

Tarjan
leader
SENT

OK, we're ready now for the nitty-gritty details of an efficient algorithm. Algorithm T maintains a stack whose top is SINK, linked together in LINK fields, which holds all the inactive vertices that aren't yet settled. It also gives every vertex a REP field for the equivalence class representatives: Vertices v and w are strongly connected if and only if $\mathtt{REP}(v) = \mathtt{REP}(w)$ at termination.

Tarjan noticed in 2021 that we can save time and space by encoding PRE and LOW in the existing LINK and REP fields, because we need only know whether or not LOW = PRE. While vertex v is active, we shall therefore let

$$\mathtt{LINK}(v) = \begin{cases} \mathtt{SENT}, & \text{if } \mathtt{LOW}(v) = \mathtt{PRE}(v); \\ \Lambda, & \text{if } \mathtt{LOW}(v) \neq \mathtt{PRE}(v). \end{cases} \qquad (11)$$

Moreover, after v has been seen, we shall let

$$\mathtt{REP}(v) = \begin{cases} \mathtt{LOW}(v), & \text{if } v \text{ isn't settled;} \\ \mathtt{SENT} + v', & \text{if } v \text{ is settled with strong component leader } v'. \end{cases} \qquad (12)$$

Algorithm T (*Strong components*). Given a digraph g in SGB format, this algorithm determines the strong components of g. Each vertex is assumed to contain utility fields PARENT, ARC, LINK, and REP, as explained above. Auxiliary vertex pointer variables u, v, w, ROOT, and SINK are used; also an arc pointer a and an integer p. A dummy vertex $\mathtt{SENT} = \mathtt{VERTICES}(g) + \mathtt{N}(g)$ is assumed to be present, with $\mathtt{REP}(\mathtt{SENT}) = 0$.

T1. [Initialize.] Set $\mathtt{PARENT}(w) \leftarrow \Lambda$ for $\mathtt{VERTICES}(g) \le w < \mathtt{SENT}$, until eventually $w = \mathtt{SENT}$. Also set $p \leftarrow 0$ and $\mathtt{SINK} \leftarrow \mathtt{SENT}$.

T2. [Done?] (All vertices reachable from a vertex $v \ge w$ are now settled.) Terminate if $w = \mathtt{VERTICES}(g)$. Otherwise set $w \leftarrow w - 1$, and repeat this step if $\mathtt{PARENT}(w) \neq \Lambda$. Otherwise set $v \leftarrow w$, $\mathtt{PARENT}(v) \leftarrow \mathtt{SENT}$, and $\mathtt{ROOT} \leftarrow v$.

T3. [Begin to explore from v.] Set $a \leftarrow \mathtt{ARCS}(v)$, $p \leftarrow p + 1$, $\mathtt{REP}(v) \leftarrow p$, and $\mathtt{LINK}(v) \leftarrow \mathtt{SENT}$.

T4. [Done with v?] If $a = \Lambda$, go to T7.

T5. [Visit the next arc, $v \longrightarrow u$.] Set $u \leftarrow \mathtt{TIP}(a)$ and $a \leftarrow \mathtt{NEXT}(a)$.

T6. [If u is new, move to it.] If $\mathtt{PARENT}(u) = \Lambda$, set $\mathtt{PARENT}(u) \leftarrow v$, $\mathtt{ARC}(v) \leftarrow a$, $v \leftarrow u$, and go to T3. Otherwise, if $u = \mathtt{ROOT}$ and $p = \mathtt{N}(g)$ (we're in the last component), while $v \neq \mathtt{ROOT}$ set $\mathtt{LINK}(v) \leftarrow \mathtt{SINK}$, $\mathtt{SINK} \leftarrow v$, $v \leftarrow \mathtt{PARENT}(v)$; then set $u \leftarrow \mathtt{SENT}$ and go to T8. Otherwise, if $\mathtt{REP}(u) < \mathtt{REP}(v)$, set $\mathtt{REP}(v) \leftarrow \mathtt{REP}(u)$, $\mathtt{LINK}(v) \leftarrow \Lambda$ (see (11) and (12)). Go to T4.

T7. [Finish with v.] Set $u \leftarrow \mathtt{PARENT}(v)$. If $\mathtt{LINK}(v) = \mathtt{SENT}$, go to T8. (See Theorem T.) Otherwise, if $\mathtt{REP}(v) < \mathtt{REP}(u)$, set $\mathtt{REP}(u) \leftarrow \mathtt{REP}(v)$ and $\mathtt{LINK}(u) \leftarrow \Lambda$. Then set $\mathtt{LINK}(v) \leftarrow \mathtt{SINK}$, $\mathtt{SINK} \leftarrow v$, and go to T9.

T8. [New strong component.] (Now v and its unsettled descendants form a strong component that will be represented by v.) While $\mathtt{REP}(\mathtt{SINK}) \ge \mathtt{REP}(v)$, set $\mathtt{REP}(\mathtt{SINK}) \leftarrow \mathtt{SENT} + v$ and $\mathtt{SINK} \leftarrow \mathtt{LINK}(\mathtt{SINK})$. (See (12).) Finally set $\mathtt{REP}(v) \leftarrow \mathtt{SENT} + v$.

T9. [Tree done?] If $u = \mathtt{SENT}$, go back to T2. Otherwise set $v \leftarrow u$, $a \leftarrow \mathtt{ARC}(v)$, and go to T4. ▍

Step T6 may terminate the algorithm before all of the arcs have been seen, in cases where the final strong component is already known. (See exercise 52.)

One can only marvel at how everything fits together like clockwork in this algorithm! Algorithm T traverses each vertex and each arc at most once, and whenever it needs to make a crucial decision the necessary information just happens to be right there at its fingertips.

The running time is, of course, linear in the number of vertices and edges. With m edges and n vertices, Algorithm T makes at most $5m + 17n$ accesses to memory (see exercise 53).

And there's also an extra bonus: The strong components C_1, C_2, ..., C_s of g are not only discovered, they are actually produced in "topological order": Every arc *not* inside a strong component runs from C_j to a component C_i for which $i < j$. In particular, if g has no cycles, every strong component will contain a single vertex, and the vertices will be *topologically sorted.* (Algorithm 2.2.3T is a bit faster, in this special case, but of course it solves a much simpler problem.)

With minor extensions to Algorithm T, we can also (i) compute the dag analogous to (6) that results when strong components shrink to a point; and (ii) compute a subset of the arcs that's sufficient to certify the strength of each strong component. (See exercises 55 and 56.)

Weak components. Let's write '$v \Rightarrow w$' if w is reachable from v, and '$v \not\Rightarrow w$' otherwise. If vertices v and w of a digraph g are mutually reachable from each other (written '$v \Leftrightarrow w$'), they lie in the same strong component, so they're "strongly equivalent." On the other hand if they're *mutually unreachable* (written '$v \parallel w$'), meaning that neither one is reachable from the other, we shall regard them as "weakly equivalent." Two vertices that are strongly equivalent should naturally also be considered to be weakly equivalent; so let's write '$v \approx w$' to mean that either $v \Leftrightarrow w$ or $v \parallel w$. We can now rely on transitivity to say that v is weakly equivalent to w (written '$v \asymp w$') if and only if there's a chain

$$v = v_0 \approx v_1 \approx \cdots \approx v_r = w, \quad \text{for some } r \geq 0. \tag{13}$$

Weak equivalence is obviously an equivalence relation, and the corresponding equivalence classes are called the *weak components* of g. (Some authors define weak components differently, by saying that weak components are simply the components of the undirected graph that's obtained by ignoring the orientations of arcs. It's better, however, to call those the *undirected components* of g.)

Every weak component is clearly a union of strong components. For example, the weak components of the digraph (1) are $\{\{95,0,8\},\{31624\},\{7\}\}$, obtained by combining the mutually unreachable strong components in (6). In general, *strong components are partially ordered and weak components are totally ordered*, by the reachability relation between their vertices. Indeed, strong components and weak components are perhaps best understood as the finest set partitions of the vertices for which that statement is true.

A graph is *weakly connected* if it has only one weak component, which means that $u \asymp v$ for all vertices u and v. Exercises 80–85 discuss alternative ways to

topological order
certify
weak components–
mutually unreachable
notation $v \Leftrightarrow w$
notation $v \parallel w$
transitivity
notation $v \asymp w$
weak components
undirected components
partial ordering
total ordering
linear ordering, see total
set partitions
weakly connected

dag
source
sink

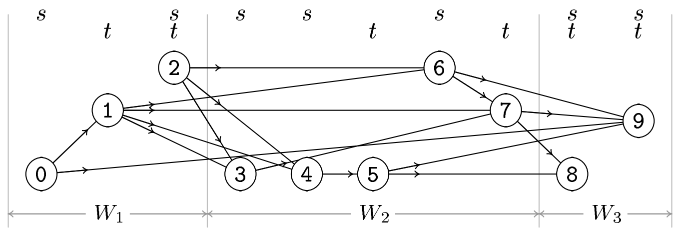


Fig. 600. A directed acyclic graph with ten vertices, seventeen arcs, and three weak components (W_1, W_2, W_3). Each source vertex within a component is marked with 's', and each sink with 't'. Weak components are linearly ordered; and when two weak components are adjacent, there are arcs from each sink of W_j to each source of W_{j+1}.

define weak equivalence and weak connectedness; one of the most noteworthy is the characterization that's proved in exercise 84:

Lemma W. *A digraph is weakly connected if and only if we can't split its vertices into two nonempty parts L and R such that (i) every vertex of R is reachable from every vertex of L; (ii) no vertex of L is reachable from any vertex of R.* ▮

(Notice that (ii) characterizes *ordinary* connectedness of *undirected* graphs.)

When we shrink all strong components of a digraph into single loopless vertices, we're left with a directed *acyclic* graph (dag), whose weak components have important structural properties. Consider, for example, the dag in Fig. 600, whose weak components (W_1, W_2, W_3) are depicted with vertical lines between them. A strong component is called a *source* with respect to weak component W_j if it has no predecessors in W_j; it's called a *sink* of W_j if it has no successors in W_j.

A weak component with more than one vertex must have at least two sources, and at least two sinks, because of Lemma W.

In general the weak components of any digraph can always be placed into a linear order and labeled $(W_1, W_2, \ldots, W_r)$ in such a way that

$$v \Rightarrow w \quad \text{for all } v \in W_j \text{ and } w \in W_k \text{ with } 1 \le j < k \le r. \qquad (14)$$

This condition is equivalent to requiring that $t \longrightarrow s$ for all sinks t of W_j and all sources s of W_{j+1}, for $1 \le j < r$. (For example, when $j = 1$ in Fig. 600 we have $\mathtt{1} \longrightarrow \mathtt{3}$, $\mathtt{1} \longrightarrow \mathtt{4}$, $\mathtt{1} \longrightarrow \mathtt{6}$, $\mathtt{2} \longrightarrow \mathtt{3}$, $\mathtt{2} \longrightarrow \mathtt{4}$, and $\mathtt{2} \longrightarrow \mathtt{6}$.) Thus all dags with $r > 1$ weak components can be obtained by starting with an arbitrary sequence of r weakly connected dags, then hooking together sinks to sources at the $r - 1$ boundaries between them, and adding optional arcs between W_i and W_j for $i < j$.

Suppose we introduce a new vertex v and insert it at the left of Fig. 600. There are 2^{10} different ways to make a new 11-vertex dag by adding arcs from v to a subset of $\{\mathtt{0}, \mathtt{1}, \ldots, \mathtt{9}\}$, but only a few essentially different cases arise. For example, Case 1 occurs when there are arcs $v \longrightarrow \mathtt{0}$ and $v \longrightarrow \mathtt{2}$ to both of the sources in W_1; then $\{v\}$ becomes a new weak component W_0, all by itself, in which v is both a source and a sink. The other weak components don't change.

Case 2 arises when there are arcs $v \longrightarrow \mathtt{3}$, $v \longrightarrow \mathtt{4}$, $v \longrightarrow \mathtt{6}$ from v to all the sources of W_2, but no arcs from v into W_1. Then v becomes part of W_1, as a new source and sink; W_2 and W_3 remain unchanged.

Tarjan
leader
HIT
WHIT

There's an intermediate Case 1′ between Case 1 and Case 2, namely when v "hits" at least one of the vertices $\{0,1,2\}$ of W_1 but not both of its sources $\{0,2\}$. In Case 1′, as in Case 2, v joins W_1 as a source vertex, and the other components stay the same. But in this case v is not a sink; furthermore, 0 and 2 will still be sources only if they aren't hit.

Case 3, by analogy with Cases 1 and 2, arises when v hits both sources of W_3 but doesn't touch either W_1 or W_2. There's also Case 2′, analogous to Case 1′, when v hits W_2 at least once but without touching W_1 and without hitting all three of W_2's sources. In both of these cases the weak components W_1 and W_2 will merge together, and together with v they'll become a new W_1. The former sink vertices of W_1 will no longer be sinks, and the former source vertices of W_2 will no longer be sources; perhaps 3, 4, and/or 6 will no longer be sources either. Weak component W_3 won't change, except that it will get the new name W_2.

The reader should now be able to guess what happens in the remaining two cases, namely Case 4 and Case 3′, after which only one weak component remains.

In all cases it's easy to determine the weak components, sources, and sinks of the new dag, if we know them before v was introduced. In fact, sinks aren't important; we need only keep track of the sources. Exercise 88 has full details.

Algorithm T has the nice property that it discovers the strong components from right to left. Therefore, after each strong component is found, the task of maintaining the current *weak* components is essentially the same as introducing a new vertex v at the left, as we've just discussed! So there's good news: It's actually possible to discover the weak components at the same time as we learn the strong ones, by extending Algorithm T slightly. The following algorithm [R. E. Tarjan, *Information Processing Letters* **3** (1974), 13–15] achieves this while still needing only $O(m+n)$ steps of computation.

Of course we'll need to extend our data structures to include additional fields; each vertex record gains two link fields, `WLINK` and `SRC`. Suppose the strong components discovered so far belong to weak components $(W_1,\ldots,W_r)$. The leader of the leftmost strong component of W_j (the one found last) is called the leader of W_j. Let vertex v be that leader; then `WLINK`(v) is the leader of W_{j+1}, or `SENT` if $j=r$. Variable `WP` is the leader of W_1. Hence, for example, Fig. 600 is represented by `WP` $=$ `0`, `WLINK(0)` $=$ `3`, `WLINK(3)` $=$ `8`, `WLINK(8)` $=$ `SENT`.

The `SRC` fields are similar, but they track sources. If v is the leader of a strong component that's a source of W_j, `SRC`(v) is the next such source to the right, or `SENT` if v is rightmost. In Fig. 600, for example, `SRC(0)` $=$ `2`, `SRC(2)` $=$ `SENT`; `SRC(3)` $=$ `4`, `SRC(4)` $=$ `6`, `SRC(6)` $=$ `SENT`; `SRC(8)` $=$ `9`, `SRC(9)` $=$ `SENT`.

The values of `WLINK`(v) and `SRC`(v) are unused, and undefined, unless vertex v plays one of the special roles already described. So we needn't set them.

Each vertex also has two additional 1-bit fields, called `HIT` and `WHIT`; they turn out to be necessary in order to keep the running time down to $O(m+n)$. We have `WHIT`$(v)=1$ if and only if there's an arc to v from the current vertex w. And `HIT`$(v)=1$ implies that vertex v has been hit; hence it's definitely *not* a source, although the `SRC` links might not yet have been updated to indicate that fact.

serial number
topological order
SETTLED
data structures

The vertex records of our data structure have eight fields so far: PARENT, LINK, REP, ARC, WLINK, SRC, WHIT, and HIT. To round them out, we add a ninth field SRANK(v), which will hold the serial number of v's strong component when v is a leader. Every strong component will be assigned a serial number in topological order, ending with $\mathtt{N}(g)-1$; the smallest number will be $\mathtt{N}(g)-c$, if c strong components have been identified.

Algorithm W (*Weak components*). Given a digraph g in SGB format, this algorithm extends Algorithm T by also identifying g's weak components. In fact, steps W1–W8 take the place of step T8 in that algorithm. Step T1 should be amended so that it initializes WHIT(w) $\leftarrow$ HIT(w) $\leftarrow 0$ at the same time as it sets PARENT(w) $\leftarrow \Lambda$. Step T1 should also set WP $\leftarrow$ PREV $\leftarrow$ SENT, $s \leftarrow$ SRANK(SENT) $\leftarrow \mathtt{N}(g)$, HIT(SENT) $\leftarrow 0$, and SETTLED $\leftarrow \Lambda$. Upon termination, SETTLED will point to the beginning of a list of all vertices, linked in the LINK fields. The vertices of each component, whether strong or weak, will appear consecutively in that list, beginning with the component leader.

W1. [Begin new strong component v.] Set $u'' \leftarrow u$, $s \leftarrow s-1$, SRANK(v) $\leftarrow s$, SRC(v) $\leftarrow$ PREV, PREV $\leftarrow v$, LINK(v) $\leftarrow$ SINK, $t \leftarrow v$. Perform step T8 of Algorithm T, also setting $t \leftarrow$ SINK just before SINK $\leftarrow$ LINK(SINK) in that loop. Finally set LINK(t) $\leftarrow$ SETTLED and SETTLED $\leftarrow v$.

W2. [Find v's least successor, v'.] Set $w' \leftarrow$ SETTLED and $v' \leftarrow$ SENT. (We will reexamine all arcs that lead from strong component v.) For $a \leftarrow$ ARCS(w'), while $a \neq \Lambda$, set $u \leftarrow$ REP(TIP(a)) $-$ SENT and $a \leftarrow$ NEXT(a); if $u \neq v$, set WHIT(u) $\leftarrow 1$ and if SRANK(u) $<$ SRANK(v') also set $v' \leftarrow u$. Then if $w' \neq t$, set $w' \leftarrow$ LINK(w') and run through all arcs of w' in the same way.

W3. [Find w', the weak component of v'.] (Now v' is the leader of the successor strong component of v that has smallest SRANK.) If $v' =$ SENT, set WP $\leftarrow$ SENT and go to W6. Otherwise set $w' \leftarrow$ SRC(v), and while SRANK(WP) $\leq$ SRANK(v') set $w' \leftarrow$ WP and WP $\leftarrow$ WLINK(w'). Finally set $u \leftarrow w'$.

W4. [All sources of w' hit?] If $u =$ SENT, go to W5. Otherwise, if WHIT(u) $= 0$, go to W6. Otherwise set $u' \leftarrow u$, $u \leftarrow$ SRC(u), and while HIT(u) $= 1$ set $u \leftarrow$ SRC(u) and SRC(u') $\leftarrow u$; then repeat step W4. (See exercise 89.)

W5. [Move WP back one.] Set WP $\leftarrow w'$. If $w' =$ SRC(v), also set SRC(v) $\leftarrow$ SENT.

W6. [Finish WLINK(v).] Set WLINK(v) $\leftarrow$ WP.

W7. [Reset WHITs.] Set $w' \leftarrow$ SETTLED. (This step is similar to W2.) For $a \leftarrow$ ARCS(w'), while $a \neq \Lambda$, set $u \leftarrow$ REP(TIP(a)) $-$ SENT and $a \leftarrow$ NEXT(a); if $u \neq v$, set WHIT(u) $\leftarrow 0$ and if SRANK(u) $<$ SRANK(WP) also set HIT(u) $\leftarrow 1$. Then if $w' \neq t$, set $w' \leftarrow$ LINK(w') and run through all arcs of w' in the same way.

W8. [Restore u.] Set $u \leftarrow u''$ and go to step T9. ▌

The reader will find it instructive to compare this algorithm to exercise 88, in order to see how the current weak components are updated efficiently in each of the various cases that can arise. Once again, the information that we need to make each decision is “magically” present at our fingertips in the data structures.

(Who knows what I might say next?)

biconnected components–
biconnectivity
strong connectivity
TARJAN

Biconnected components.

Some day this section will include another major algorithm (Algorithm 7.4.1.2B), namely the Hopcroft–Tarjan for bicomponents.

The similarity between biconnectivity and strong connectivity
revealed by the depth-first search approach is striking.
— ROBERT TARJAN, *SIAM Journal on Computing* (1972)

Historical notes. The term depth-first search was introduced by Nils Nilsson in his classic book *Problem-Solving Methods in Artificial Intelligence* (New York: McGraw–Hill, 1971). His methods resembled Algorithm Q. The graphs of particular interest to him were "state spaces," where the vertices are the configurations of some model and the arcs are the allowable transitions; the problem was to search for good ways to get from an initial state to a goal state.

The special case of depth-first search in which the given digraph is a connected and undirected graph was devised in the 19th century by C. P. Trémaux, in the context of a person who is able to drop pebbles while exploring the tunnels of a mine. The problem was to start at an arbitrary point and to traverse every passage exactly twice, once in each direction, eventually returning to the start; thus it was to find an Eulerian trail in a balanced digraph, whose existence is guaranteed by Theorem 2.3.4.2G. [See É. Lucas, *Récréations Mathématiques* **1** (Paris: Gauthier-Villars, 1882), 47–50.] A similar but less efficient procedure had previously been published by C. Wiener [*Mathematische Annalen* **6** (1873), 29–30]. The validity of Trémaux's algorithm was first proved rigorously by D. König, *Theorie der endlichen und unendlichen Graphen* (1936), Chapter 3.

The fact that depth-first search is extremely useful for graph algorithms was discovered by John Hopcroft and Robert Tarjan when they shared an office at Stanford University during Hopcroft's first sabbatical year. After discovering Algorithm B, they continued to develop related methods for planarity testing and other problems; see ACM Algorithm 447, *CACM* **16** (1973), 372–378.

The concept of a strongly connected digraph seems to have arisen gradually. H. E. Robbins introduced "orientable graphs" by considering a city that wants to convert all of its roads to one-way streets on weekends during football season [*AMM* **46** (1939), 281–283]. R. D. Luce developed the theory of digraphs that remain strongly connected after k arcs are removed [*American J. Math.* **74** (1952), 701–722]. He called them "networks that are connected"; the term "strongly connected" was introduced by F. Harary [*Trans. Amer. Math. Soc.* **78** (1955), 459].

R. E. Tarjan described the beauties of the depth-first jungle and introduced a precursor to Algorithm T in *SICOMP* **1** (1972), 146–160. His original algorithm used a different definition of `LOW`; the present version was inspired in part by J. Eve and R. Kurki-Suonio [*Acta Informatica* **8** (1977), 303–314]. Lemma S is based largely on ideas developed by E. W. Dijkstra in *A Discipline of Programming* (Prentice–Hall, 1976), Chapter 25. Interesting but less efficient algorithms had been published previously by R. W. H. Sargent and A. W. Westerberg [*Transactions of the Institution of Chemical Engineers* **42** (1964), 190–197]; L. Ya. Leifman [*Kibernetika* **2**, 5 (1966), 19–23]; P. Purdom Jr. [*BIT* **10** (1970), 76–94]; I. Munro [*Information Processing Letters* **1** (1971), 56–58].

Tarjan revisited this work 50 years later, in a review paper written with Uri Zwick [to appear].

EXERCISES

1. [*16*] Complete (2). Also show the stack contents at the beginning of step Q6.

3. [*18*] Instead of using the linked mechanism `ARCS`, `TIP`, `NEXT` of SGB format, the arcs of digraphs are sometimes represented sequentially in a single array called, say, `TIP`. The arcs from vertex v then are $v \longrightarrow \mathtt{TIP}[k]$ for $\mathtt{ARCS}[v] \le k < \mathtt{ARCS}[v+1]$, where $\mathtt{ARCS}[v_0+n]$ is suitably defined. Modify Algorithm Q to work with this convention.

5. [*M20*] Exactly how often are each of the operations or tests (a) $\mathtt{MARK}(w) \leftarrow \Lambda$, (b) $\mathtt{MARK}(v) \ne \Lambda$, (c) $\mathtt{MARK}(v) \leftarrow \mathtt{SENT}$, (d) $a \leftarrow \mathtt{ARCS}(v)$, (e) $u \leftarrow \mathtt{TIP}(a)$, (f) $a \leftarrow \mathtt{NEXT}(a)$, (g) $\mathtt{MARK}(u) = \Lambda$, (h) $\mathtt{MARK}(u) \leftarrow s$, and (i) $s \leftarrow \mathtt{MARK}(v)$ performed by Algorithm Q? Express your answers in terms of $m = \mathtt{M}(g)$, $n = \mathtt{N}(g)$, and the number r of times that step Q3 is executed.

▶ **7.** [*20*] (*Breadth-first search.*) Whenever we traverse a digraph, its vertices begin “unseen”; then they're “active”; and finally they're “explored.” A general search procedure operates repeatedly as follows, until all vertices have been explored:

> Choose an active vertex, v; or if none are active, activate an unseen vertex v.
> If v has no unseen arcs, call it explored. Otherwise let $v \longrightarrow u$ be an unseen arc;
> if vertex u is unseen, add the arc $v \longrightarrow u$ to the spanning forest and activate u.

Algorithm D is the special case where we always choose the most recently activated v.

By contrast, breadth-first search, which we've used for example in Section 7.3 to find the shortest paths from a root vertex, always chooses the *least* recently activated v. Show that simple changes to Algorithm Q will implement breadth-first search efficiently.

8. [*17*] What spanning forest does the algorithm of exercise 7 construct from (1)?

10. [*M20*] Count exercise 7's memory references, in comparison with exercise 5.

12. [*18*] Complete (3) by listing *all* of the arcs a that are seen in step D4.

13. [*16*] Identify the back arcs, forward arcs, loops, and cross arcs in (4).

▶ **15.** [*M20*] Prove that a digraph is acyclic if and only if it has no back arcs or loops.

16. [*20*] Sometimes we know that every vertex of a digraph is reachable from a single vertex v_0. Show that Algorithm D can be simplified in such cases.

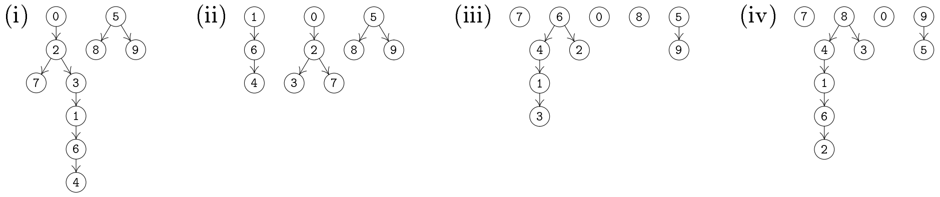
18. [*20*] Prove that $\mathtt{PRE}(v) < \mathtt{PRE}(u)$ implies $\mathtt{POST}(v) > \mathtt{POST}(u)$ in Theorem D.

▶ **19.** [*M21*] Count Algorithm D's memory references, in comparison with exercise 5.

20. [*20*] Rewrite Algorithm D as a recursive procedure. (In this formulation, the `PARENT` and `ARC` fields won't be needed.)

▶ **21.** [*21*] Modify Algorithm D so that it classifies every arc a in step D5 as either a tree arc, a back arc, a forward arc, a loop, or a cross arc.

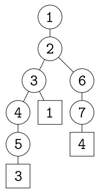
22. [*20*] Which of the following oriented forests could be produced by Algorithm D from the digraph (1), if the internal orderings of arcs and vertices are changed?



ARCS
TIP
NEXT
SGB format
sequentially
Breadth-first search
unseen
active
explored
general search procedure
back arcs
forward arcs
loops
cross arcs
acyclic
reachable
recursive procedure
tree arc

▸ **23.** [*21*] The *depth-first jungle forest* of a digraph g is obtained as follows: Start with one internal node for each vertex, represented as a circle labeled with the name of that vertex. The kth child of internal node v is internal node w, if the kth arc $v \longrightarrow w$ is a tree arc; otherwise that kth child is an external node, represented as a square labeled with w.

For example, the depth-first jungle forest for the digraph whose depth-first jungle appears after (10) in the text is shown here.



a) Draw the depth-first jungle forest that corresponds to (4).

b) How many digraphs have exactly the same depth-first jungle forest as (a), except for the labels in the external nodes?

c) How many digraphs have exactly the same depth-first jungle forest as (a), except for the node labels (either internal or external)?

24. [*M22*] Generalize the results of exercise 23, considering all possible digraphs.

25. [*22*] Test Algorithm D empirically on random digraphs with $n = 1000000$ vertices, where each of $m = 5n$ arcs goes from a uniformly random vertex v to a uniformly random vertex u. How many tree arcs, back arcs, forward arcs, and cross arcs are formed?

26. [*22*] Continuing exercise 25, test Algorithm D when the arcs from each vertex v are obtained as follows: "With probability $p = 5/6$, generate a uniformly random vertex u, create the arc $v \longrightarrow u$, and repeat this process. Otherwise stop." (The expected number of arcs from v will be $(p + 2p^2 + 3p^3 + \cdots)(1-p) = p/(1-p) = 5$.)

▸ **27.** [*HM27*] At first glance the "uniform" model of random digraphs in exercise 25 might seem to be roughly equivalent to the "geometric" model in exercise 26, when $m = pn/(1-p)$ and $n \to \infty$. Prove, however, that the models are actually quite different, by explaining how to compute the probability that a given depth-first jungle forest with m edges is obtained with (a) the uniform model; (b) the geometric model when $p = m/(m+n)$. Evaluate those probabilities for the forest of exercise 23(a).

28. [*HM21*] What is the probability that a random digraph has exactly k loops $v \longrightarrow v$, under the (a) uniform (b) geometric model, assuming that $p = m/(m+n)$?

29. [*HM22*] (P. Jacquet.) Generalize the geometric model of exercise 26 as follows: Every vertex v independently has out-degree d with probability π_d, where $(\pi_0, \pi_1, \ldots)$ is an *arbitrary* probability distribution. Once d has been chosen, each arc $v \longrightarrow w_j$ for $1 \le j \le d$ then goes to a uniformly random vertex w_j.

Show that a random depth-first jungle forest with n internal nodes, conforming to this model, can be generated with a simple Markov process.

▸ **30.** [*HM30*] Given a forest F with n nodes, explain how to construct the multivariate generating function g_F for which $[v^l w^t x^b y^f z^c]\, g_F(v,w,x,y,z)$ is the probability that the geometric model of exercise 26 will produce a random digraph whose depth-first forest is F and whose depth-first jungle contains exactly l loops, t tree arcs, b back arcs, f forward arcs, and c cross arcs. For example, there are two possibilities when $n = 2$:

$$g_F = \begin{cases} (1-p)^2 pw/\bigl(2(1-(v+x)p/2)(1-vp/2)(1-(v+y)p/2)\bigr), & \text{if } F = {:};\\ (1-p)^2/\bigl((1-vp/2)(1-(v+z)p/2)\bigr), & \text{if } F = {\cdot\cdot}. \end{cases}$$

Notice that g_F is a function of the geometric probability parameter p.

depth-first jungle forest
internal node
external node
random digraphs
random digraphs, uniform model
uniform model of random digraphs
random digraphs, geometric model
geometric model of random digraphs
depth-first jungle forest
loops
analysis of algorithms+
Jacquet
Markov process
forest
multivariate generating function
generating function
geometric model

31. [*HM23*] Exercise 30 allows us to compute the probability $g_F(1,1,1,1,1)$ that a geometrically random digraph will have F as its depth-first forest. When $n=2$, for example, that forest will be $\genfrac{}{}{0pt}{}{\bullet}{\bullet}$ with probability $(1-p)^2p/(2(1-p)(1-p/2)(1-p)) = p/(2-p)$; thus $\genfrac{}{}{0pt}{}{\bullet}{\bullet}$ is more likely than $\bullet\,\bullet$ if and only if the parameter p exceeds $\frac{2}{3}$.

There are 14 possible forests when $n=4$ (see Tables 1–3 in Section 7.2.1.6). Which of them is (a) most (b) least likely to be the depth-first forest, as p increases from 0 to 1?

32. [*HM23*] Continuing exercise 30, let g_n be the sum of g_F over all n-node forests F. Also let $g_n^l(z) = g_n(z,1,1,1,1)$, $g_n^t(z) = g_n(1,z,1,1,1)$, $g_n^b(z) = g_n(1,1,z,1,1)$, $g_n^f(z) = g_n(1,1,1,z,1)$, and $g_n^c(z) = g_n(1,1,1,1,z)$ be the univariate generating functions for the loops, tree arcs, back arcs, forward arcs, and cross arcs in isolation. (Of course $g_n(1,1,1,1,1) = g_n^l(1) = \cdots = g_n^c(1) = 1$. Exercise 28(b) analyzes g_n^l.)

a) What are $g_2^l(z)$, $g_2^t(z)$, $g_2^b(z)$, $g_2^f(z)$, and $g_2^c(z)$?
b) What is $g_n(z,z,z,z,z)$?
c) Find surprises in $g_4^{l\prime}(1)$, $g_4^{t\prime}(1)$, $g_4^{b\prime}(1)$, $g_4^{f\prime}(1)$, and $g_4^{c\prime}(1)$.
d) For what values of p does the average number of back arcs exceed the average number of tree arcs, in the geometric model when $n=4$?

34. [*M28*] Exercise 32 mentions that $g_2^b(z) = g_2^f(z)$. Prove that, for all n, the number of back arcs has the same mean as the number of forward arcs, in the depth-first jungles of random digraphs generated by the geometric model of exercise 26 with probability p. Furthermore the number of tree arcs has the same mean as the number of cross arcs.

35. [*HM37*] Continuing exercise 34, prove that in fact the joint probability distribution of the random variables $(L,F,B{+}C,T)$ is exactly the same as the joint probability distribution of $(L,B,F{+}C,T)$, where L, B, F, C, and T are respectively the numbers of loops, back arcs, forward arcs, cross arcs, and tree arcs of random geometric depth-first forests, for all n and p.

36. [*HM30*] Study the depth-first jungles of random mappings (random "functional digraphs"). How many tree arcs, back arcs, . . . , cross arcs do they have, asymptotically?

37. [*HM40*] Determine the asymptotic number of arcs of different kinds, in the depth-first jungles of random digraphs defined by the geometric model with probability p.

38. [*HM47*] Extend the analysis of exercise 37 to the generalized model of exercise 29.

39. [*15*] What are the strong components of the "Orient Express" digraph of Fig. 3, near the beginning of Chapter 7? (These are the groups of murder suspects whose alibis depend on each other. Answer the question without computer help.)

40. [*M20*] A *Markov process* is an algorithm that performs a random walk on a set of states. If there are n states it can be defined by an $n\times n$ matrix (p_{ij}), where p_{ij} is the probability of going from state i to state j at each step.

In exercise 2.3.4.2–26 we studied Markov processes that terminate with probability 1; however, it is more usual to assume that $p_{i1}+\cdots+p_{in}=1$ for $1\le i\le n$. Thus the process *never* terminates, although it might reach an "absorbing" state in which it remains forever. The digraph g on vertices $\{1,\ldots,n\}$ whose arcs are $i \longrightarrow j$ when $p_{ij}>0$ represents all possible walks of the process.

State j is called *transient* if there's positive probability that a walk starting at j will never return to j; otherwise state j is called *recurrent*. Characterize transient and recurrent states in terms of the strong components of g.

▸ **41.** [*22*] Rewrite Algorithm T as a recursive procedure. (Compare with exercise 20.) However, omit the shortcut that tests for $u = \texttt{ROOT}$ and $p = \texttt{N}(g)$ in step T6.

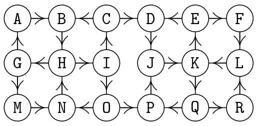
random mappings
functional digraphs
Orient Express
Markov process
random walk
absorbing
transient
recurrent
strong components
recursive procedure

43. [*20*] In what order do the arcs of digraph (1) mature during a depth-first search? (See (10) for the definition of "maturing.")

▸ **44.** [*21*] Extend Algorithm D by including a `TIME` field in each arc record. If a is the rth arc to mature, your algorithm should set $\mathtt{TIME}(a) \leftarrow r$.

45. [*20*] Describe all the `LOW` computations in the 7-vertex digraph following (10).

47. [*21*] What are the preorder and postorder of the depth-first forest when g is the digraph shown? (Begin at vertex `A`. Go left before right and up before down. Notice that every vertex has out-degree and in-degree 1 or 2.) What are the strong components? What are the final values of `LOW(`v`)` at the end of Algorithm T?



48. [*20*] How many strong components does the digraph of exercise 47 have when one of its arcs has been removed? (Answer this question for each of the 26 arcs.)

▸ **50.** [*M25*] Prove Theorem T, which justifies step T7 of Algorithm T.

51. [*18*] Why is it important to have $\mathtt{REP(SENT)} = 0$ in Algorithm T?

52. [*20*] Why is it sometimes permissible for step T6 to go directly to step T8?

53. [*M21*] Count Algorithm T's memory references, in comparison with exercise 19.

54. [*20*] Extend Algorithm T so that it prints out the vertices of each strong component, preceded by their leader. In what order are strongly equivalent vertices printed?

▸ **55.** [*25*] Let $\widehat{g}$ be the directed acyclic graph whose vertices are the strong components of g and whose arcs correspond to directly related components, as in (6). Extend Algorithm T so that it prints out exactly one arc of g for each arc of $\widehat{g}$.

▸ **56.** [*30*] Extend Algorithm T so that it prints out a subset of the arcs for which the resulting subgraph has the same strong components as g. Make the subset as small as you can without losing efficiency.

▸ **58.** [*M30*] The *period* of vertex v in a digraph is the greatest common divisor of the lengths of all oriented cycles that contain v (not necessarily simple).

a) Find the periods of the vertices in the digraph whose arcs are `A` $\longrightarrow$ `B`, `A` $\longrightarrow$ `E`, `B` $\longrightarrow$ `C`, `B` $\longrightarrow$ `G`, `C` $\longrightarrow$ `A`, `C` $\longrightarrow$ `F`, `D` $\longrightarrow$ `H`, `E` $\longrightarrow$ `C`, `F` $\longrightarrow$ `B`, `F` $\longrightarrow$ `D`, `H` $\longrightarrow$ `A`, `H` $\longrightarrow$ `F`.

b) Show that vertices in the same strong component have the same period.

c) Extend Algorithm T so that it computes the period of every vertex. Illustrate your algorithm by applying it to the digraph in (a). *Hint:* Give each vertex a `DEPTH` field, representing its distance from the root of a tree in the depth-first jungle.

61. [*22*] Study Algorithm T empirically by using the random digraph models of exercises 25 and 26. Estimate the number of strong components, the size of the largest strong component, the number of arcs actually examined in step T5, and the number of times `REP` changes in steps T6 and T7.

▸ **65.** [*22*] (*A whirlpool maze.*) The 11×11 matrix (a_{ij}) shown here represents a digraph with $11^2 + 1$ vertices, named (i, j) for $0 \le i, j < 11$ and `OUT`. There's an arc from (i, j) to (i', j') if and only if (i', j') is a_{ij} steps away from (i, j) in a horizontal, vertical, or diagonal direction. And there's an arc from (i, j) to `OUT` if and only if either i or j is either $a_{ij} - 1$ or $11 - a_{ij}$. For example, the legal moves from $(1, 2)$ are to $(1, 5)$, $(4, 2)$, $(4, 5)$, and `OUT`. There are eight legal moves from the '*2*' in the center cell $(5, 5)$.

3	1	4	1	5	9	6	5	3	5	9
7	9	3	2	3	8	4	6	2	6	4
5	8	8	4	6	5	8	5	4	8	6
6	9	7	6	6	5	6	5	5	5	5
9	8	8	6	6	2	6	4	4	8	8
9	9	4	7	5	2	6	7	8	8	3
9	9	9	3	2	4	3	4	8	4	4
3	9	5	3	5	5	3	5	9	3	5
5	6	6	5	6	5	6	6	5	5	5
3	9	2	6	6	1	3	1	5	5	6
3	4	7	3	8	8	8	9	1	7	8

a) Can you find (by hand) a path from $(5, 5)$ to `OUT`?

mature
LOW
preorder
postorder
directed acyclic graph
period
greatest common divisor
simple
giant strong component
whirlpool maze
maze
pi, for random example

b) Use Algorithm T to find the strong components of this digraph.
c) What subgraph does the extended algorithm in exercise 56 give?

70. [*M21*] Say something interesting about the strong components of a tournament.

▸ **80.** [*M20*] True or false (see (13)): $v \asymp w$ if and only if either $v \Leftrightarrow w$ or there's a chain

$$v = v_0 \parallel v_1 \parallel \cdots \parallel v_r = w, \quad \text{for some } r \geq 0.$$

81. [*M21*] Prove that $v \asymp w$ if and only if either $v \Leftrightarrow w$ or there's a nonpath from v to w and a nonpath from w to v, where a *nonpath* from v to w is a sequence

$$v = v_0 \not\Rightarrow v_1 \not\Rightarrow \cdots \not\Rightarrow v_r = w, \quad \text{for some } r \geq 1.$$

82. [*M20*] Construct a weakly connected digraph that contains two vertices v and w for which the relation of exercise 80 holds between v and w only when $r \geq 1000000$.

83. [*M22*] True or false: $u \asymp v$ and $v \Rightarrow w$ and $u \not\Rightarrow w$ implies $u \asymp w$.

84. [*M21*] Prove Lemma W.

▸ **85.** [*M27*] Prove that a dag with $n > 1$ vertices is weakly connected if and only if its vertices can be topologically sorted in two ways $u_1 u_2 \ldots u_n$ and $v_1 v_2 \ldots v_n$, such that $\{u_1, \ldots, u_l\} \neq \{v_1, \ldots, v_l\}$ for $1 \leq l < n$. (Topological sorting means that $u_i \longrightarrow u_j$ or $v_i \longrightarrow v_j$ implies $i < j$.)

86. [*16*] If a digraph has r weak components, in how many ways can its vertices be divided into nonempty parts L and R that satisfy conditions (i) and (ii) of Lemma W?

88. [*M21*] Let D be a dag with n vertices and weak components $(W_1, \ldots, W_r)$. Suppose we augment D by introducing an $(n+1)$st vertex v at the left, together with optional arcs from v. Say that vertex w is "hit" if there's a new arc $v \longrightarrow w$; otherwise w is "missed." Case j, for $1 \leq j \leq r$, arises when all the sources of W_j are hit, but all vertices of $W_1 \cup \cdots \cup W_{j-1}$ are missed. Case j' is similar, but at least one source of W_j is missed while W_j isn't missed entirely. Finally, Case $r+1$ arises when v hits nothing.
a) Exactly how many of the 2^n possible augmentations belong to each case, when each weak component W_j has exactly s_j sources and n_j vertices?
b) In each case, explain how the weak components, sources, and sinks are changed.

89. [*20*] What vertices are strong and weak leaders when Algorithm W terminates?

▸ **90.** [*25*] Prove that the running time of Algorithm W is linear in the number of vertices and arcs of g, and explain why the `HIT` fields enable such efficiency.

92. [*25*] Show that the number of sequences $(s_1, n_1; \ldots; s_r, n_r)$ with $1 \leq s_j \leq n_j$ for $1 \leq j \leq r$ and $n_1 + \cdots + n_r = n$ is the Fibonacci number F_{2n}. Also devise an easily computed one-to-one correspondence between such sequences and the integers $[0\,..\,F_{2n})$.

▸ **93.** [*M29*] For each sequence $\sigma = (s_1, n_1; \ldots; s_r, n_r)$ considered in exercise 92, let $g_\sigma(z)$ be the generating function $\sum_{m \geq 0} a_{\sigma m} z^m$, where $a_{\sigma m}$ is the number of subgraphs of the transitive tournament $K_n^{\rightarrow}$ that have m arcs and r weak components $(W_1, \ldots, W_r)$, where each W_j has exactly s_j sources and n_j vertices.
a) Compute $g_\sigma(z)$ for all σ with $n \leq 16$.
b) Use (a) to count the number of weakly connected dags on $\{1, \ldots, 16\}$ for $n \leq 16$.
c) Use (a) to determine the probability that a random subgraph of $K_{16}^{\rightarrow}$ has exactly r weak components, for $1 \leq r \leq 16$, where each arc is independently present with probability (i) 2/3; (ii) 1/2; (iii) 1/3; (iv) 1/16.

strong components
tournament
nonpath
Topological sorting
leaders
analysis of algorithms
linear
HIT
Fibonacci number
generating function
weakly connected
random subgraph

▶ **95.** [*M33*] Prove that a weakly connected subgraph of the transitive tournament $K_n^{\to}$ cannot have more than $\binom{n-1}{2}$ arcs. Furthermore, when $n > 1$, exactly 3^{n-2} such subgraphs have exactly $\binom{n-1}{2}$ arcs, and there's a nice one-to-one correspondence between them and ternary sequences $a_2 \ldots a_{n-1}$, where $a_j \in \{\texttt{-}, \texttt{0}, \texttt{+}\}$ for $1 < j < n$.

transitive tournament
$K_n^{\to}$
ternary sequences

99. [*00*] this is a temporary dummy exercise

999. [*M00*] this is a temporary exercise (for dummies)

SECTION 7.4.1.1

1. Yes. Indeed, any isolated vertices of a graph are among its connected components.

99. ...

999. ...

SECTION 7.4.1.2

1.

Stack	Arcs	Stack	Arcs
Begin 9	$9 \longrightarrow 7$, 3, 5, (7)	736	$6 \longrightarrow 2$, (4)
735	$5 \longrightarrow (9)$, (3), 8	732	$2 \longrightarrow (6)$, (3), (7)
738	$8 \longrightarrow 4$, (3), (8)	73	$3 \longrightarrow (1)$, (3)
734	$4 \longrightarrow 1$, (1)	7	$7 \longrightarrow$
731	$1 \longrightarrow 6$	Begin 0	$0 \longrightarrow (2)$

3. (The `TIP` array for (1) would be (2, 6, 6, ..., 5, 7), and the `ARCS` array would be (0, 1, 2, 4, ..., 17, 21).) Change "`ARCS(`v`)`" to "`ARCS[`v`]`" in step Q3. Step Q4 becomes "Go to Q6 if $a =$ `ARCS[`$v+1$`]`. Otherwise set $u \leftarrow$ `TIP[`a`]` and $a \leftarrow a+1$."

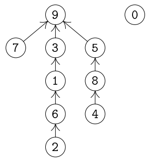
[Of course the sequential convention makes the task of visiting every arc trivial, so we wouldn't really be using Algorithm Q! But a similar change applies to Algorithm D and many others. Notice that we avoid fetching `NEXT(`a`)`; but we need an extra memory reference to fetch `ARCS[`$v+1$`]`. On the other hand, `TIP(`a`)` and `NEXT(`a`)` are typically fetched *simultaneously* with SGB format, as in the `MMIX` program of exercise 7–71.]

5. (a, b) n. (c) r. (d) n (r in step Q3 and $n-r$ in step Q6). (e, f, g) m. (h, i) $n-r$.

7. Introduce a new pointer t. Set `MARK(`v`)` $\leftarrow s \leftarrow t \leftarrow v$ in step Q2. In step Q5, set `MARK(`u`)` $\leftarrow$ `MARK(`t`)` $\leftarrow u$ and $t \leftarrow u$ (without changing s). In step Q6, change '$s =$ `SENT`' to '$s = t$', and change '$v \leftarrow s$, $s \leftarrow$ `MARK(`v`)`' to '$s \leftarrow$ `MARK(`v`)`, $v \leftarrow s$'.

(We've essentially changed the stack into a queue. If desired one can also set `PARENT(`v`)` $\leftarrow$ `SENT` in Q3 and `PARENT(`u`)` $\leftarrow v$ in Q5.)

8.



10. Step Q5 now makes $n-r$ more references, because it sets `MARK(`t`)`. Add another n if `PARENT` links are also to be set.

12. ... Λ (end 1); $3 \longrightarrow 3$; Λ (end 3); $9 \longrightarrow 5$; $5 \longrightarrow 9$; $5 \longrightarrow 3$; $5 \longrightarrow 8$; $8 \longrightarrow 4$; $8 \longrightarrow 3$; $8 \longrightarrow 8$; Λ (end 8); Λ (end 5); $9 \longrightarrow 7$; Λ (end 9); $0 \longrightarrow 2$; Λ (end 0).

13. Back arcs: $2 \longrightarrow 3$; $2 \longrightarrow 6$; $4 \longrightarrow 1$; $4 \longrightarrow 1$; $5 \longrightarrow 9$. Forward arcs: $9 \longrightarrow 7$. Loops: $3 \longrightarrow 3$; $8 \longrightarrow 8$. Cross arcs: $0 \longrightarrow 2$; $2 \longrightarrow 7$; $5 \longrightarrow 3$; $8 \longrightarrow 3$; $8 \longrightarrow 4$.

15. A back arc or loop certainly makes a cycle. Conversely, if there's a cycle $v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_{t-1} \longrightarrow v_0$, with v_0 smallest in preorder, each of v_1, ..., v_{t-1} is reachable from v_0 but was unseen when v_0 was first seen. Hence they are descendants of v_0. It follows that $v_{t-1} \longrightarrow v_0$ is either a back arc or a loop.

16. Change step D2 so that it simply sets $w \leftarrow v_0$. Terminate when $v =$ `SENT` in step D8. (Similar changes will also simplify Algorithm T.)

18. If `PRE(`v`)` $<$ `PRE(`u`)`, vertex u was unseen when v became active. And if $v \longrightarrow u$ is a nontree arc of g, vertex u must be reachable by tree arcs. Thus u is a proper descendant of v, and $v \longrightarrow u$ is a forward arc. Apply exercise 2.3.2–20.

isolated vertices
`MMIX`
stack
queue
preorder
reachable
unseen
descendants

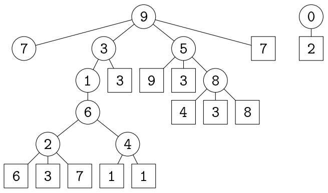
19. (ARC, ARCS, NEXT, PARENT, TIP) are fetched respectively $(n-r, n, m, m+2n, m)$ times; we store into (ARC, PARENT, POST, PRE) respectively $(n-r, 2n, n, n)$ times. (So it's $< 4n$ more memory accesses than in Algorithm Q, and we've got PRE and POST.) Tarjan

20. Set $\texttt{PRE}(w) \leftarrow 0$ for $\texttt{VERTICES}(g) \le w < \texttt{VERTICES}(g) + \texttt{N}(g)$, and $p \leftarrow q \leftarrow 0$, where p and q are global variables. Then, while $w > \texttt{VERTICES}(g)$, set $w \leftarrow w - 1$ and execute dfs(w) if $\texttt{PRE}(w) = 0$, where dfs(v) is the following procedure: "Set $p \leftarrow p+1$, $\texttt{PRE}(v) \leftarrow p$, and $a \leftarrow \texttt{ARCS}(v)$, where a is a local variable. While $a \ne \Lambda$, set $u \leftarrow \texttt{TIP}(a)$, $a \leftarrow \texttt{NEXT}(a)$, and execute dfs$(u)$ if $\texttt{PRE}(u) = 0$. Finally set $q \leftarrow q+1$ and $\texttt{POST}(v) \leftarrow q$."

21. Set $\texttt{POST}(w) \leftarrow 0$ for all w in step D1. Then, at the end of step D5, if $\texttt{PARENT}(u) = \Lambda$, a is a tree arc; otherwise if $u = v$, a is a loop; otherwise if $\texttt{PRE}(u) > \texttt{PRE}(v)$, a is a forward arc; otherwise if $\texttt{POST}(u) = 0$, a is a back arc; otherwise a is a cross arc.

22. (i) results when all the orderings used in (4) are reversed. (iii) might also occur. But (ii) and (iv) violate Theorem D with respect to nontree arcs $6 \longrightarrow 2$ and $2 \longrightarrow 3$. [Tarjan proved that any oriented spanning forest consistent with Theorem D is possible. The jungle (4) is obtained from 9! different orderings of the vertices and respectively $(1,1,6,2,1,6,2,1,6,3)$ independently different orderings of the arcs from $(0,\ldots,9)$.]

23. (a) Notice that the jungle forest carries more information than the jungle itself, because it encodes the order of the arcs as well as their initial and final vertices.



(b) The label in every external node must agree with the label of some internal node that precedes it in preorder of the forest. Thus, the first three external nodes in (a) could be labeled in six ways (either 9 or 7 or 3 or ... or 2); the next three in seven ways (also 4); and so on. The total number of possible labelings is therefore $6^3 7^3 8^2 9^4 10^1 = 311098475520$.

(c) The internal labels can be any permutation of the vertex names, except that the root of each forest must be labeled with the first vertex of its tree that is encountered in step D1; the roots must also be in "encounter-order" from left to right. In (a) this means only that the first root must be labeled 9. Hence the total number of possibilities for a depth-first jungle forest of that shape is $311098475520 \cdot 9! = 112891414796697600$.

24. In general, the depth-first jungle forest for a digraph with m arcs and n vertices has m edges and n internal nodes. If it contains r trees, it has $m+r$ nodes altogether. Hence it has $m+r-n$ external nodes, none of which are roots and all of which are leaves.

Conversely, if we start with *any* forest of r trees and $m+r$ nodes, where $m+r-n$ nonroot leaves are designated "external," the resulting structure can be labeled so that it becomes the depth-first jungle forest of a digraph with m arcs and n vertices.

The number of ways to label the external nodes is the product, over all external nodes, of the number of internal predecessors in preorder. The number of ways to label the internal nodes is $n!/(s_1 \ldots s_r)$, where $s_j = t_j + \cdots + t_r$ and t_j is the number of the internal nodes of the jth tree (see exercise 5.1.4–20).

25. First generate m random vertices and count how many times a_v each v occurs. Then set $\texttt{ARCS}[v] \leftarrow a_0 + \cdots + a_{v-1}$ for $0 \le v \le n$. Generate m more random vertices $\texttt{TIP}[k]$ for $0 \le k < m$, and use the convention of exercise 3.

The number of tree arcs is n minus the number of roots. Several thousand trials gave approximately 6860 ± 80 roots, $2.036 \cdot 10^6 \pm 1200$ back arcs, $1.507 \cdot 10^6 \pm 1100$ forward arcs, and $0.464 \cdot 10^6 \pm 1250$ cross arcs. (The forward arcs included 5 ± 2 loops. The maximum tree depth was $0.6725 \cdot 10^6 \pm 500$.)

26. With this random model it's not necessary to store individual arcs. Several thousand trials now gave approximately 6850 ± 90 roots, $1.507 \cdot 10^6 \pm 3300$ back arcs, $1.507 \cdot 10^6 \pm 3300$ forward arcs, and $0.993 \cdot 10^6 \pm 2200$ cross arcs. (The forward arcs included 5 ± 2 loops. The maximum tree depth was $0.4782 \cdot 10^6 \pm 700$.)

27. (a) Let $d_k = \text{out-degree}(v_k)$. There are $\binom{m}{d_1,\ldots,d_n}$ admissible orderings of the m arcs, and this should be multiplied by the probability m^{-2n} of obtaining them. In exercise 23 this comes to $21!/(1!1!3!2!2!3!2!0!3!4!10^{42}) = 1.23193822752 \times 10^{-27}$.

(b) All depth-first jungle forests with n internal nodes can be generated by a finite-state automaton whose states (d,t) represent the current depth and the number of internal nodes generated so far. Begin in state $(0,0)$ and stop when state $(0,n)$ is reached. When in state $(0,t)$ for $t < n$, generate a new internal node, which will be the root of a new tree, and go to state $(1,t+1)$; there are $n-t$ candidate labels for that new node. When in state (d,t) for $d > 0$, there are three choices: We either go to state $(d-1,t)$, meaning that node t has no further children; or we remain in state (d,t) after generating a new external child of node t; or we go to state $(d+1,t+1)$ after generating a new internal child of node t. In the second case there are t candidate labels; in the third case there are $n-t$.

In the geometric model the automaton corresponds to a Markov process in which the transition probabilities are 1 for $(0,t) \to (1,t+1)$, $1-p$ for $(d,t) \to (d-1,t)$, tp/n for $(d,t) \to (d,t)$, and $(n-t)p/n$ for $(d,t) \to (d+1,t+1)$. That produces a random depth-first jungle forest whose node labels are randomized as in exercise 23(c).

However, to match a given forest, the transition probabilities for both $(d,t) \to (d,t)$ and $(d,t) \to (d+1,t+1)$ become p/n, and the process terminates unsuccessfully if an incorrect label is generated. So the overall probability reduces to $(p/n)^m(1-p)^n$, which comes to $(5/60)^{21}/6^{10} \approx 3.6 \cdot 10^{-31}$ in the forest of exercise 23. (The probability of getting a forest with exactly m arcs is $[z^m]\,((1-p)/(1-pz))^n = \binom{n+m-1}{m}p^m(1-p)^n$; in exercise 23 it's $\binom{30}{21}(5/6)^{21}/6^{10} \approx .00514$.)

[Another way to explain the discrepancies between the empirical results of exercises 25 and 26 is to note that each vertex in the uniform model has the same distribution for its in-degree and out-degree, but that is far from true in the geometric model. Indeed, the probability of in-degree k in that model is $[z^k]\,\bigl((1-p)/(1-p(1+(z-1)/n))\bigr)^n$; for fixed p and k it's $e^{-\mu}\mu^k/k!+O(1/n)$, strongly concentrated about its mean $\mu=p/(1-p)$.]

28. (a) Each arc is a loop with probability $1/n$, so the answer is $[z^k]\,(n-1+z)^m/n^m = \binom{m}{k}(n-1)^{m-k}/n^m$; mean m/n, variance $m(n-1)/n^2$. (A binomial distribution.)

(b) Each vertex has j loops with probability $[z^j]\,(1-p)/(1-(n-1+z)p/n)$, so the answer is $[z^k]\,((1-p)/(1-(n-1+z)p/n))^n$; mean $p/(1-p) = m/n$, variance $(n-(n-1)p)p/(n(1-p)^2) = m(m+n^2)/n^3$. (A negative binomial distribution.)

29. To generate such a random digraph, we could decide all out-degrees in advance, and build an `ARCS` array as in exercise 3; the `TIP` array could then be filled at random.

Alternately, we can generate an equivalent random jungle forest, by beginning with an empty stack and an empty forest. If the stack is empty and we haven't got n internal nodes, create a new root node q, label it with the first unused internal label, and choose its out-degree d; allocate d children, q_1 through q_d, but don't decide as yet whether those nodes will be internal or external; put the children on the stack, with q_1 on top.

If the stack is nonempty, pop off its top element, which is an unlabeled node, say q; give node q a uniformly random label, t. If an internal node has already been labeled t, call node q external. Otherwise call q internal, choose its out-degree d, allocate child nodes q_1 through q_d, and place those yet-unlabeled nodes onto the stack with q_1 on top.

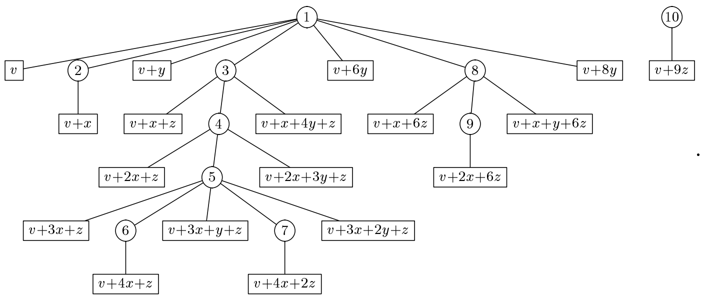
finite-state automaton
Markov process
in-degree
out-degree
binomial distribution
negative binomial distribution

That procedure corresponds to a Markov process with states (s,t), where s is the number of nodes on the stack and t is the number of internal node labels used so far. Begin in state $(0,0)$ and continue until reaching state $(0,n)$. The transition probabilities from state (s,t) are to state $(d,t+1)$ with probability π_d, if $s=0$; or to state $(s-1+d,$ $t+1)$ with probability $(1-t/n)\pi_d$ and to state $(s-1,t)$ with probability t/n, if $s>0$.

extended forest
preorder

30. Every forest F with m edges and n nodes yields an *extended* forest F^+ with n internal nodes, $m+n$ external nodes, and $2m+n$ edges, if we insert an external leaf node before, between, and after every child of an internal node. (Thus a node q with k children $q_1, \ldots, q_k$ in F becomes an internal node q with $2k+1$ children q_0', q_1, q_1', $\ldots$, q_k, q_k' in F^+, where $\{q_1,\ldots,q_k\}$ are internal and $\{q_0',\ldots,q_k'\}$ are external.) Label every internal node of F^+ with its rank in preorder. Label every external child q_j' of q with the sum $\alpha_1+\cdots+\alpha_t$, where t is the number of internal nodes that precede q_j' in preorder of F^+, and α_i is respectively (v,x,y,z) if an arc from internal node q to internal node i is respectively a (loop, back arc, forward arc, cross arc).

For example, here's F^+ when F is the forest of (4):



The generating function g_F is then $(1-p)^n(wp/n)^{n-r}(n!/(s_1\ldots s_r))$ divided by the product of $(1-p\alpha/n)$ taken over all external labels α of F^+, where r is the number of trees and $(s_1\ldots s_r)$ is defined in answer 24.

31. Of course the disconnected forest is forced when $p=0$. It remains most likely until $p=\frac{2}{35}(16-4\sqrt[3]{6}+\sqrt[3]{36})\approx .69$, after which the tallest forest takes the lead.

The five least likely forests, when $p=\epsilon$ and ϵ is very tiny, are $<$ $<$ $<$ $<$. They retain this relative order as p increases to $1-\epsilon$, when they have in fact become the five *most* likely forests. An eight-way occurs when $p=\frac{4}{5}$, at which point each of , , , , , , , occurs with probability $\frac{1}{24}$. Ultimately, when $p>\frac{4}{5}$, the disconnected forest becomes the least likely.

[Curiously, it turns out that is less likely than if and only if $\frac{2}{3}<p<\frac{4}{5}$. And is more likely than if and only if $.52\approx\frac{4}{33}(6-\sqrt{3}\,)<p<\frac{4}{33}(6+\sqrt{3}\,)\approx .94$.]

32. (a) By exercise 30, $g_2^l(z)=4(1-p)^2/(2-p-pz)^2$; $g_2^t(z)=(2-2p+pz)/(2-p)$; $g_2^b(z)=2(1-p)(2-pz)/((2-p)(2-p-pz))$; $g_2^f(z)=2(1-p)(2-pz)/((2-p)(2-p-pz))$; and $g_2^c(z)=(4-6p+3p^2-p^2z)/((2-p)(2-p-pz))$. Notice that $g_2^f(z)=g_2^b(z)$.

(b) The generating function for total arcs is $\bigl((1-p)/(1-pz)\bigr)^n$.

(c) After calculating 14 generating functions g_F as in exercise 31, we find $g_4^{l\,\prime}(1)=p/(1-p)$; but that's no surprise (see exercise 28). Also $g_4^{t\,\prime}(1)=(1536p-4224p^2+4384p^3-2160p^4+536p^5-63p^6)/((4-p)(4-3p)^3(2-p)^2)$; and $g_4^{b\,\prime}(1)=(768p^2-1856p^3+1664p^4-748p^5+226p^6-45p^7)/(2(4-p)(4-3p)^3(2-p)^2(1-p))$. The surprises are that $g_4^{f\,\prime}(1)=g_4^{b\,\prime}(1)$ and $g_4^{c\,\prime}(1)=g_4^{t\,\prime}(1)$. (See exercise 34 — and also exercise 35!)

(d) When $p>p_0$, where $p_0\approx .732792$ is the root of a sixth-degree polynomial.

34. (Solution by S. Janson.) Whenever vertex v has a new descendant w, we generate arcs from w with the geometric distribution, and each of those arcs goes back to v with probability $1/n$. Hence the mean number of back arcs from w to v is $p/(n(1-p))$. Later on, we generate further arcs from v with the geometric distribution, each of which goes to w with probability $1/n$; the mean number of forward arcs from v to w is therefore also $p/(n(1-p))$. These expectations are independent of the past history. Summing over all v and w, with v a proper ancestor of w, the expected total number of back arcs and of forward arcs are both equal to $p/(1-p)$ times the average number of ancestors of a random vertex.

Notice further that the expected number of arcs to a previous vertex in preorder is the expected number of arcs to a later vertex. Indeed, we have $\mathrm{E}(\text{back}+\text{cross}) = \mathrm{E}(\text{tree}+\text{forward})$, even in the much more general model of exercise 29. Since $\mathrm{E}(\text{loop}) = \mathrm{E}(\text{all})/n$ in that model, we have $\mathrm{E}(\text{back}+\text{cross}) = \frac{n-1}{2n}\,\mathrm{E}(\text{all})$.

35. Let $\widehat{G}_1(w,x,z) = \breve{G}_1(w,x,z) = 1/(1-w)$ and, for $n > 1$,

$$\widehat{G}_n(w,x,z) = \sum_{k=1}^{n-1} \frac{\widehat{G}_k(w,x,z)\widehat{G}_{n-k}(w+kz,x,z)}{1-w-(n-1)x};$$

$$\breve{G}_n(w,x,z) = \sum_{k=1}^{n-1} \frac{\breve{G}_k(w+x,x,z)\breve{G}_{n-k}(w+kz,x,z)}{1-w}.$$

S. Janson, in arXiv:2301.04131 [math.CO] (2023), 6 pages, showed that the desired result is equivalent to the identity $\widehat{G}_n(w,x,z) = \breve{G}_n(w,x,z)$ for all $n \ge 1$; then Z. Nie, in arXiv:2301.05704 [math.CO] (2023), 6 pages, proved that identity and a bit more.

36. There clearly are no forward arcs. Loops have the probability generating function $\bigl((n-1+z)/n\bigr)^n$, with mean 1 and variance $1-1/n$.

Arcs are learned one at a time in step D5. The kth arc learned is a tree arc with probability $(n-k)/n$; hence $\prod_{k=1}^{n}\bigl((k+(n-k)z)/n\bigr)$ is the generating function for tree arcs, with mean $(n-1)/2$ and variance $(n^2-1)/(6n)$. [Interesting connections of this generating function to other combinatorial problems have been found by I. Gessel and S. Seo, *Electronic Journal of Combinatorics* **11**, 2 (2004–6), #R27, 1–23.]

Each component of a functional digraph is a cycle with trees feeding into it; hence there's exactly one back arc or loop in every component. The generating function for back arcs plus loops is therefore the generating function for components, namely $t_n(z)/n^n$, where $t_n(z) = [w^n]\,\bigl(1-T(w)\bigr)^z$ is the so-called tree polynomial and $T(w) = \sum_{n\ge1} n^{n-1}w^n/n!$ is the tree function, 2.3.4.4–(30). We have, for example, $t_1(z) = z$, $t_2(z) = z^2+3z$, $t_3(z) = z^3+9z^2+17z$. See D. E. Knuth and B. Pittel, *Proc. Amer. Math. Soc.* **105** (1989), 335–349, who showed that the mean value $t_n'(1)/n^n$ is $\frac12 H_{2n} + O(n^{-1/2})$ and that the variance is also $O(\log n)$. (See also exercise 3.1–14.)

Finally, the number of cross arcs is n minus the number of arcs of other kinds. So its mean value is $n - \frac12(n-1) - \frac12 H_{2n} + O(n^{-1/2}) = \frac12(n+1-H_{2n}) + O(n^{-1/2})$.

37. Let $P_t(z) = \sum_d p(d,t)z^d$, where $p(d,t)$ is the probability that the Markov process of answer 27(b) takes the transition $(d,t) \to (d+1,t+1)$ when started in state $(0,0)$. Then $P_0(z) = 1$; $P_1(z) = \rho_1 + (1-\rho_1)z$; $P_2(z) = (\rho_1+\rho_2-\rho_1\rho_2)\rho_2 + (\rho_1+\rho_2-\rho_1\rho_2)(1-\rho_2)z + (1-\rho_2)(1-\rho_1)z^2$; and in general $P_t(z) = \Omega_{\ge} Q_t(z)P_{t-1}(z) + \rho_t^2 P_{t-1}(\rho_t)$, where $\Omega_{\ge}$ is the operator on Laurent series that removes negative powers of z,

$$\rho_t = \frac{1-p}{1-pt/n}, \quad\text{and}\quad Q_t(z) = (1-\rho_t)z + (\rho_t-\rho_t^2) + (\rho_t^2-\rho_t^3)z^{-1} + \cdots = \frac{z^2(1-\rho_t)}{z-\rho_t}.$$

Janson
Janson
recurrence
Nie
probability generating function
generating function
Gessel
Seo
component
tree polynomial
tree function
Knuth
Pittel
Laurent series

Jacquet
Janson
a.s.
giant
Jacquet
Janson
giant
sinks

Based on these formulas, Philippe Jacquet found that the mean value of d in state (d,t) is asymptotically $\max\bigl(t + ((1-p)/p)n\ln(1-t/n), 0\bigr) + O(1)$, with variance $O(n)$.

In fact, he and Svante Janson [*Leibniz International Proceedings in Informatics* **225** (2022), 11:1–11:15] have carried out a very instructive and comprehensive asymptotic analysis of depth-first search in the geometric model. Let $\lambda = p/(1-p)$ be the average out-degree of each vertex, and let θ be the largest number such that $1-\theta = e^{-\lambda\theta}$. (For example, $\theta = \frac{1}{2}$ when $\lambda = 2\ln 2 \approx 1.3863$; $\theta = \frac{4}{5}$ when $\lambda = \frac{5}{4}\ln 5 \approx 2.0118$.) If $p \le \frac{1}{2}$, or equivalently if $\lambda \le 1$, or equivalently if $\theta = 0$, the trees of the forest a.s. never get very large. Otherwise depth-first search will a.s. construct a tree with $\theta n + \overline{O}(\sqrt{n}\,)$ vertices and height $(\lambda-1-\ln\lambda)n/\lambda + \overline{O}(\sqrt{n}\,)$, where $\overline{O}(\sqrt{n}\,)$ means $O(\sqrt{n}\,\omega(n))$ for every function $\omega(n)$ that increases monotonically to ∞ as $n \to \infty$ (however slowly). This "giant" tree will be the only one in the forest with $\Omega(n)$ nodes. Many small trees will follow it, and few might also precede it.

The average number of trees in the forest is a.s. $\rho n + \overline{O}(\sqrt{n}\,)$, where $\rho = 1-\theta+\frac{\lambda}{2}(1-\theta)^2$. The average number of back arcs is a.s. $\beta n + \overline{O}(\sqrt{n}\,)$, where $\beta = (\lambda-1)\theta - \frac{\lambda}{2}\theta^2$. The average number of tree arcs is a.s. $(1-\rho)n + \overline{O}(\sqrt{n}\,)$. Loops are analyzed in exercise 28(b). The average total number of arcs is, of course λn; notice that $\beta + 1 - \rho = \frac{\lambda}{2}$.

38. In particular, random digraphs of constant out-degree $d > 1$ should be interesting. The paper of Jacquet and Janson has preliminary results; for example, the depth-first forest again has at most one giant tree, of size $\theta n + \overline{O}(\sqrt{n}\,)$, and a total of $\rho n + \overline{O}(\sqrt{n}\,)$ trees on average, if the out-degree distribution has mean λ and finite variance.

39. $\{4,5\}$, $\{6,15\}$, $\{8,14\}$, $\{10,11\}$, and nine singleton components $\{0\}$, …, $\{16\}$.

40. Recurrent states are the elements of strong components that are sinks.

41. Set $\mathtt{REP}(w) \leftarrow 0$ for $\mathtt{VERTICES}(g) \le w < \mathtt{SENT}$; also $p \leftarrow 0$ and $\mathtt{SINK} \leftarrow \Lambda$, where p and $\mathtt{SINK}$ are global variables. Then, while $w > \mathtt{VERTICES}(g)$, set $w \leftarrow w-1$ and execute dfs(w) if $\mathtt{REP}(w) = 0$, where dfs(v) is the following procedure: "Set $p \leftarrow p+1$, $\mathtt{REP}(v) \leftarrow p$, $\mathtt{LINK}(v) \leftarrow \mathtt{SENT}$, and $a \leftarrow \mathtt{ARCS}(v)$, where a is a local variable. While $a \ne \Lambda$, set $u \leftarrow \mathtt{TIP}(a)$, $a \leftarrow \mathtt{NEXT}(a)$, execute dfs$(u)$ if $\mathtt{REP}(u) = 0$, and then if $\mathtt{REP}(u) < \mathtt{REP}(v)$ set $\mathtt{REP}(v) \leftarrow \mathtt{REP}(u)$ and $\mathtt{LINK}(v) \leftarrow \Lambda$. Finally if $\mathtt{LINK}(v) = \mathtt{SENT}$, do step T8; otherwise set $\mathtt{LINK}(v) \leftarrow \mathtt{SINK}$ and $\mathtt{SINK} \leftarrow v$."

(The shortcut from T6 to T8 does not fit nicely into a recursive formulation.)

43. $9 \longrightarrow 7$, $6 \longrightarrow 2$, $2 \longrightarrow 6$, $2 \longrightarrow 3$, $2 \longrightarrow 7$, $6 \longrightarrow 4$, $4 \longrightarrow 6$, $4 \longrightarrow 1$, $1 \longrightarrow 6$, $3 \longrightarrow 1$, $3 \longrightarrow 3$, $9 \longrightarrow 3$, $5 \longrightarrow 9$, $5 \longrightarrow 3$, $8 \longrightarrow 4$, $8 \longrightarrow 3$, $8 \longrightarrow 8$, $5 \longrightarrow 8$, $9 \longrightarrow 5$, $9 \longrightarrow 7$, $0 \longrightarrow 2$. (Tree arcs $v \longrightarrow w$ mature in postorder of w; nontree arcs mature as in answer 12.)

44. Set $r \leftarrow 0$ in step D1. Omit '$a \leftarrow \mathtt{NEXT}(a)$' from step D5. Replace step D6 by

D6.0. [If u is new, don't continue at v.] If $\mathtt{PARENT}(u) = \Lambda$, go to D7.

D6.1. [Stamp a.] Set $r \leftarrow r+1$, $\mathtt{TIME}(a) \leftarrow r$, $a \leftarrow \mathtt{NEXT}(a)$, and go to D4.

And in step D8, go to D6.1 instead of to D4.

45.

Begin 1	$\mathtt{LOW}(1) \leftarrow 1$	$(5 \longrightarrow 3)$	$\mathtt{LOW}(5) \leftarrow 3$	$2 \longrightarrow 6$	$\mathtt{LOW}(6) \leftarrow 6$
$1 \longrightarrow 2$	$\mathtt{LOW}(2) \leftarrow 2$	$(4 \longrightarrow 5)$	$\mathtt{LOW}(4) \leftarrow 3$	$6 \longrightarrow 7$	$\mathtt{LOW}(7) \leftarrow 7$
$2 \longrightarrow 3$	$\mathtt{LOW}(3) \leftarrow 3$	$(3 \longrightarrow 4)$	$\mathtt{LOW}(3) = 3$	$(7 \longrightarrow 4)$	$\mathtt{LOW}(7) \leftarrow 3$
$3 \longrightarrow 4$	$\mathtt{LOW}(4) \leftarrow 4$	$(3 \longrightarrow 1)$	$\mathtt{LOW}(3) \leftarrow 1$	$(6 \longrightarrow 7)$	$\mathtt{LOW}(6) \leftarrow 3$
$4 \longrightarrow 5$	$\mathtt{LOW}(5) \leftarrow 5$	$(2 \longrightarrow 3)$	$\mathtt{LOW}(2) \leftarrow 1$	$(2 \longrightarrow 6)$	$\mathtt{LOW}(2) = 1$

Here '$(u \longrightarrow v)$' means that the arc is maturing. At the end, after $(1 \longrightarrow 2)$, a strong component will be recognized because $\mathtt{LOW}(1) = 1 = \mathtt{PRE}(1)$.

active vertices
shortcut
dense graphs
worst case
Pearce
SETTLED

47. Preorder ABHGMNICDJKEFLQPRO; postorder NMGLFEPRQKJDCOIHBA; strong components DEFJKLPQR, ABCGHIMNO; (LOW(A), ..., LOW(R)) = (1, 1, 2, 9, 9, 11, 1, 1, 2, 9, 9, 11, 3, 3, 3, 10, 10, 11). (Of course the LOW values are actually encoded in REP, via (12), until they are obliterated in step T8.)

48. Deleting $\mathtt{B}\longrightarrow\mathtt{H}$, $\mathtt{H}\longrightarrow\mathtt{G}$, $\mathtt{H}\longrightarrow\mathtt{I}$, $\mathtt{N}\longrightarrow\mathtt{H}$, $\mathtt{K}\longrightarrow\mathtt{E}$, $\mathtt{J}\longrightarrow\mathtt{K}$, $\mathtt{L}\longrightarrow\mathtt{K}$ or $\mathtt{K}\longrightarrow\mathtt{Q}$ eliminates two cycles, leaving five strong components (three singletons). Deleting $\mathtt{C}\longrightarrow\mathtt{D}$ or $\mathtt{O}\longrightarrow\mathtt{P}$ retains two. Otherwise one cycle is broken, leaving three (one of which is a singleton).

50. At step T7, the active vertices are precisely the ancestors of v, and v is becoming inactive. According to Lemma S, a new strong component should be identified at the moment when the first vertex of the sink component of $\hat{g}$ becomes inactive. If $\mathtt{LOW}(v) < \mathtt{PRE}(v)$, there's a downpath from v to an earlier vertex; hence v can't be that first vertex. Conversely, if v is not that first vertex, there's at least one "escape path" $v = v_0 \longrightarrow v_1 \longrightarrow \cdots \longrightarrow v_r$ without forward arcs for which $\mathtt{PRE}(v_r) < \mathtt{PRE}(v)$. We will show that the lexicographically smallest escape path, with respect to $(\mathtt{PRE}(v_0), \mathtt{PRE}(v_1), \ldots)$, is in fact a downpath; hence $\mathtt{LOW}(v) < \mathtt{PRE}(v)$ and $\mathtt{LINK}(v) \neq \mathtt{SENT}$.

Indeed, let $v_{j-1} \longrightarrow v_j$ be the first nontree arc on that escape path. If $\mathtt{PRE}(v_j)$ were greater than $\mathtt{PRE}(v)$, v_j would be a descendant of v, and there'd be a lexicographically earlier path from v to v_j. Hence $\mathtt{PRE}(v_j) < \mathtt{PRE}(v)$, $j = r$, and we have a downpath.

51. Step T8 mustn't find $\mathtt{REP(SINK)} \geq \mathtt{REP}(v)$ when $\mathtt{SINK} = \mathtt{SENT}$.

52. If all vertices have been seen and an arc goes from an active vertex to the current root, all remaining vertices belong to a strong component whose leader is ROOT. (In practice this shortcut often reduces the running time to $O(n)$ on dense graphs.)

[Notice that it is possible to have $u = v = \mathtt{ROOT}$ in unusual cases. There also are cases when early termination is valid but not detected in step T6, such as $1\longrightarrow 2$, $2\longrightarrow 3$, $3\longrightarrow 4$, $4\longrightarrow 2$, $3\longrightarrow 1$, $2\longrightarrow 5$, $5\longrightarrow 4$, ..., when $\mathtt{N}(g) = 5$.]

53. It's possible to keep $\mathtt{REP}(v)$ in a register without fetching it from memory each time. Then, if step T6 doesn't go to T8, (ARC, ARCS, LINK, NEXT, PARENT, REP, TIP) are fetched respectively $(n-r,\ n,\ 2n-s,\ m,\ m+2n,\ m+r-s,\ m)$ times; we store into (ARC, LINK, PARENT, REP) respectively $(n-r,\ 3n+t_6+t_7,\ 2n,\ 2n+t_6+t_7)$ times. Here r is the number of roots, s is the number of strong components, and t_j is the number of times REP is changed in step Tj. Note that $t_6 \leq m$, $t_7 \leq n-s$, and $r \leq s$.

Or, if we don't store into REP unless necessary, we store it $3n-r-s+t_7$ times. That's better in the worst case, but not often actually better.

A slightly faster (but trickier) implementation uses REP alone to encode PRE and LOW: If v has been seen but not settled, we can let $\mathtt{REP}(v)$ be $2\cdot\mathtt{LOW}(v) - 2 + [\mathtt{LOW}(v) \neq \mathtt{PRE}(v)]$; also $\mathtt{REP}(v) = \mathtt{SENT} + \mathtt{N}(g) + v'$ after v is settled. Then $\mathtt{REP}(v) \leftarrow \mathtt{REP}(u) \mid 1$ in T6, $\mathtt{REP}(u) \leftarrow \mathtt{REP}(v) \mid 1$ in T7, and T7 goes to T8 if $\mathtt{REP}(v)$ is even.

[See D. J. Pearce, *Information Processing Letters* **116** (2016), 47–52.]

54. Print $\mathtt{NAME}(v)$ at the beginning of step T8, then NAME(SINK) while $\mathtt{REP(SINK)} \geq \mathtt{REP}(v)$. The vertices appear in reverse postorder of the depth-first forest, because the SINK stack is in postorder from bottom to top.

55. Set $\mathtt{SETTLED} \leftarrow \Lambda$ in step T1, where SETTLED heads a new stack. Change step T8 to

T8′. [New strong component.] If $\mathtt{REP(SINK)} < \mathtt{REP}(v)$, set $\mathtt{REP}(v) \leftarrow \mathtt{SENT} + v$, $\mathtt{LINK}(v) \leftarrow \mathtt{SETTLED}$, and $\mathtt{SETTLED} \leftarrow v$ (a singleton component). Otherwise set $\mathtt{LINK}(v) \leftarrow \mathtt{SETTLED}$, $\mathtt{SETTLED} \leftarrow \mathtt{SINK}$; then while $\mathtt{REP(SINK)} \geq \mathtt{REP}(v)$, set $t \leftarrow \mathtt{SINK}$, $\mathtt{REP(SINK)} \leftarrow \mathtt{SENT} + v$ and $\mathtt{SINK} \leftarrow \mathtt{LINK(SINK)}$; finally set

$\mathtt{REP}(v) \leftarrow \mathtt{SENT} + v$ and $\mathtt{LINK}(t) \leftarrow v$. (We've essentially moved vertices from SINK to SETTLED, together with v, using the ordering of exercise 54 in reverse.)

Let each vertex have a field called FROM. (This field could share space with ARC.) After Algorithm T terminates, set $\mathtt{FROM}(v) \leftarrow \Lambda$ for all v. Then set $v \leftarrow \mathtt{SETTLED}$, and do the following while $v \neq \Lambda$: Set $u \leftarrow \mathtt{REP}(v) - \mathtt{SENT}$ and $\mathtt{FROM}(u) \leftarrow u$. For $a \leftarrow \mathtt{ARCS}(v)$, while $a \neq \Lambda$ set $w \leftarrow \mathtt{REP}(\mathtt{TIP}(a)) - \mathtt{SENT}$ and, if $\mathtt{FROM}(w) \neq u$, set $\mathtt{FROM}(w) \leftarrow u$ and print $\mathtt{NAME}(v) \longrightarrow \mathtt{NAME}(\mathtt{TIP}(a))$ as a representative of the arc in $\widehat{g}$ from component $\mathtt{NAME}(u)$ to component $\mathtt{NAME}(w)$, then set $a \leftarrow \mathtt{NEXT}(a)$. Finally set $v \leftarrow \mathtt{LINK}(v)$.

[See *The Stanford GraphBase* (1993), *roget_components*, §17.]

Stanford GraphBase
ear decomposition
NP-complete
Hamiltonian cycle
depth-first jungle
strata
cycle arc
leader

56. Change (11) so that, if $\mathtt{LOW}(v) < \mathtt{PRE}(v)$ and $\mathtt{LOW}(v)$ was determined by the nontree arc $v \longrightarrow u$, then $\mathtt{LINK}(v) = u$. In step T6, change '$\mathtt{LINK}(v) \leftarrow \Lambda$' to '$\mathtt{LINK}(v) \leftarrow u$'. [But keep '$\mathtt{LINK}(u) \leftarrow \Lambda$' in step T7.] If step T6 takes the "last component" shortcut, print the nontree arc $\mathtt{NAME}(v) \longrightarrow \mathtt{NAME}(\mathtt{ROOT})$ if $v \neq \mathtt{ROOT}$. Just after finding $\mathtt{LINK}(v) \neq \mathtt{SENT}$ in step T7, print the nontree arc $\mathtt{NAME}(v) \longrightarrow \mathtt{NAME}(\mathtt{LINK}(v))$ if $\mathtt{LINK}(v) \neq \Lambda$.

Those nontree arcs (part of an "ear decomposition") suffice to preserve the LOW values within strong components. Also print the tree arcs within strong components, by printing $\mathtt{NAME}(\mathtt{PARENT}(\mathtt{SINK})) \longrightarrow \mathtt{NAME}(\mathtt{SINK})$ in the while loop of step T8 or T8′.

These nontree and tree arcs within strong components, together with the arcs between components in exercise 55, give the desired subgraph.

This efficiently found subgraph isn't always minimum. For example, it removes no arcs from the digraph $1 \longrightarrow 2$, $2 \longrightarrow 3$, $2 \longrightarrow 4$, $3 \longrightarrow 1$, $4 \longrightarrow 3$, although a clairvoyant algorithm could remove $2 \longrightarrow 3$. Indeed, the problem of minimizing arcs is NP-complete, because n arcs suffice to make g strongly connected if and only if g has a Hamiltonian cycle.

The subgraph that's found isn't even minimum, in general, if we start with a given depth-first jungle and insist on including all of its tree arcs within strong components. (Consider, for example, $1 \longrightarrow 2$, $2 \longrightarrow 1$, $2 \longrightarrow 3$, $3 \longrightarrow 1$.) But it's pretty good.

58. (a) Vertex G is in no cycle, so its period is ∞. But the others all have period 3, since the arcs not involving G are a subset of $\{\mathtt{A},\mathtt{F}\} \longrightarrow \{\mathtt{B},\mathtt{D},\mathtt{E}\} \longrightarrow \{\mathtt{C},\mathtt{H}\} \longrightarrow \{\mathtt{A},\mathtt{F}\}$.

(b) Let v have period $p = \gcd P$, where $P = \{p_1, p_2, \ldots\}$ is the set of lengths of cycles containing v. Similarly, let w have period $q = \gcd Q = \{q_1, q_2, \ldots\}$; and assume that there are paths $v \longrightarrow^* w$ and $w \longrightarrow^* v$ of lengths a and b. Every cycle $w \longrightarrow^* w$ of length q_k containing w is part of a cycle $v \longrightarrow^* w \longrightarrow^* w \longrightarrow^* v$ of length $a + q_k + b$ containing v. Since p divides both $a + q_k + b$ and $a + b$, it divides q_k. Hence p is a common divisor of Q, and $p \le q$. Similarly $q \le p$. (Notice that we can also partition v's strong component into a cycle of p disjoint "strata," as we did for $p = 3$ in (a).)

(c) Say that the arc $v \longrightarrow u$ of a depth-first jungle is a *cycle arc* if it's neither a tree arc nor a link between different strong components. (This condition holds if and only if $\mathtt{PARENT}(u) \neq \Lambda$ and $\mathtt{REP}(u) < \mathtt{SENT}$ in step T6.) Then every cycle arc is part of some cycle; and every cycle $w = w_0 \longrightarrow w_1 \longrightarrow \cdots \longrightarrow w_t = w$ is composed entirely of tree arcs and cycle arcs. Let $T(w) = \{t \mid w \text{ is in a cycle of length } t\}$ and $J(w) = \{\,\mathrm{jump}(v,u) \mid v \longrightarrow u \text{ is a cross arc in the strong component of } w\}$, where

$$\mathrm{jump}(v,u) \;=\; \mathtt{DEPTH}(v) + 1 - \mathtt{DEPTH}(u).$$

Lemma: $\gcd T(w) = \gcd J(w)$. *Proof:* Let $v \longrightarrow u$ be a cycle arc. By (b) we may assume that w is the leader of its strong component, so that there are paths $w \longrightarrow^* u$ and $w \longrightarrow^* v$ by tree arcs alone, and w is reachable from u. Let the cycle $w \longrightarrow^* u \longrightarrow^* w$ have length t. Then the cycle $w \longrightarrow^* v \longrightarrow u \longrightarrow^* w$ has length $t + \mathrm{jump}(v,u)$; hence any common divisor of $T(w)$ divides the difference between these lengths, and must be a common divisor of $J(w)$. Conversely, given any cycle of length t containing w, we have $\mathrm{jump}(w_j, w_{j+1}) = 0$ when $w_j \longrightarrow w_{j+1}$ is a tree arc. Hence any common divisor of $J(w)$ divides $\sum_{j=0}^{t-1} \mathrm{jump}(w_j, w_{j+1}) = t$, and must be a common divisor of $T(w)$.

The desired algorithm extends Algorithm T, but without the "shortcut" to T8 in step T6. Every vertex has two new fields, `GCD` and `DEPTH`; we initialize $\mathtt{GCD}(v) \leftarrow 0$ and $\mathtt{DEPTH}(v) \leftarrow \mathtt{DEPTH(PARENT}(v)) + 1$ in step T3, where $\mathtt{DEPTH(SENT)} = 0$. In step T6 we set $\mathtt{GCD}(v) \leftarrow \gcd(\mathtt{GCD}(v), \mathrm{jump}(v, u))$ whenever $\mathtt{PARENT}(u) \neq \Lambda$ and $\mathtt{REP}(u) < \mathtt{SENT}$. In step T8 we calculate $\gcd\{\mathtt{GCD}(w) \mid w \in S\}$, where $S = \{w \mid w$ is removed from $\mathtt{SINK}\} \cup v$ is the new strong component; this gcd is the period of S, and it can be stored in the `GCD` fields. In step T9 we set $\mathtt{GCD}(v) \leftarrow \gcd(\mathtt{GCD}(v), \mathtt{GCD}(u))$ before $v \leftarrow u$.

In digraph (a), for example, beginning in T2 with vertex `H`, we find ($\mathtt{DEPTH(A)}$, $\mathtt{DEPTH(B)}$, ..., $\mathtt{DEPTH(H)}$) = (2, 3, 4, 6, 3, 5, 4, 1). The jumps at cycle arcs $\mathtt{C} \longrightarrow \mathtt{A}$, $\mathtt{F} \longrightarrow \mathtt{B}$, $\mathtt{D} \longrightarrow \mathtt{H}$, $\mathtt{E} \longrightarrow \mathtt{C}$, $\mathtt{H} \longrightarrow \mathtt{F}$ are respectively $5-2$, $6-3$, $7-1$, $4-4$, $2-5$.

[This algorithm appeared in D. Knuth, *Lecture Note* **2** (Inst. of Math., Univ. of Oslo, 1973), 14–17. R. E. Tarjan, in unpublished lecture notes (CS259, Stanford, Spring 1976), observed that the worst-case running time is $O(m+n)$, because the gcd of a set $D \subseteq \{1, \ldots, n\}$ can be computed in $O(n)$ steps from the formula $\gcd(D) = \gcd(D \setminus x \cup (x-y))$, where x and y are the largest and second-largest elements of D. See also M. J. Atallah, *SICOMP* **11** (1982), 567–570.]

61. Several thousand trials with the parameters of exercise 25 (or 26) gave 14000 ± 120 (or 205600 ± 530) strong components of maximum size 986100 ± 120 (or 794400 ± 530); $t_6 = 789000 \pm 3000$ (or 611500 ± 850), $t_7 = 827000 \pm 700$ (or 636000 ± 680). Essentially all 5000000 arcs were examined. But if we have, say, only 100000 vertices (and change p to 50/51), the number of arcs examined drops to 1300000 ± 160000 and is probably $O(n)$.

65. (a) The only such oriented paths (*not* easy to find by hand!) are $(5,5) \longrightarrow (3,7) \longrightarrow ((8,2) \longrightarrow (2,8) \longrightarrow (6,4) \longrightarrow)^p (6,2) \longrightarrow \mathtt{OUT}$, for $p \geq 1$.

(b) The results shown here are obtained when arcs are visited in lexicographically decreasing order (for example, $(1,2) \longrightarrow (4,5)$, $(1,2) \longrightarrow (4,2)$, $(1,2) \longrightarrow (1,5)$, $(1,2) \longrightarrow \mathtt{OUT}$). There are 38 strong components, whose leaders are indicated in bold type. The largest is **aa** (50 vertices); then **a7**, **5a**, **04**, **77**, **82**, **83**, **49**, **4a**, **87**, **91**, **a6**, **a8**, with respectively (18, 5, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2) vertices; and 25 singletons including `OUT`.

a7	**01**	a7	a7	**04**	**05**	a7	a7	**08**	04	91
10	a7	a7	a7	a7	**15**	a7	a7	**18**	**19**	a7
aa	aa	aa	a7	aa	aa	a6	77	82	aa	aa
aa	**31**	aa	aa	aa	aa	aa	**37**	83	aa	aa
40	49	4a	aa	aa	a7	aa	a7	**48**	**49**	**4a**
5a	5a	a7	**53**	04	**55**	**56**	5a	**58**	5a	**5a**
aa	aa	**62**	aa	82	aa	aa	a7	**68**	aa	aa
aa	**71**	77	aa	**74**	aa	aa	**77**	**78**	aa	aa
aa	87	**82**	**83**	aa	aa	**86**	**87**	83	aa	aa
aa	**91**	**92**	aa	aa	**95**	aa	a8	**98**	aa	aa
aa	aa	aa	aa	aa	aa	**a6**	**a7**	**a8**	aa	**aa**

(c) There are, of course, $122 - 38 = 84$ tree arcs such as $(1,2) \longrightarrow (4,5)$. The digraph g has 424 arcs, of which 183 are links between different strong components. That leaves $424 - 183 - 84 = 157$ nontree arcs within components; answer 56 prints 40

Historical notes
Knuth
Tarjan
gcd
Atallah

of them, such as $(5,2) \longrightarrow (1,2)$ within **a7**. The reduced graph $\widehat{g}$ has 97 arcs; answer 55 prints 97 representative arcs from g, such as $(1,2) \longrightarrow (4,2)$ from **a7** to **4a**.

[This exercise was inspired by Sam Loyd's original whirlpool maze in *Tit-Bits* **32** (31 July 1897), 327, later refashioned as his famous "Back from the Klondike" prize puzzle in *New York Journal and Advertiser* (24 April 1898), 32. See Martin Gardner, *Scientific American* **235**, 4 (October 1976), 131–132.]

70. Suppose X and Y are different strong components of a tournament. If $x \in X$ and $y \in Y$ we have either $x \longrightarrow y$ or $y \longrightarrow x$; say $x \longrightarrow y$. Then we must have $x' \longrightarrow y'$ for every $x' \in X$ and $y' \in Y$. Consequently the strong components can be linearly ordered, from least to greatest; all arcs between components go "the same way." And the strong components are the same as the weak components.

[Hence there are $n!\,[z^n]\,\bigl(2 - 1/\sum_{k\ge 0} 2^{\binom{k}{2}} z^k/k!\bigr)$ strongly connected tournaments on n labeled vertices. See E. M. Wright, *Glasgow Math. J.* **11** (1970), 97–101.]

80. True: Clearly $u \Leftrightarrow v \Leftrightarrow w$ implies $u \Leftrightarrow w$; $u \Leftrightarrow v \parallel w$ implies $u \parallel w$; and $u \parallel v \Leftrightarrow w$ implies $u \parallel w$. The *shortest* chains (13) will therefore have the stated form.

[This is Lemma 2 in the paper by R. L. Graham, D. E. Knuth, and T. S. Motzkin, *Discrete Mathematics* **2** (1972), 17–29, where weak components were first defined. See also the notion of "weak ordering" in exercise 5.3.1–3.]

81. Since $v \parallel w$ if and only if $v \not\Rightarrow w$ and $w \not\Rightarrow v$, answer 80 proves the "only if" part.

Suppose there are nonpaths both ways. If the shortest nonpath from v to w has $r \ge 2$, we actually have $v_j \parallel v_{j+1}$ for $0 \le j < r$, because $u \not\Rightarrow v \not\Rightarrow w$ and $u \Rightarrow w$ implies $v \not\Rightarrow u$ and $w \not\Rightarrow v$. Therefore we have $v \asymp w$ whenever the shortest nonpath in either direction has length exceeding 1. Otherwise, however, $v \not\Rightarrow w$ and $w \not\Rightarrow v$.

82. Let the vertices be $\{0, 1, \ldots, 1000000\}$, with $v \longrightarrow w$ if and only if $w \ge v + 2$. Then $v \parallel w$ if and only if $|v - w| = 1$.

83. True. (Assume that $u \asymp v$, $v \Rightarrow w$, and $u \not\Rightarrow w$. If $u \Leftrightarrow v$ then $u \Rightarrow w$. Hence $u = u_0 \parallel \cdots \parallel u_{r-1} \parallel u_r = v$ for some $r \ge 1$. If $v \Leftrightarrow w$ then $u_{r-1} \parallel w$. Otherwise $w \not\Rightarrow u_r \not\Rightarrow \cdots \not\Rightarrow u_0$ is a nonpath from w to u, and $u \not\Rightarrow w$ is a nonpath the other way.)

84. A digraph with such an (L, R) partition never has $u \Leftrightarrow v$ or $u \parallel v$ when u and v lie in different parts; hence it isn't weakly connected.

Conversely, if $v \not\Rightarrow u$, let L be the set of all vertices w such that, whenever $v \asymp v'$, we have $w \Rightarrow v'$ but $v' \not\Rightarrow w$; and let R be the other vertices. Suppose $w \in L$ and $w' \in R$. Then there's a $v' \asymp v$ such that either (a) $w' \not\Rightarrow v'$ or (b) $w' \Leftrightarrow v'$. Case (b), or case (a) with $v' \not\Rightarrow w'$, give $w' \asymp v$; then (i) and (ii) hold by definition of L. Otherwise $w \Rightarrow v' \Rightarrow w'$ proves (i); and (ii) must also be true, lest $w' \Rightarrow w \Rightarrow v'$.

Clearly $v \in R \ne \emptyset$, since $v \not\Rightarrow v$ is false. Finally, $u \in L \ne \emptyset$: For if $v \asymp v' \Rightarrow u$ and $v \not\Rightarrow u$, exercise 83 tells us $u \asymp v$. Hence $v' \not\Rightarrow u$, and we must have $u \Rightarrow v'$ lest $u \asymp v$.

85. If a dag D is not weakly connected, the (L, R) partition of Lemma W shows that the first $|L|$ elements of any topological sort are the elements of L.

Conversely, if D is weakly connected and $\{u_1, v_1\}$ are any two of its sources, we will show by induction on n that such sequences $u_1 \ldots u_n$ and $v_1 \ldots v_n$ exist. The result is clear if $n = 2$. If $n > 2$, let s be a source, and let $(W_1, \ldots, W_r)$ be the weak components of $D \setminus s$. If $r = 1$, suppose $u_1 \ldots u_{n-1}$ and $v_1 \ldots v_{n-1}$ work for $D \setminus s$. If u_1 and v_1 are sources of D, then $u_1 s u_2 \ldots u_{n-1}$ and $v_1 s v_2 \ldots v_{n-1}$ work for D. Otherwise we can assume $s \longrightarrow u_1$ and $s \not\longrightarrow v_1$; then $s u_1 \ldots u_{n-1}$ and $v_1 s v_2 \ldots v_{n-1}$ work.

If $r > 1$, the sources of D are s and the sources of W_1; also $s \longrightarrow w$ implies $w \in W_r$. Let $u_1 \ldots u_{n-1}$ and $v_1 \ldots v_{n-1}$ be any two topological sorts of $D \setminus s$. If W_r is a single

historical notes
Loyd
Klondike
Gardner
linearly ordered
weak components
generating functions
Wright
Graham
Knuth
Motzkin
historical note
sources

vertex t, we can use $su_1 \ldots u_{n-1}$ and $v_1 \ldots v_{n-1}s$; also $u_1 s u_2 \ldots u_{n-1}$ and $v_1 \ldots v_{n-1}s$ if $u_1 \neq v_1$. Otherwise W_r has at least two sources, u' and v', with $s \not\longrightarrow v'$, and two topological sorts $(u' \ldots u'', v' \ldots v'')$. We can assume that $u_1 \ldots u_{n-1}$ ends with $u' \ldots u''$ and $v_1 \ldots v_{n-1}$ ends with $v' \ldots v''$. Then $su_1 \ldots u_{n-1}$ or perhaps $u_1 s u_2 \ldots u_{n-1}$ can be used with $v_1 \ldots v' s \ldots v''$ (meaning that s is inserted just after v' in $v_1 \ldots v_{n-1}$).

(In exercise 82, use 0 2 1 . . . 1000000 999999 and 1 0 . . . 999999 999998 1000000.)

SETTLED
no consecutive 1s
consecutive 1s
Morse code
11-free binary strings
OEIS

86. In $r-1$ ways, with $L = W_1 \cup \cdots \cup W_j$ and $R = W_{j+1} \cup \cdots \cup W_r$ for $1 \le j < r$.

88. (a) Cases j and j' together arise in $2^{n_j+\cdots+n_r} - 2^{n_{j+1}+\cdots+n_r}$ ways. Case j accounts for $2^{n_j-s_j+n_{j+1}+\cdots+n_r}$ of these. (And of course Case $r+1$ occurs in just one way.)

(b) In Case 1, $\{v\}$ becomes a singleton weak component, with v as both source and sink; all else stays unchanged (except each W_j becomes W_{j+1}). In Cases j' and $j+1$, a new weak component is formed from vertices $\{v\} \cup W_1 \cup \cdots \cup W_j$; its sources are v and the unhit sources of W_1; its sinks are the sinks of W_j, together with v in Case $j+1$ only. Components $W_{j+1}, \ldots, W_r$ are unchanged, but their "subscript" decreases by $j-1$.

89. In the `SETTLED` list, strong components begin at vertices v with `REP`(v) = `SENT`$+v$. Weak components begin at vertices `WP`, `WLINK(WP)`, and so on until `SENT` occurs.

90. All steps clearly take linear time except perhaps W3 and W4. Step W3 is linear because we lose a weak component whenever `WP` advances.

The interesting case is step W4, which would be nonlinear if we traversed a long list in order to purge unhit sources. When that loop looks at vertex u, we've either had an arc $v \longrightarrow u$ (`WHIT`$(u) = 1$), or we exit because `WHIT`$(u) = 0$, or we remove u from a list of sources because `HIT`$(u) = 1$. A vertex is never removed from such a list twice.

92. If we map $(s_j, n_j) \mapsto (10)^{s_j-1}(01)^{n_j-s_j}00$, we obtain a one-to-one correspondence between such sequences and binary strings of length $2n$ that have no consecutive 1s and end with 00. And there's a well known "Morse code" correspondence between $[0\,..\,F_n)$ and 11-free binary strings of length $n-2$; see, for example, *CMath* (6.115).

93. (a) When $n = 1$, $g_\sigma(z) = 1$. If we know all the generating functions for n, we can use the case analysis of exercise 88 to compute the contribution of each $g_\sigma(z)$ to the generating functions for $n+1$. Let σj and $\sigma j'$ be the sequences that correspond to augmentation in Cases j and j'. In Case $1'$ we also need to know h, the number of sources hit in W_1; we shall denote that sequence by $\sigma 1' h$. Let $n'_j = n_j + \cdots + n_r$.

With that notation, $g_\sigma = g_\sigma(z)$ contributes $z^{s_j}(1+z)^{n'_j - s_j} g_\sigma$ to $g_{\sigma j}$, for $1 \le j \le r$; $\binom{s_1}{h} z^h (1+z)^{n'_1 - s_1} g_\sigma$ to $g_{\sigma 1'(s_1-h)}$, for $0 < h < s_1$; $\bigl((1+z)^{n'_1-s_1} - (1+z)^{n'_2}\bigr) g_\sigma$ to $g_{\sigma 1' 0}$; $\bigl((1+z)^{n'_j} - z^{s_j}(1+z)^{n'_j - s_j} - (1+z)^{n'_{j+1}}\bigr) g_\sigma$ to $g_{\sigma j'}$ for $1 < j \le r$; and g_σ to $g_{\sigma(r+1)}$.

For example, the generating functions for $n = 2$ are $g_{1111} = z$, $g_{12} = 0$, $g_{22} = 1$, if we write $(s_1, n_1; \ldots; s_r, n_r)$ without punctuation. Let $\sigma = (2,3;2,3)$; it turns out that $g_{2323} = 9z^6 + 45z^7 + 90z^8 + 90z^9 + 45z^{10} + 9z^{11}$. This g_σ contributes $z^2(1+z)^4 g_\sigma$ to $g_{\sigma 1} = g_{112323}$ and $z^2(1+z) g_\sigma$ to $g_{\sigma 2} = g_{3423}$; $2z(1+z)^4 g_\sigma$ to $g_{\sigma 1' 1} = g_{2423}$; $\bigl((1+z)^4 - (1+z)^3\bigr) g_\sigma$ to $g_{\sigma 1' 0} = g_{3423}$ and $\bigl((1+z)^3 - z^2(1+z) - 1\bigr) g_\sigma$ to $g_{\sigma 2'} = g_{37}$; g_σ to $g_{\sigma 3} = g_{37}$. Notice that, in general, we have $\sigma 1' 0 = \sigma 2$, and $\sigma j' = \sigma(j+1)$ for $1 < j \le r$.

(The generating function g_σ will be zero if and only if $s_j = 1$ and $n_j > 1$ for some j; hence the number of nonzero cases is not F_{2n} but (1, 2, 5, 12, 28, . . .) for n = (1, 2, 3, 4, 5, . . .), a sequence whose growth rate is approximately 2.32472^n instead of 2.61803^n.)

(b) The sum $\sum_{s=1}^{n} g_{(sn)}(1)$ is (1, 1, 4, 31, 474, 14357, 865024, 103931595, . . . , 30350576522944869091259632762857142 7) for $1 \le n \le 16$; see OEIS A350608.

(c) Let $g_n^{(r)}$ be the sum of g_σ for all σ with r weak components and $n_1 + \cdots + n_r = n$; for example, $g_4^{(2)} = 8z^3 + 7z^4$, and $g_n^{(n)} = z^{n-1}(1+z)^{(n-1)(n-2)/2}$. Then the probability of r weak components when each arc is independently present with probability p is $(1-p)^{n(n-1)/2} g_n^{(r)}(p/(1-p))$. In particular, when $1 \le r \le n = 16$ we get the following approximate results: (i) (.007, .017, .034, .057, .082, .106, .123, .130, .124, .108, .085, .060, .037, .019, .008, .002); (ii) (.228, .154, .167, .140, .108, .078, .052, ..., .00003); (iii) (.808, .095, .057, .023, .010, .004, .001, ..., 10^{-7}); (iv) (.99999990, 10^{-7}, 10^{-8}, 10^{-9}, 10^{-10}, 10^{-11}, $2 \cdot 10^{-12}$, ..., $9 \cdot 10^{-19}$). [See OEIS A350609.]

95. Suppose D is a weakly connected subgraph of $K_n^{\rightarrow}$. For $1 \le k < n$, there are vertices (u_k, v_k) with $u_k \le k < v_k$ with $u_k \not\Rightarrow v_k$, by Lemma W. Let's choose them so that $v_k - u_k$ is minimum. Then $(u_k, v_k) \ne (u_{k'}, v_{k'})$ when $k < k'$; otherwise we'd have $u_k \le k < k' < v_k$ and $u_k \Rightarrow k+1 \Rightarrow v_k$, a contradiction. Hence we've found at least $n-1$ arcs $u_k \longrightarrow v_k$ that are *not* in D, and D has at most $\binom{n}{2} - (n-1) = \binom{n-1}{2}$ arcs.

Given $a_2 \ldots a_{n-1}$, let $b_1 = c_n = 0$ and $b_j = 1 + b_{j-1}$ if $a_j =$ `-`, otherwise $b_j = 0$; $c_j = 1 + c_{j+1}$ if $a_j =$ `+`, otherwise $c_j = 0$. The corresponding digraph D omits the arcs from $j - b_j$ to $j + 1 + c_{j+1}$, for $1 \le j < n$.

For example, the digraph that corresponds to `--++0+-00-` omits $1 \longrightarrow 2$, $1 \longrightarrow 3$, $1 \longrightarrow 6$, $4 \longrightarrow 6$, $5 \longrightarrow 6$, $6 \longrightarrow 8$, $7 \longrightarrow 8$, $7 \longrightarrow 9$, $9 \longrightarrow 10$, $10 \longrightarrow 11$, $10 \longrightarrow 12$.

99. ...

999. ...

INDEX AND GLOSSARY

I, for my part, venerate the inventor of indexes;
and I know not to whom to yield the preference,
either to Hippocrates, who was the first great anatomiser of the human body,
or to that unknown labourer in literature,
who first laid open the nerves and arteries of a book.

— ISAAC D'ISRAELI, *Miscellanies* (1796)

When an index entry refers to a page containing a relevant exercise, see also the *answer* to that exercise for further information. An answer page is not indexed here unless it refers to a topic not included in the statement of the exercise.