Deconstructing Coroutines

Donald E. Knuth and Frank Ruskey

(to Ole-Johan Dahl on his 70th birthday)

Abstract. We study an interesting family of cooperating coroutines, which is able to generate all patterns of bits that satisfy certain fairly general ordering constraints, changing only one bit at a time. (More precisely, the directed graph of constraints is required to be cycle-free when it is regarded as an undirected graph.) If the coroutines are implemented carefully, they yield an algorithm that needs only a bounded amount of computation per bit change, thereby solving an open problem in the field of combinatorial pattern generation.

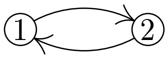
Much has been written about the transformation of procedures from recursive to iterative form, but little is known about the more general problem of transforming *coroutines* into equivalent programs that avoid unnecessary overhead. The present paper attempts to take a step in that direction by focusing on a reasonably simple yet nontrivial family of cooperating coroutines for which significant improvements in efficiency are possible when appropriate transformations are applied. The authors hope that this example will inspire other researchers to develop and explore the potentially rich field of coroutine transformation.

Coroutines are analogous to subroutines, but they are symmetrical with respect to caller and callee: When coroutine A invokes coroutine B, the action of A is temporarily suspended and the action of B resumes where B had most recently left off. (See, for example, Section 1.4.2 of [3].) Programming languages such as SIMULA I [1] and its object-oriented descendants have made it easy for programmers to specify families of parameterized coroutines that cooperate with each other in natural but nontrivial ways. In this paper we will study examples in which a compiler can transform such programs into optimized code, just as compilers can often transform recursive procedures into iterative routines that require less space and/or time.

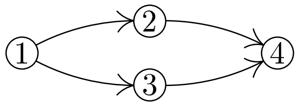
The ideas presented here were motivated by applications to the exhaustive generation of combinatorial objects. For example, consider a coroutine that wants to look at all permutations of n elements; it can call repeatedly on a permutation-generation coroutine to produce the successive arrangements. The latter coroutine repeatedly forms a new permutation and calls on the former coroutine to inspect the result. The permutation coroutine has its own internal state — its own local variables and its current location in an ongoing computational process — so it does not consider itself to be a "subroutine" of the inspection coroutine. The permutation coroutine might also invoke other coroutines, which in turn are computational objects with their own internal states.

We shall consider the problem of generating all n-tuples $a_1a_2\ldots a_n$ of 0s and 1s with the property that $a_j \le a_k$ whenever $j \to k$ is an arc in a given directed graph with n vertices. These n-tuples are supposed to form a "Gray path," in the sense that only one bit a_j should change at each step.

The general problem just stated does not always have a solution. For example, if the given digraph is


,

then we are asking for a way to generate the tuples 00 and 11 by changing only one bit at a time, and this is clearly impossible. Even if we stipulate that there should be no directed cycles, we might encounter an example like



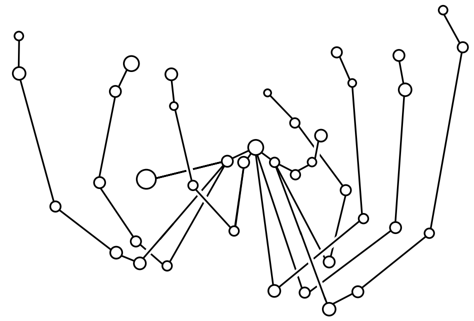
,

for which the Gray constraint cannot be achieved, because the corresponding 4-tuples

$$0000, 0001, 0011, 0101, 0111, 1111$$

include four of even weight and two of odd weight; a Gray path must alternate between even and odd. Reasonably efficient methods for solving the problem without Grayness are known [10, 11], but we want to insist on single-bit changes.

Therefore we shall restrict consideration to directed graphs that are *totally acyclic*, in the sense that they contain no cycles even if the directions of the arcs are ignored. Each component of such a graph is a free tree in which a direction has been assigned to each branch between two vertices. Such digraphs are called *spiders*, because of their resemblance to arachnids:



(In this diagram, as in others below, we assume that all arcs are directed upwards.) The general problem of finding all $a_1 \ldots a_n$ such that $a_j \le a_k$ when $j \to k$ in such a digraph is formally called the task of "generating all ideals of an acyclic poset"; it also is called, informally, "spider squishing."

Sections 1–3 of this paper discuss simple examples of the problem in preparation for Section 4, which presents a constructive proof that suitable Gray paths always exist. The proof of Section 4 is implemented with coroutines in Section 5, and Section 6 discusses the nontrivial task of getting all the coroutines properly launched.

Section 7 describes a simple technique that is often able to improve the running time. A slight generalization of that technique leads in Section 8 to an efficient coroutine-free implementation. Additional optimizations, which can be used to construct a loopless algorithm for the spider-squishing problem, are discussed in Section 9.

1. The simplest case. Let's begin by imagining a long line of friendly trolls. This line ends with T_0 and extends infinitely far to the left, with troll T_k preceded by T_{k+1} for all $k \ge 0$. Each troll carries a lamp that is either off or on; he also can be either awake or asleep. Initially all the trolls are awake, and all their lamps are off.

Changes occur to the system when a troll is "poked," according to the following simple rules: If T_k is poked when he is awake, he changes the state of his lamp from off to on or vice versa; then he becomes tired and goes to sleep. Later, when the sleeping T_k is poked again, he wakes up and pokes T_{k+1}, without making any change to his own lamp.

At periodic intervals an external driving force D pokes the rightmost troll T_0, initiating a chain of events that culminates in one lamp changing its state. The process begins as follows, if we use the digits 0 and 1 to represent lamps that are respectively off or on, and if we underline the digit of a sleeping troll:

$\ldots 0000$	Initial state
$\ldots 000\underline{1}$	D pokes T_0
$\ldots 00\underline{1}1$	D pokes T_0, who wakes up and pokes T_1
$\ldots 00\underline{1}\underline{0}$	D pokes T_0
$\ldots 0\underline{1}10$	D pokes T_0, who pokes T_1, who pokes T_2
$\ldots 0\underline{1}1\underline{1}$	D pokes T_0,
$\ldots 0\underline{1}\underline{0}1$	D pokes T_0, who pokes T_1

The sequence of underlined versus not-underlined digits acts essentially as a binary counter. And the sequence of digit patterns, in which exactly one bit changes at each step, is a *Gray binary* counter, which follows the well-known Gray binary code; it also corresponds to the process of replacing rings in the classic Chinese ring puzzle [4]. (This troll-oriented way to generate Gray binary code was presented by the first author in a lecture at the University of Oslo in October, 1972 [5].)

Suppose now that only finitely many trolls are present. In this case we shall name them $T_1, T_2, \ldots, T_n$ from left to right, so that the driving force D pokes T_n, while T_1 has no left-hand neighbor. Now it makes sense for T_k to pass a message back to his right-hand neighbor T_{k+1} (or to D, when $k = n$), telling whether a lamp has changed state. Such extended rules can be expressed in an ad hoc Algol-like language as follows:

```
   Boolean coroutine poke[k];
      while true do begin
awake: a[k] := 1 − a[k];  return true;
asleep: if k > 1 then return poke[k − 1] else return false;
      end.
```

Coroutine *poke*$[k]$ describes the action of T_k, implicitly retaining its own state of wakefulness. Thus, *poke*$[k]$ will resume its program at label 'asleep' when it is next activated after having executed the statement '**return true**'; and it will resume at label 'awake' when it is next activated after '**return** *poke*$[k-1]$' or '**return** *false*'.

The system therefore goes through the following steps when $n = 2$:

00	Initial state
$0\underline{1}$	$poke[2] = true$
$\underline{1}1$	$poke[2] = poke[1] = true$
$\underline{1}\underline{0}$	$poke[2] = true$
10	$poke[2] = poke[1] = false$
$1\underline{1}$	$poke[2] = true$
$\underline{0}1$	$poke[2] = poke[1] = true$
$\underline{0}\underline{0}$	$poke[2] = true$
00	$poke[2] = poke[1] = false$

The same cycle will repeat indefinitely, because everything has returned to its initial state.

Notice that in this example, the repeating cycle consists of two distinct parts. The first half cycle, before *false* is returned, generates all two-bit patterns in Gray binary order $(00, 01, 11, 10)$; the other half generates those patterns again, but in the *reverse* order $(10, 11, 01, 00)$. This behavior will be characteristic of all the coroutines that we shall consider for the spider-squishing problem: Their task will be to run through all n-tuples $a_1 \dots a_n$ such that $a_j \le a_k$ for certain pairs (j, k), always returning *true* until all possible patterns have been generated; then they are supposed to run through those n-tuples again in reverse order, and to repeat the process ad infinitum.

In general, we can see without difficulty that the coroutines *poke*[1], *poke*[2], ..., *poke*[n] solve the problem of generating all n-tuples properly in the special case when no constraints $a_j \le a_k$ are present. Under our conventions, the following driver routine will cycle through the answers, printing a line of dashes between each complete listing:

```
while true do begin
   for k := 1 step 1 until n do write(a[k]);
   write(newline);
   if not poke[n] then write("-----", newline);
   end.
```

(In practice, of course, the driver would normally carry out some interesting process on the bits $a_1 \dots a_n$, instead of merely outputting them to a file.)

2. Chains. Now let's go to the opposite extreme and suppose that the digraph of constraints is an oriented path or chain,

$$1 \to 2 \to \cdots \to n.$$

In other words, we want now to generate all n-tuples $a_1 a_2 \dots a_n$ such that

$$0 \le a_1 \le a_2 \le \cdots \le a_n \le 1,$$

proceeding alternately forward and backward in Gray order. Of course this problem is trivial, but we want to do it with coroutines so that we'll be able to tackle more difficult problems later.

Here are some coroutines that do the new job.

```
        Boolean coroutine bump[k];
          while true do begin
awake0:   while k < n ∧ bump[k + 1] do return true;
          a[k] := 1;  return true;
asleep1:  return false;
awake1:   a[k] := 0;  return true;
asleep0:  while k < n ∧ bump[k + 1] do return true;
          return false;
          end.
```

In this case, the driver program initiates action by involving *bump*[1]. For example, the process plays out as follows when $n = 3$:

000	Initial state	123
$00\underline{1}$	$\mathit{bump}[1] = \mathit{bump}[2] = \mathit{bump}[3] = \mathit{true}$	$12\underline{3}$
$0\underline{1}1$	$\mathit{bump}[1] = \mathit{bump}[2] = \mathit{true},\ \mathit{bump}[3] = \mathit{false}$	$1\underline{2}$
$\underline{1}11$	$\mathit{bump}[1] = \mathit{true},\ \mathit{bump}[2] = \mathit{false}$	$\underline{1}$
111	$\mathit{bump}[1] = \mathit{false}$	1
$\underline{0}11$	$\mathit{bump}[1] = \mathit{true}$	$\underline{1}2$
$\underline{0}\underline{0}1$	$\mathit{bump}[1] = \mathit{bump}[2] = \mathit{true}$	$\underline{1}\underline{2}3$
$\underline{0}\underline{0}\underline{0}$	$\mathit{bump}[1] = \mathit{bump}[2] = \mathit{bump}[3] = \mathit{true}$	$\underline{1}\underline{2}\underline{3}$
000	$\mathit{bump}[1] = \mathit{bump}[2] = \mathit{bump}[3] = \mathit{false}$	123

Each troll's action now depends on whether his lamp is lit as well as on his state of wakefulness. A troll with an unlighted lamp always passes each bump to the right, without taking any notice unless a *false* reply comes back. In the latter case, he acts as if his lamp had been lit — namely, he either returns *false* (if just awakened), or he changes the lamp, returns *true*, and nods off.

(*Note:* The numbers '123', '$12\underline{3}$', ... at the right of this example correspond to an encoding that will be explained in Section 8 below. A similar column of somewhat inscrutable figures will be given with other examples we will see later, so that the principles of Section 8 will be easier to understand when we reach that part of the story. There is no need to decipher such notations until then; all will be revealed eventually.)

The dual situation, in which all inequalities are reversed so that we generate all $a_1a_2\ldots a_n$ with

$$1 \ge a_1 \ge a_2 \ge \cdots \ge a_n \ge 0,$$

is obtained by interchanging the roles of 0 and 1:

```
        Boolean coroutine cobump[k];
          while true do begin
awake0:   a[k] := 1;  return true;
asleep1:  while k < n ∧ cobump[k+1] do return true;
          return false;
awake1:   while k < n ∧ cobump[k+1] do return true;
          a[k] := 0;  return true;
asleep0:  return false;
          end.
```

A mixed situation in which the constraints are

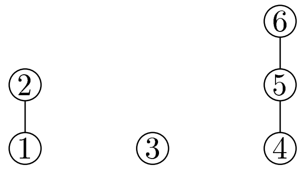
$$0 \le a_n \le a_{n-1} \le \cdots \le a_{m+1} \le a_1 \le a_2 \le \cdots \le a_m \le 1$$

is also worthy of note. Again the underlying digraph is a chain and the driver repeatedly bumps troll T_1, but when $1 < m < n$, the coroutines are a mixture of those we've just seen:

Boolean coroutine $mbump[1]$;
while $true$ **do begin**
awake0: **while** $1 \le k \le m \wedge mbump[k+1]$ **do return** $true$;
$a[k] := 1$; **return** $true$;
asleep1: **while** $(k = 1 \wedge mbump[m{+}1]) \vee (m \le k \le n \wedge mbump[k{+}1])$ **do return** $true$;
return $false$;
awake1: **while** $(k = 1 \wedge mbump[m{+}1]) \vee (m \le k \le n \wedge mbump[k{+}1])$ **do return** $true$;
$a[k] := 0$; **return** $true$;
asleep0: **while** $1 \le k \le m \wedge mbump[k+1]$ **do return** $true$;
return $false$;
end.

When $m \approx \frac{1}{2}n$, signals need to propagate only half as far as they do when $m = 1$.

Still another simple but significant variant arises when several separate chains are present. The digraph might, for example, be


,

in which case we want all 6-tuples of bits $a_1 \ldots a_6$ with $a_1 \le a_2$ and $a_4 \le a_5 \le a_6$. In general, suppose there is a set of endpoints $E = \{e_1, \ldots, e_m\}$ such that

$$1 = e_1 < \cdots < e_m \le n,$$

and we want

$$a_k \in \{0,1\} \quad \text{for } 1 \le k \le n; \qquad a_{k-1} \le a_k \quad \text{for } k \notin E.$$

(The set E is $\{1,3,4\}$ in the example shown.) The following coroutines $ebump[k]$, for $1 \le k \le n$, generate all such n-tuples if the driver invokes $ebump[e_m]$:

Boolean coroutine $ebump[k]$;
while $true$ **do begin**
awake0: **while** $k+1 \notin E \cup \{n+1\} \ \wedge \ ebump[k+1]$ **do return** $true$;
$a[k] := 1$; **return** $true$;
asleep1: **if** $k \in E \setminus \{1\}$ **return** $ebump[k']$ **else return** $false$;
awake1: $a[k] := 1$; **return** $true$;
asleep0: **while** $k+1 \notin E \cup \{n+1\} \ \wedge \ ebump[k+1]$ **do return** $true$;
if $k \in E \setminus \{1\}$ **return** $ebump[k']$ **else return** $false$;
end.

Here k' stands for e_{j-1} when $k = e_j$ and $j > 1$. These routines reduce to *poke* when $E = \{1, 2, \ldots, n\}$ and to *bump* when $E = \{1\}$. If $E = \{1, 3, 4\}$, they will generate all 24 bit patterns such that $a_1 \le a_2$ and $a_4 \le a_5 \le a_6$ in the order

$$\begin{array}{l}
000000,\ 00000\underline{1},\ 0000\underline{1}1,\ 000\underline{1}11,\ 00\underline{1}111,\ 00\underline{1}\underline{0}11,\ 00\underline{1}\underline{0}\underline{0}1,\ 00\underline{1}\underline{0}\underline{0}\underline{0},\\
0\underline{1}1000,\ 0\underline{1}100\underline{1},\ 0\underline{1}10\underline{1}1,\ 0\underline{1}1\underline{1}11,\ 0\underline{1}\underline{0}111,\ 0\underline{1}\underline{0}\underline{0}11,\ 0\underline{1}\underline{0}\underline{0}\underline{0}1,\ 0\underline{1}\underline{0}\underline{0}\underline{0}\underline{0},\\
\underline{1}10000,\ \underline{1}1000\underline{1},\ \underline{1}100\underline{1}1,\ \underline{1}10\underline{1}11,\ \underline{1}1\underline{1}111,\ \underline{1}1\underline{1}\underline{0}11,\ \underline{1}1\underline{1}\underline{0}\underline{0}1,\ \underline{1}1\underline{1}\underline{0}\underline{0}\underline{0};
\end{array}$$

then the sequence will reverse itself:

$$\begin{array}{l}
111000,\ 11100\underline{1},\ 1110\underline{1}1,\ 111\underline{1}11,\ 11\underline{0}111,\ 11\underline{0}\underline{0}11,\ 11\underline{0}\underline{0}\underline{0}1,\ 11\underline{0}\underline{0}\underline{0}\underline{0},\\
\underline{0}10000,\ \underline{0}1000\underline{1},\ \underline{0}100\underline{1}1,\ \underline{0}10\underline{1}11,\ \underline{0}1\underline{1}111,\ \underline{0}1\underline{1}\underline{0}11,\ \ \underline{0}1\underline{1}\underline{0}\underline{0}1\ \ \underline{0}1\underline{1}\underline{0}\underline{0}\underline{0},\\
\underline{0}\underline{0}1000,\ \underline{0}\underline{0}100\underline{1},\ \underline{0}\underline{0}10\underline{1}1,\ \underline{0}\underline{0}1\underline{1}11,\ \underline{0}\underline{0}\underline{0}111,\ \underline{0}\underline{0}\underline{0}\underline{0}11,\ \underline{0}\underline{0}\underline{0}\underline{0}\underline{0}1,\ \underline{0}\underline{0}\underline{0}\underline{0}\underline{0}\underline{0}.
\end{array}$$

In our examples so far we have discussed several families of cooperating coroutines and claimed that they generate certain n-tuples, but we haven't proved anything rigorously. A formal theory of coroutine semantics is beyond the scope of this paper, but we should at least try to construct a semi-formal demonstration that *ebump* is correct.

The proof is by induction on $|E|$, the number of chains. If $|E| = 1$, *ebump*$[k]$ reduces to *bump*$[k]$, and we can argue by induction on n. The result is obvious when $n = 1$. If $n > 1$, suppose repeated calls on *bump*$[2]$ cause $a_2 \ldots a_n$ to run through the $(n-1)$-tuples α_0, α_1, α_2, $\ldots$, where *bump*$[2]$ is *false* when it produces $\alpha_t = \alpha_{t-1}$. Such a repetition will occur if and only if t is a multiple of n, the number of distinct $(n-1)$-tuples with $a_2 \le \cdots \le a_n$. We know by induction that the sequence has reflective symmetry: $\alpha_j = \alpha_{2n-1-j}$ for $0 \le j \le n$. Furthermore, $\alpha_{j+2n} = \alpha_j$ for all $j \ge 0$. To complete the proof we observe that repeated calls on *bump*$[1]$ will produce the n-tuples

$$\begin{array}{l}
0\alpha_0,\ 0\alpha_1,\ \ldots,\ 0\alpha_{n-1},\ \underline{1}\alpha_n,\\
1\alpha_n,\ \underline{0}\alpha_n,\ \underline{0}\alpha_{n+1},\ \ldots,\ \underline{0}\alpha_{2n-1},\\
0\alpha_{2n},\ 0\alpha_{2n+1},\ \ldots,\ 0\alpha_{3n-1},\ \underline{1}\alpha_{3n},
\end{array}$$

and so on, returning *false* every $(n+1)^{\text{st}}$ step as desired.

If $|E| > 1$, let $E = \{e_1, \ldots, e_m\}$, so that $e'_m = e_{m-1}$, and suppose that repeated calls on *ebump*$[e_{m-1}]$ produce the $(e_m - 1)$-tuples $\alpha_0, \alpha_1, \alpha_2, \ldots$. Also suppose that calls on *ebump*$[e_m]$ would set the remaining bits $a_{e_m} \ldots a_n$ to the $(n + 1 - e_m)$-tuples β_0, β_1, β_2, $\ldots$, if E were empty instead of $\{e_1, \ldots, e_m\}$; this sequence β_0, β_1, β_2, $\ldots$ is like the output of *bump*. The α and β sequences are periodic, with respective periods of length $2M$ and $2N$ for some M and N; they also have reflective symmetry $\alpha_j = \alpha_{2M-1-j}$, $\beta_k = \beta_{2N-1-k}$. It follows that *ebump*$[e_m]$ is correct, because it produces the sequence

$$\begin{aligned}
\gamma_0, \gamma_1, \gamma_2, \ldots = {} & \alpha_0\beta_0,\ \alpha_0\beta_1,\ \ldots,\ \alpha_0\beta_{N-1},\\
& \alpha_1\beta_N,\ \alpha_1\beta_{N+1},\ \ldots,\ \alpha_1\beta_{2N-1},\\
& \quad \vdots\\
& \alpha_{M-1}\beta_{(M-1)N},\ \alpha_{M-1}\beta_{(M-1)N+1},\ \ldots,\ \alpha_{M-1}\beta_{MN-1},\\
& \alpha_M\beta_{MN},\ \alpha_M\beta_{MN+1},\ \ldots,\ \alpha_M\beta_{(M+1)N-1},\\
& \quad \vdots\\
& \alpha_{2M-1}\beta_{(2M-1)N},\ \alpha_{2M-1}\beta_{(2M-1)N+1},\ \ldots,\ \alpha_{2M-1}\beta_{2MN-1},\ \ldots
\end{aligned}$$

which has period length $2MN$ and satisfies

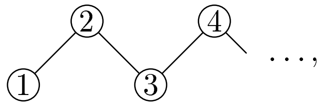
$$\gamma_{Nj+k} = \alpha_j \beta_{Nj+k} = \alpha_{2M-1-j} \beta_{2MN-1-Nj-k} = \gamma_{2MN-1-Nj-k}$$

for $0 \le j < M$ and $0 \le k < N$.

The patterns output by *ebump* are therefore easily seen to be essentially the same as the so-called *reflected Gray paths* for radices $e_2 + 1 - e_1, \ldots, e_m + 1 - e_{m-1}, n + 2 - e_m$ (see [4]); the total number of outputs is

$$(e_2 + 1 - e_1) \ldots (e_m + 1 - e_{m-1})(n + 2 - e_m).$$

3. Ups and downs. Now let's consider a "fence" digraph



which leads to n-tuples that satisfy

$$a_1 \le a_2 \ge a_3 \le a_4 \ge \cdots.$$

A reasonably simple set of coroutines can be shown to handle this case:

Boolean coroutine *nudge*$[k]$;
  while *true* **do begin**
awake0: **while** $k' \le n \ \wedge \ nudge[k']$ **do return** *true*;
  $a[k] := 1$; **return** *true*;
asleep1: **while** $k'' \le n \ \wedge \ nudge[k'']$ **do return** *true*;
  return *false*;
awake1: **while** $k'' \le n \ \wedge \ nudge[k'']$ **do return** *true*;
  $a[k] := 0$; **return** *true*;
asleep0: **while** $k' \le n \ \wedge \ nudge[k']$ **do return** *true*;
  return *false*;
  end.

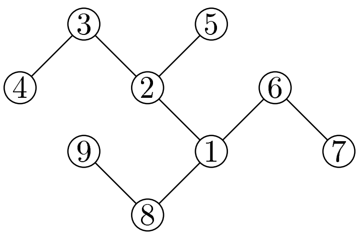
Here $(k', k'') = (k{+}1, k{+}2)$ when k is odd, $(k{+}2, k{+}1)$ when k is even. But these coroutines do *not* work when they all begin at 'awake0' with $a_1 a_2 \ldots a_n = 00 \ldots 0$; they need to be initialized carefully. For example, when $n = 6$ it turns out that exactly 11 patterns of odd weight need to be generated, and exactly 10 patterns of even weight, so a Gray path cannot begin or end with 000000 or 111111. In the proper starting configuration, $a_1 \ldots a_n$ will be set to the first n bits of $000111000111\ldots$, and coroutine *nudge*$[k]$ will begin at 'awake0' if $a_k = 0$, at 'awake1' if $a_k = 1$. For example, the sequence of results when $n = 4$

is

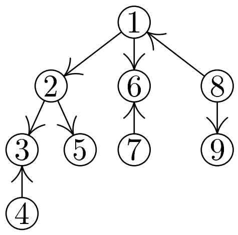
0001	Initial configuration	124
$000\underline{0}$	$nudge[1] = nudge[2] = nudge[4] = true$	$12\underline{4}$
$0\underline{1}00$	$nudge[1] = nudge[2] = true,\ nudge[4] = false$	$1\underline{2}34$
$0\underline{1}0\underline{1}$	$nudge[1] = nudge[2] = nudge[3] = nudge[4] = true$	$1\underline{2}3\underline{4}$
$0\underline{1}\underline{1}1$	$nudge[1] = nudge[2] = nudge[3] = true,\ nudge[4] = false$	$1\underline{2}\underline{3}$
$\underline{1}111$	$nudge[1] = true,\ nudge[2] = nudge[3] = false$	$\underline{1}3$
$\underline{1}1\underline{0}1$	$nudge[1] = nudge[3] = true$	$\underline{1}\underline{3}4$
$\underline{1}1\underline{0}\underline{0}$	$nudge[1] = nudge[3] = nudge[4] = true$	$\underline{1}\underline{3}\underline{4}$
1100	$nudge[1] = nudge[3] = nudge[4] = false$	134
$110\underline{1}$	$nudge[1] = nudge[3] = nudge[4] = true$	$13\underline{4}$
$11\underline{1}1$	$nudge[1] = nudge[3] = true,\ nudge[4] = false$	$1\underline{3}$
$\underline{0}111$	$nudge[1] = true,\ nudge[3] = false$	$\underline{1}23$
$\underline{0}1\underline{0}1$	$nudge[1] = nudge[2] = nudge[3] = true$	$\underline{1}2\underline{3}4$
$\underline{0}1\underline{0}\underline{0}$	$nudge[1] = nudge[2] = nudge[3] = nudge[4] = true$	$\underline{1}2\underline{3}\underline{4}$
$\underline{0}\underline{0}00$	$nudge[1] = nudge[2] = true,\ nudge[3] = nudge[4] = false$	$\underline{1}\underline{2}4$
$\underline{0}\underline{0}0\underline{1}$	$nudge[1] = nudge[2] = nudge[4] = true$	$\underline{1}\underline{2}\underline{4}$
0001	$nudge[1] = nudge[2] = nudge[4] = false$	124

Again the cycle repeats with reflective symmetry; and again, some cryptic notations appear that will be explained in Section 8. The correctness of *nudge* will follow from results we shall prove later.

4. The general case. Now let's turn to the general case, where an arbitrary totally acyclic digraph is given. The spider



illustrates most of the complications that might face us, so we shall use it as a running example. In general we shall assume that the vertices have been numbered in *preorder*, when the digraph is considered to be a forest (ignoring the arc directions). This means (see [4, Section 2.3.2] that the smallest vertex in each component is the root of that component, and that all vertex numbers of a component are consecutive. Furthermore, the children of each node are immediately followed in the ordering by their descendants. The descendants of each node k form a subspider consisting of nodes k through scope(k), inclusive; we shall call this "spider k." For example, spider 2 consists of nodes $\{2, 3, 4, 5\}$, and scope$(2) = 5$. Our sample spider has been numbered in preorder because it can be drawn as a properly numbered tree with directed branches:



Many other vertex numberings are possible, because any vertex of the digraph could have been chosen to be the root, and because the resulting trees can often be embedded several ways into the plane by permuting the children of each family.

Assume for the moment that the digraph is connected; thus it is a tree with root 1. A nonroot vertex x is called *positive* if the path from 1 to x ends with an arc directed towards x, *negative* if that path ends with an arc directed away from x. Thus the example spider has positive vertices $\{2, 3, 5, 6, 9\}$ and negative vertices $\{4, 7, 8\}$.

Let us write $x \to^* y$ if there is a directed path from x to y in the digraph. Removing all vertices x such that $x \to^* 1$ disconnects the graph into a number of pieces having positive roots; in our example, the removal of $\{1, 8\}$ leaves three components rooted at $\{2, 6, 9\}$. We call these roots the *positive vertices near* 1, and we denote that set by U_1. Similarly, the *negative vertices near* 1 are obtained when we remove all vertices y such that $1 \to^* y$; the set of resulting roots, denoted by V_1, is $\{4, 7, 8\}$ in our example, because we remove $\{1, 2, 3, 5, 6\}$.

The relevant bit patterns $a_1 \ldots a_n$ for which $a_1 = 0$ are precisely those that we obtain if we set $a_j = 0$ whenever $j \to^* 1$ and then we supply bit patterns for each subspider rooted at a vertex of U_1. Similarly, the bit patterns for which $a_1 = 1$ are precisely those we obtain by setting $a_k = 1$ whenever $1 \to^* k$ and supplying patterns for each subspider rooted at a vertex of V_1. Thus if n_k denotes the number of bit patterns for spider k, the total number of suitable patterns $a_1 \ldots a_n$ is $\prod_{u \in U_1} n_u + \prod_{v \in V_1} n_v$.

The sets U_k and V_k of positive and negative vertices near k are defined in the same way for each spider k.

Every positive child of k appears in U_k, and every negative child appears in V_k. These are called the *principal* elements of U_k and V_k. Every nonprincipal member of U_k is a member of U_v for some unique principal vertex v of V_k. Similarly, every nonprincipal member of V_k is a member of V_u for some unique principal vertex u of U_k. For example, the principal members of U_1 are 2 and 6; the other member, 9, belongs to U_8, where 8 is a principal member of V_1.

We will prove that the bit patterns $a_1 \ldots a_n$ can always be arranged in a Gray path such that bit a_1 begins at 0 and ends at 1, changing exactly once. By induction, such paths exist for the n_u patterns in each spider u for $u \in U_1$. And we can combine such paths into a single path that passes through all of the $\prod_{u \in U_1} n_u$ ways to combine those patterns, using a reflected Gray code analogous to the output of *ebump* in Section 3 above. Thus, if we set $a_k = 0$ for all k such that $k \to^* 1$, we get a Gray path P_1 for all suitable patterns with $a_1 = 0$. Similarly we can construct a Gray path Q_1 for the $\prod_{v \in V_1} n_v$ suitable patterns with $a_1 = 1$. All we need to do is prove that it is possible to construct P_1 and Q_1

in such a way that the last pattern in P_1 differs from the first pattern of Q_1 only in bit a_1. Then $G_1 = (P_1, Q_1)$ will be a suitable Gray path that solves our problem.

For example, consider the subspiders for $U_1 = \{2, 6, 9\}$ in the example spider. An inductive construction shows that they have respectively $(n_2, n_6, n_9) = (8, 3, 2)$ patterns, with corresponding Gray paths

$$\begin{aligned} G_2 &= 0000, 0001, 0101, 0100, 0110, 0111, 1111, 1101; \\ G_6 &= 00, 10, 11; \\ G_9 &= 0, 1. \end{aligned}$$

We obtain 48 patterns P_1 by setting $a_1 = a_8 = 0$ and using G_2 for $a_2a_3a_4a_5$, G_6 for a_6a_7, and G_9 for a_9, taking care to end with $a_2 = a_6 = 1$. Similarly, the subspiders for $V_1 = \{4, 7, 8\}$ have $(n_4, n_7, n_8) = (2, 2, 3)$ patterns, and paths

$$\begin{aligned} G_4 &= 0, 1; \\ G_7 &= 0, 1; \\ G_8 &= 00, 01, 11. \end{aligned}$$

We obtain 12 patterns Q_1 by setting $a_1 = a_2 = a_3 = a_5 = a_6 = 1$ and using G_4 for a_4, G_7 for a_7, and G_8 for a_8a_9, taking care to begin with $a_8 = 0$. Combining these observations, we see that P_1 should end with 011011100, and Q_1 should begin with 111011100.

In general, the last element of P_k and the first element of Q_k can be determined as follows: For all children j of k, set $a_j \ldots a_{\mathrm{scope}(j)}$ to the last element of the previously computed Gray path G_j if j is positive, or to the first element of G_j if j is negative. Then set $a_k = 0$ in P_k, $a_k = 1$ in Q_k. It is easy to verify that these rules make $a_j = 0$ whenever $j \to^* k$, for all j such that $k < j \le \mathrm{scope}(k)$. A reflected Gray code based on the paths G_u for $u \in U_k$ can be used to construct P_k ending at the transition values, having $a_k = 0$, and Q_k can be constructed from those starting values based on the paths G_v for $v \in V_k$, having $a_k = 1$. Thus we obtain a Gray path $G_k = (P_k, Q_k)$.

We have therefore constructed a Gray path for spider 1, proving that the spider-squishing problem has a solution when the underlying digraph is connected. To complete the construction for the general case, we can artificially ensure that the graph is connected by introducing a new vertex 0, with arcs from 0 to the roots of the components. Then P_0 will be the desired Gray path, if we suppress bit a_0 (which is zero throughout P_0).

5. Implementation via coroutines. By constructing families of sets U_k and V_k and identifying principal vertices in those sets, we have shown the existence of a Gray path for any given spider-squishing problem. Now let's make the proof explicit by constructing a family of coroutines that will dynamically generate the successive patterns $a_1 \ldots a_n$, as in the examples worked out in Sections 1–3 above.

First let's review some basic facts about coroutines of the type we need. Consider the following coroutines X and Y:

```
Boolean coroutine X( );
   while true do begin
      while A( ) do return true;
      return false;
      while B( ) do return true
      return false;
      while C( ) do return true;
      return false;
      end;
Boolean coroutine Y( );
   while true do begin
      while X( ) do return true;
      return Z( );
      end.
```

Here X invokes three coroutines A, B, C; Y invokes X and an arbitrary coroutine $Z \neq X, Y$. Clearly Y carries out essentially the same actions as the slightly faster coroutine XZ we get from X by substituting $Z(\ )$ wherever X returns *false*:

```
Boolean coroutine XZ( );
   while true do begin
      while A( ) do return true;
      return Z( );
      while B( ) do return true;
      return Z( );
      while C( ) do return true;
      return Z( );
      end.
```

This substitution principle can be used whenever all **return** statements of X are either '**return** *true*' or '**return** *false*'. And we could cast XZ into this same mold, if desired, by writing '**if** $Z(\ )$ **then return** *true* **else return** *false*' in place of '**return** $Z(\ )$'.

Next, consider coroutines whose actions produce infinite sequences $\alpha_1, \alpha_2, \ldots$ of period length $2M$, where $(\alpha_M, \ldots, \alpha_{2M-1})$ is the reverse of $(\alpha_0, \ldots, \alpha_{M-1})$, and where the coroutine returns *false* after producing α_t if and only if t is a multiple of M. We proved at the end of Section 2 that a construction like

```
Boolean coroutine AtimesB( );
   while true do begin
      while B( ) do return true;
      return A( );
      end
```

yields a coroutine that produces such sequences of period length $2MN$ from coroutines $A(\ )$ and $B(\ )$ of period lengths $2M$ and $2N$.

The following somewhat analogous coroutine produces such sequences of period length $2(M+N)$:

> **Boolean coroutine** *AplusB*( );
>   **while** *true* **do begin**
>   **while** $A(\ )$ **do return** *true*;
> $a[1] := 1$; **return** *true*;
>   **while** $B(\ )$ **do return** *true*;
>   **return** *false*;
>   **while** $B(\ )$ **do return** *true*;
> $a[1] := 0$; **return** *true*;
>   **while** $A(\ )$ **do return** *true*;
>   **return** *false*;
>   **end.**

This construction assumes that $A(\ )$ and $B(\ )$ generate reflective periodic sequences α and β on bits $a_2 \ldots a_n$ where $\alpha_M = \beta_0$. The first half of *AplusB* produces

$$0\alpha_0,\ \ldots,\ 0\alpha_{M-1},\ 1\beta_0,\ \ldots,\ 1\beta_{N-1},$$

and returns *false* after forming $1\beta_N$ (which equals $1\beta_{N-1}$). The second half produces the n-tuples

$$1\beta_N,\ \ldots,\ 1\beta_{2N-1},\ 0\alpha_M,\ \ldots,\ 0\alpha_{2M-1},$$

which are the first $M+N$ outputs in reverse; then it returns *false*, after forming $0\alpha_{2M}$ (which equals $0\alpha_0$).

The coroutines that we need to implement spider squishing can be built up from variants of the primitive constructions for product and sum just mentioned. Consider the following coroutines $gen[1], \ldots, gen[n]$, each of which receives an integer parameter l whenever being invoked:

> **Boolean coroutine** $gen[k](l)$; **integer** l;
>   **while** *true* **do begin**
> awake0: **while** $U_k \neq \emptyset \ \wedge\ gen[\max U_k](k)$ **do return** *true*;
> $a[k] := 1$; **return** *true*;
> asleep1: **while** $V_k \neq \emptyset \ \wedge\ gen[\max V_k](k)$ **do return** *true*;
>   **if** $\mathrm{prev}(k) > l$ **then return** $gen[\mathrm{prev}(k)](l)$ **else return** *false*;
> awake1: **while** $V_k \neq \emptyset \ \wedge\ gen[\max V_k](k)$ **do return** *true*;
> $a[k] := 0$; **return** *true*;
> asleep0: **while** $U_k \neq \emptyset \ \wedge\ gen[\max V_k](k)$ **do return** *true*;
>   **if** $\mathrm{prev}(k) > l$ **then return** $gen[\mathrm{prev}(k)](l)$ **else return** *false*;
>   **end.**

Here $\max U_k$ denotes the largest element of U_k, and $\mathrm{prev}(k)$ is a function that we shall define momentarily. This function, like the sets U_k and V_k, is statically determined from the given totally acyclic digraph. Since U_k and V_k are often empty and since $\mathrm{prev}(k)$ is often zero, many of the individual coroutines $gen[k]$ can be simplified by removing statements like '**while** *false* **do return** *true*'.

The idea of 'prev' is that all elements of U_l can be listed as u, $\mathrm{prev}(u)$, $\mathrm{prev}\bigl(\mathrm{prev}(u)\bigr)$, ..., until reaching an element $\le l$, if we start with $u = \max U_l$. Similarly, all elements of V_l can be listed as v, $\mathrm{prev}(v)$, $\mathrm{prev}\bigl(\mathrm{prev}(v)\bigr)$, ..., while those elements exceed l, starting with $v = \max V_l$. The basic meaning of $gen[k]$ with parameter l is to run through all bit patterns for the spiders $u \le k$ in U_l, if k is positive, or for the spiders $v \le k$ in V_l, if k is negative.

The example spider of Section 4 will help clarify the situation. The following table shows the sets U_k, V_k, and a suitable function $\mathrm{prev}(k)$, together with some auxiliary functions by which $\mathrm{prev}(k)$ can be determined in general:

k	$\mathrm{scope}(k)$	U_K	V_K	$\mathrm{prev}(k)$	$\mathrm{ppro}(k)$	$\mathrm{npro}(k)$
1	9	$\{2,6,9\}$	$\{4,7,8\}$	0	1	0
2	5	$\{3,5\}$	$\{4\}$	0	2	0
3	4	$\emptyset$	$\{4\}$	0	3	0
4	4	$\emptyset$	$\emptyset$	0	3	4
5	5	$\emptyset$	$\emptyset$	3	5	0
6	7	$\emptyset$	$\{7\}$	2	6	0
7	7	$\emptyset$	$\emptyset$	4	6	7
8	9	$\{9\}$	$\emptyset$	7	1	8
9	9	$\emptyset$	$\emptyset$	6	9	8

If u is a positive vertex, not a root, let v_1 be the parent of u. Then if v_1 is negative, let v_2 be the parent of v_1, and continue in this manner until reaching a positive vertex v_t, the least positive ancestor of v_1. We call v_t the *positive progenitor* of v_1, denoted $\mathrm{ppro}(v_1)$. The main point of this construction is that $u \in U_k$ if and only if k is one of the vertices $\{v_1, v_2, \ldots, v_t\}$. Consequently

$$U_k \;=\; U_l \cap \{k, k+1, \ldots, \mathrm{scope}(k)\}$$

if l is the positive progenitor of k. Furthermore U_k and $U_{k'}$ are disjoint whenever k and k' are distinct positive vertices. Therefore we can define $\mathrm{prev}(u)$ for all positive nonroots u as the largest element less than u in the set $U_k \cup \{0\}$, where k is the positive progenitor of u's parent.

Every element also has a negative progenitor, if we regard the dummy vertex 0 as a negative vertex that is parent to all the roots of the digraph. Thus we define $\mathrm{prev}(v)$ for all negative v as the largest element less than v in the set $V_k \cup \{0\}$, where k is the negative progenitor of v's parent.

Notice that 9 is an element of both U_1 and U_8 in the example spider, so both $gen[9](1)$ and $gen[9](8)$ will be invoked at various times. The former will invoke $gen[6](1)$, which will invoke $gen[2](1)$; the latter, however, will merely flip bit a_9 on and off, because $\mathrm{prev}(9)$ is less than 8. There is only coroutine $gen[9]$; its parameter l is reassigned each time $gen[9]$ is invoked. (The two usages do not conflict, because $gen[9](1)$ is invoked only when $a_1 = 0$, in which case $a_8 = 0$ and $gen[8]$ cannot be active.) Similarly, $gen[4]$ can be invoked with $l = 1, 2$, or 3; but in this case there is no difference in behavior because $\mathrm{prev}(4) = 0$.

In order to see why $gen[k]$ works, let's consider first what would happen if its parameter l were ∞, so that the test 'prev$(k) > l$' would always be false. In such a case $gen[k]$ is simply the *AplusB* construction applied to $A(\ ) = gen[\max U_k](k)$ and $B(\ ) = gen[\max V_k](k)$. On the other hand when l is set to a number such that $k \in U_l$ or $k \in V_l$, the coroutine $gen[k]$ is essentially the *AtimesB* construction substituted into this *AplusB*, having the effect of multiplying with $gen[\mathrm{prev}(k)](l)$. Thus we see that '**while** $U_k \neq \emptyset \wedge gen[\max U_k](k)$ **do return** *true*' generates the sequence P_k described in Section 4, and '**while** $V_k \neq \emptyset \wedge gen[\max V_k](k)$ **do return** *true*' generates Q_k. It follows that $gen[k](\infty)$ generates the Gray path G_k. And we get the overall solution to our problem, path P_0, by invoking $gen[\max U_0](0)$.

Well, there is one hitch: Every time the *AplusB* construction is used, we must be sure that coroutines $A(\ )$ and $B(\ )$ have been set up so that the last pattern of $A(\ )$ equals the first pattern of $B(\ )$. We shall deal with that problem in Section 6.

In the simplest case, where the given digraph has no arcs whatsoever, we have $U_0 = \{1, \ldots, n\}$ and all other U's and V's are empty. Thus $\mathrm{prev}(k) = k - 1$ for $1 \le k \le n$, and $gen[k](0)$ reduces to the coroutine $poke[k]$ of Section 1.

If the given digraph is the chain $1 \to 2 \to \cdots \to n$, the nonempty U's and V's are $U_k = \{k+1\}$ for $0 \le k < n$. Thus $\mathrm{prev}(k) = 0$ for all k, and $gen[k](l)$ reduces to the coroutine $bump[k]$ of Section 2.

If the given digraph is the fence $1 \to 2 \leftarrow 3 \to 4 \leftarrow \cdots$, we have $U_k = \{k'\}$ and $V_k = \{k''\}$ for $1 \le k < n$, where $(k', k'') = (k+1, k+2)$ if k is odd, $(k+2, k+1)$ if k is even, except that $U_{n-1} = \emptyset$ if n is odd, $V_{n-1} = \emptyset$ if n is even. Also $U_0 = \{1\}$. Therefore $\mathrm{prev}(k) = 0$ for all k, and $gen[k](l)$ reduces to the coroutine $nudge[k]$ of Section 3.

6. Launching. Ever since 1968, Section 1.4.2 of *The Art of Computer Programming* [3] has contained the following remark: "Initialization of coroutines tends to be a little tricky, although not really difficult." Perhaps that statement needs to be amended, from the standpoint of the coroutines considered here. We need to decide at which label each coroutine $gen[k]$ should begin execution when it is first invoked: awake0, asleep1, awake1, or asleep0. And our discussion in Sections 3 and 4 shows that we also need to choose the initial setting of $a_1 \ldots a_n$ very carefully.

Let's consider the initialization of $a_1 \ldots a_n$ first. The reflected Gray path mechanism that we use to construct the paths P_k and Q_k, as explained in Section 4, complements some of the bits. If, for example, $U_k = \{u_1, u_2, \ldots, u_m\}$, where $u_1 < u_2 < \cdots < u_m$, path P_k will contain $n_{u_1} n_{u_2} \ldots n_{u_m}$ bit patterns, and the value of bit a_{u_i} at the end of P_k will equal the value it had at the beginning if and only if $n_{u_1} n_{u_2} \ldots n_{u_{i-1}}$ is even. The reason is that subpath G_{u_i} is traversed $n_{u_1} n_{u_2} \ldots n_{u_{i-1}}$ times, alternately forward and backward.

In general, let

$$\delta_{jk} = \prod_{\substack{u<j \\ u \in U_k}} n_u, \text{ if } j \in U_k; \qquad \delta_{jk} = \prod_{\substack{v<j \\ v \in V_k}} n_v, \text{ if } j \in V_k.$$

Let α_{jk} and ω_{jk} be the initial and final values of bit a_j in the Gray path G_k for spider k, and let τ_{jk} be the value of a_j at the transition point (the end of P_k and the beginning of Q_k). Then $\alpha_{kk} = 0$, $\omega_{kk} = 1$, and the construction in Section 4 defines the values of α_{ik}, τ_{ik}, and ω_{ik} for $k < i \le \text{scope}(k)$ as follows: Suppose i belongs to spider j, where j is a child of k.

- If j is positive, so that j is a principal element of U_k, we have $\tau_{ik} = \omega_{ij}$, since P_k ends with $a_j = 1$. Also $\alpha_{ik} = \omega_{ij}$ if δ_{jk} is even, $\alpha_{ik} = \alpha_{ij}$ if δ_{jk} is odd. If $k \to^* i$ we have $\omega_{ik} = 1$; otherwise i belongs to spider j', where j' is a nonprincipal element of V_k. In the latter case $\omega_{ik} = \alpha_{ij'}$ if $\omega_{j'j} + \delta_{j'k}$ is even, otherwise $\omega_{ik} = \omega_{ij'}$. (This follows because $\omega_{j'j} = \tau_{j'k}$ and $\omega_{j'k} = (\tau_{j'k} + \delta_{j'k}) \bmod 2$.)
- If j is negative, so that j is a principal element of V_k, we have $\tau_{ik} = \alpha_{ij}$, since Q_k begins with $a_j = 0$. Also $\omega_{ik} = \alpha_{ij}$ if δ_{jk} is even, $\omega_{ik} = \omega_{ij}$ if δ_{jk} is odd. If $i \to^* k$ we have $\alpha_{ik} = 0$; otherwise i belongs to spider j', where j' is a nonprincipal element of U_k. In the latter case $\alpha_{ik} = \alpha_{ij'}$ if $\alpha_{j'j} + \delta_{j'k}$ is even, otherwise $a_{ik} = \omega_{ij'}$.

For example, when the digraph is the spider of Section 4, these formulas yield

k	n_k	Initial bits α_{jk}	Transition bits τ_{jk}	Final bits ω_{jk}
9	2	$a_9 = 0$	$*$	1
8	3	$a_8a_9 = 00$	$*1$	11
7	2	$a_7 = 0$	$*$	1
6	3	$a_6a_7 = 00$	$*0$	11
5	2	$a_5 = 0$	$*$	1
4	2	$a_4 = 0$	$*$	1
3	3	$a_3a_4 = 00$	$*0$	11
2	8	$a_2a_3a_4a_5 = 0000$	$*111$	1101
1	60	$a_1a_2\ldots a_9 = 000001100$	$*11011100$	111111100

Suppose j is a negative child of k. If n_u is odd for all elements of U_k that are less than j, then $\delta_{ij} + \delta_{ik}$ is even for all $i \in U_j$, and it follows that $a_{ik} = \tau_{ij}$ for $j < i \le \text{scope}(j)$. On the other hand, if n_u is even for some $u \in U_k$ with $u < j$, then δ_{ik} is even for all $i \in U_j$, and we have $\alpha_{ik} = \alpha_{ij}$ for $j < i \le \text{scope}(j)$. This observation makes it possible to compute the initial bits $a_1 \ldots a_n$ in $O(n)$ steps (see [9]).

The special nature of vertex 0 suggests that we define $\delta_{j0} = 1$, because we use path P_0 but not Q_0. This convention makes each component of the digraph essentially independent. (Otherwise, for example, the initial setting of $a_1 \ldots a_n$ would be $01\ldots1$ in the trivial "*poke*" case when the digraph has no arcs.)

Once we know the initial bits, we start *gen*$[k]$ at label awake0 if $a_k = 0$, at label awake1 if $a_k = 1$.

7. Optimization. The coroutines *gen*$[1]$, …, *gen*$[n]$ solve the general spider-squishing problem, but they might not run very fast. For example, the *bump* routine in Section 2 takes an average of about $n/2$ steps to decide which bit should be changed. We would much prefer to use only a bounded amount of time per bit change, on the average, and this goal turns out to be achievable if we optimize the coroutine implementation.

Consider a simple coroutine of the form

> **Boolean coroutine** f;
>   **while** *true* **do begin**
> f1: **while** *ff* **do return** *true*;
>   **return** *fa*;
> f2: **while** *fg* **do return** *true*;
>   **return** *fb*;
>   **end.**

The "obvious" way to implement f, using ordinary Algol procedures, is to introduce a global variable *fpos* and to use an Algol **switch** (or an equivalent feature found in a modern descendant of that language):

> **integer** *fpos*; **comment** initialized to 1 or 2;
> **Boolean procedure** f;
>   **begin switch** *fswitch* := f1, f2; **go to** *fswitch*[*fpos*];
> f1: **if** *ff* **then return** *true*;
>   *fpos* := 2; **return** *fa*;
> f2: **if** *fg* **then return** *true*;
>   *fpos* := 1; **return** *fb*;
>   **end.**

But let us suppose that coroutine f is always invoked in the context '**while** f **do** p', and that coroutines *ff* and *fg* often return a *true* result. Then we could short-circuit f-related activities by invoking *ff* and *fg* directly and taking special action only if they return *false*. For example, we could maintain a stack of position indices and switch directly to the item at the top of the stack. Instead of '**while** f **do** p' we could say $stack[1] := (1 \text{ or } 2)$; $s := 1$; **while** *call* **do** p', where *call* is implemented as follows:

> **Boolean procedure** *call*;
>   **begin while** $s > 0$ **do begin**
>   **if** *innercall* **then return** *true*;
> $s := s - 1$;
>   **end**;
>   **return** *false*; **comment** this shouldn't happen;
>   **end**;
> **integer** s; **integer array** $stack[1:100]$;
> **Boolean procedure** *innercall*;
>   **begin switch** *cswitch* := f1, f2, f1x, f2x, ff1, fg1;
>   **go to** $cswitch\big[stack[s]\big]$;
> f1: $stack[s] := 3$; $s := s + 1$; $stack[s] := 5$; **go to** ff1;
> f1x: $stack[s] := 2$; **return** *fa*;
> f2: $stack[s] := 4$; $s := s + 1$; $stack[s] := 6$; **go to** fg1;
> f2x: $stack[s] := 1$; **return** *fb*;

ff1: ⟨ The body of procedure *ff* ⟩;
fg1: ⟨ The body of procedure *fg* ⟩;
end.

Then if *ff* in turn is a coroutine that begins with '**while** *fff* **do return** *true*', we could put new labels ff1x and fff1 into *cswitch* and use the same idea to implement *ff*.

8. The active list. The *gen* coroutines of Section 5 perform $O(n)$ operations per bit change, as they pass signals back and forth, because each coroutine carries out at most two lines of its program. This upper bound on the running time cannot be substantially improved, in general. For example, the *bump* coroutines of Section 2 typically need to interrogate about $\frac{1}{2}n$ trolls per step; and it can be shown that the *nudge* coroutines of Section 3 typically involve action by about cn trolls per step, where $c = (5+\sqrt{5})/10 \approx 0.724$. (See [3, exercise 1.2.8–12].)

Using techniques like those of Section 7, however, the *gen* coroutines can always be transformed into a procedure that performs only $O(1)$ operations per bit change, amortized over all the changes. A formal derivation of such a transformation is beyond the scope of the present paper, but we will be able to envision it by considering an informal description of the algorithm that results.

The key idea is the concept of an *active list*, which encapsulates a given stage of the computation. The active list is a sequence of nodes that are either awake or asleep. If j is a positive child of k, node j is in the active list if and only if $k = 0$ or $a_k = 0$; if j is a negative child of k, it is in the active list if and only if $a_k = 1$.

Examples of the active list in special cases have appeared in the tables illustrating *bump* in Section 2 and *nudge* in Section 3. Readers who wish to review those examples will find that the numbers listed there do indeed satisfy these criteria. Furthermore, a node number has been underlined when that node is asleep; bit a_j has been underlined if and only if j is asleep and in the active list.

Initially $a_1 \ldots a_n$ is set to its starting pattern as defined in Section 6, and all elements of the corresponding active list are awake. To get to the next bit pattern, we perform the following actions:

1) Let k be the largest nonsleeping node on the active list, and wake up all nodes that are larger. (If all elements of the active list are asleep, they all wake up and no bit change is made; this case corresponds to $gen[\max U_0](0)$ returning *false*.)
2) If $a_k = 0$, set a_k to 1, delete k's positive children from the active list, and insert k's negative children. Otherwise set a_k to 0, insert the positive children, and delete the negative ones. (Newly inserted nodes are awake.)
3) Put node k to sleep.

Again the reader will find that the *bump* and *nudge* examples adhere to this discipline.

If we maintain the active list in order of its nodes, the amortized cost of these three operations is $O(1)$, because we can charge the cost of inserting, deleting, and awakening node k to the time when bit a_k changes. Steps (1) and (2) might occasionally need to do a lot of work, but this argument proves that such difficult transitions must be rare.

Let's consider the spider of Section 4 one last time. The 60 bit patterns that satisfy its constraints are generated by starting with $a_1 \dots a_9 = 000001100$, as we observed in Section 6, and the Gray path G_1 begins as follows according to the active list protocol:

000001100	1235679
$00000110\underline{1}$	$123567\underline{9}$
$0000010\underline{0}01$	$12356\underline{7}9$
$0000010\underline{0}0\underline{0}$	$12356\underline{7}\underline{9}$
$000000\underline{0}000$	$12356\underline{6}9$
$000000\underline{0}00\underline{1}$	$1235\underline{6}\underline{9}$
$00001\underline{1}0001$	$123\underline{5}69$
$00001\underline{1}000\underline{0}$	$123\underline{5}6\underline{9}$
$0000\underline{1}\underline{1}000$	$123\underline{5}\underline{6}79$

(Notice how node 7 becomes temporarily inactive when a_6 becomes 0.) The most dramatic change will occur after the first $n_2 n_6 n_9 = 48$ patterns, when bit a_1 changes as we proceed from path P_1 to path Q_1:

$0\underline{1}1\underline{0}1\underline{1}\underline{1}0\underline{0}$	$1\underline{2}\underline{4}\underline{6}\underline{7}\underline{9}$
$\underline{1}11011100$	$\underline{1}4789$

Finally, after all 60 patterns have been generated, the active list will be $\underline{1}\underline{4}\underline{7}\underline{8}\underline{9}$ and $a_1 \dots a_9$ will be $\underline{1}11\underline{1}11\underline{1}\underline{0}\underline{0}$. All active nodes will be napping, but when we wake them up they will be ready to regenerate the 60 patterns in reverse order.

It should be clear from these examples, and from an examination of the *gen* coroutines, that steps (1), (2), and (3) faithfully implement those coroutines in an efficient iterative manner.

9. Additional optimizations. The algorithm of Section 8 can often be streamlined further. For example, if j and j' are consecutive positive children of k and if V_j is empty, then j and j' will be adjacent in the active list whenever they are inserted or deleted. We can therefore insert or delete an entire family en masse, in the special case that all nodes are positive, if the active list is doubly linked. This important special case was first considered by Koda and Ruskey [6]; see also [4, Algorithm 7.2.1.1K].

Further tricks can in fact be employed to make the active list algorithm entirely *loopless*, in the sense that $O(1)$ operations are performed between successive bit changes in *all* cases (not only in an average, amortized sense). One idea, used by Koda and Ruskey in the special case just mentioned, is to use "focus pointers" to identify the largest nonsleeping node (see [2] and [4, Algorithm 7.2.1.1L]). Another idea, which appears to be necessary when both positive and negative nodes appear in a complex family, is to perform lazy updates to the active list, changing links only gradually but before they are actually needed. Such a loopless implementation, which moreover needs only $O(n)$ steps to initialize all the data structures, is described fully in [9]. It does not necessarily run faster than a more straightforward amortized $O(1)$ algorithm, from the standpoint of total time on a sequential (not parallel) computer; but it does prove that a strong performance guarantee is achievable, given any totally acyclic digraph.

10. Conclusions and acknowledgements. We have seen that a systematic use of cooperating coroutines leads to a generalized Gray code for generating all bit patterns that satisfy the ordering constraints of any totally acyclic digraph. Furthermore these coroutines can be implemented efficiently, yielding an algorithm that is faster than previously known methods for that problem. Indeed, the algorithm is optimum, in the sense that its running time is linear in the number of outputs.

Further work is clearly suggested in the heretofore neglected area of coroutine transformation. For example, we have not discussed the implementation of coroutines such as

```
Boolean coroutine copoke[k];
  while true do begin
  while k < n ∧ copoke[k+1] do return true;
  a[k] := 1 − a[k]; return true;
  while k < n ∧ copoke[k+1] do return true;
  return false;
  end.
```

These coroutines, which are to be driven by repeatedly calling *copoke*[1], look superficially similar to *gen*, but they are not actually a special case of that construction. A rather large family of coroutine optimizations seems to be waiting to be discovered and to be treated formally.

Another important open problem is to discover a method that generates the bit patterns corresponding to an *arbitrary* acyclic digraph, with an amortized cost of only $O(1)$ per pattern. The best currently known bound is $O(\log n)$, due to M. B. Squire [10]; see also [8, Section 4.11.2]. There is always a listing of the relevant bit patterns in which at most two bits change from one pattern to the next [7, Corollary 1].

The first author thanks Ole-Johan Dahl for fruitful collaboration at the University of Oslo during 1972–1973 and at Stanford University during 1977–1978; also for sharing profound insights into the science of programming and for countless hours of delightful four-hands piano music. The second author thanks Malcolm Smith and Gang (Kenny) Li for their help in devising early versions of algorithms for spider-squishing during 1991 and 1995, respectively.

References

[1] Ole-Johan Dahl and Kristen Nygaard, "SIMULA—an ALGOL-based simulation language," *Communications of the ACM* **9** (1966), 671–678.

[2] Gideon Ehrlich, "Loopless algorithms for generating permutations, combinations and other combinatorial configurations," *Journal of the Association for Computing Machinery* **20** (1973), 500–513.

[3] Donald E. Knuth, *Fundamental Algorithms*, Volume 1 of *the Art of Computer Programming* (Reading, Massachusetts: Addison–Wesley, 1968). Third edition, 1997.

[4] Donald E. Knuth, "Generating all n-tuples," Section 7.2.1.1 of *The Art of Computer Programming*, Volume 4 (Addison–Wesley), in preparation. Preliminary excerpts of this material are available at `http://www-cs-faculty.stanford.edu/~knuth/news01.html`.

[5] Donald E. Knuth, "Selected Topics in Computer Science, Part II, *Lecture Note Series*, Number 2 (Blindern, Norway: University of Oslo, Institute of Mathematics, August 1973). See page 3 of the notes entitled "Generation of combinatorial patterns: Gray codes."

[6] Yasunori Koda and Frank Ruskey, "A Gray code for the ideals of a forest poset," *Journal of Algorithms* **15** (1993), 324–340.

[7] Gara Pruesse and Frank Ruskey, "Gray codes from antimatroids," *Order* **10** (1993), 239–252.

[8] Frank Ruskey, *Combinatorial Generation* [preliminary working draft]. Department of Computer Science, University of Victoria, Victoria B.C., Canada (1996).

[9] Donald E. Knuth, SPIDERS, a program downloadable from the website
`http://www-cs-faculty.stanford.edu/~knuth/programs.html`.

[10] Matthew Blaze Squire, *Gray Codes and Efficient Generation of Combinatorial Structures.* Ph.D. dissertation, North Carolina State University (1995), x + 145 pages.

[11] George Steiner, "An algorithm to generate the ideals of a partial order," *Operations Research Letters* **5** (1986), 317–320.