November 24, 2020 at 13:23

1. Introduction. The purpose of this program is to implement a pretty algorithm that has a very pleasant theory. But I apologize at the outset that the algorithm seems to be rather subtle, and I have not been able to think of any way to explain it to dummies. Readers who like discrete mathematics and computer science are encouraged to persevere nonetheless.

The companion program called KODA-RUSKEY should probably be read first, because it solves a significantly simpler problem. After I wrote that program, Frank Ruskey told me that he had developed a much more general procedure, in yet-unpublished work with his student Gang (Kenny) Li. I wasn't able to refrain from pondering what their method might be, so I came up with the algorithm below, which probably produces the same sequence of outputs as theirs.

```
#include <stdio.h>
  ⟨ Type definitions 6 ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Subroutines 15 ⟩
  int main(int argc, char *argv[])
  {
    ⟨ Local variables 5 ⟩;
    ⟨ Parse the command line 2 ⟩;
    ⟨ Initialize the data structures 7 ⟩;
    ⟨ Generate the answers 46 ⟩;
    return 0;
  }
```

2. Given a digraph that is *totally acyclic*, in the sense that it has no cycles when we ignore the arc directions, we want to find all ways to label its vertices with 0s and 1s in such a way that $x \to y$ implies $bit[x] \le bit[y]$. Moreover, we want to list all such labelings as a Gray path, changing only one bit at a time. The algorithm below does this, with an extra proviso: Given a designated "root" vertex v, $bit[v]$ begins at 0 and changes exactly once.

The simple three-vertex digraph with $x \to y \leftarrow z$ has only five such labelings, and they form a Gray path in essentially only one way, namely $(000, 010, 011, 111, 110)$. This example shows that we cannot require the Gray path to end at a convenient prespecified labeling like $11\ldots1$; and the dual graph, obtained by reversing all the arrows and complementing all the bits, shows that we can't require the path to start at $00\ldots0$. [Generalizations of this example, in which the vertices are $\{x_0, x_1, \ldots, x_n\}$ and each arc is either $x_{k-1} \to x_k$ or $x_{k-1} \leftarrow x_k$, have solutions related to continued fractions. Interested readers will enjoy working out the details.]

We may assume that the given graph is connected. It is convenient to describe a connected, rooted totally acyclic digraph by calling the root 0, and by assigning the integer names $\{1, \ldots, n\}$ to the other vertices in such a way that the undirected path connecting each vertex to 0 runs through vertex names monotonically. In other words, we specify for each vertex $k > 0$ a vertex $j_k < k$ such that either $j_k \to k$ or $j_k \leftarrow k$. Notice that there are $2^n\, n!$ ways to do this. The program below works on the graph defined by a sequence of the form $\pm j_1 \pm j_2 \ldots \pm j_n$, using a plus sign when $j_k \to k$ and a minus sign when $j_k \leftarrow k$. For example, the digraph $0 \to 1 \leftarrow 2$ would be defined by `+0-1`, while $1 \to 0 \leftarrow 2$ would be `-0-0`.

We may assume further that the vertices of the tree have been numbered in *preorder*, namely that we do not have $j_k < j_l < k < l$ for any k and l. This assumption limits the number of possibilities to $2^n \binom{2n}{n} \frac{1}{n+1}$.

Breaking the arc between j_k and k disconnects the graph into two pieces, one of which is a totally acyclic digraph rooted at k. Our algorithm will use the Gray paths obtained from these smaller subgraphs to construct a Gray path for the whole graph.

```
#define maxn 100 /* limit in number of vertices */
#define abort(f,d,n)
        { fprintf(stderr,f,d); exit(-n); }
⟨ Parse the command line 2 ⟩ ≡
  {
    register char *c = argv[1];
    if (argc ≠ 2) abort("Usage: %s graphspec\n", argv[0], 1);
    for (k = 1; *c; k++) {
      if (k ≥ maxn) abort("Sorry, I can only handle %d vertices!\n", maxn, 2);
      if (*c ≡ '+') js[k] = 0;
      else if (*c ≡ '-') js[k] = 1;
      else abort("Parsing error: '%s' should start with + or -!\n", c, 3);
      for (j = 0, c++; *c ≥ '0' ∧ *c ≤ '9'; c++) j = 10 * j + *c - '0';
      ⟨ Abort if j is not a legal value for j_k 3 ⟩;
      jj[k] = j;
    }
    n = k - 1;
  }
```

This code is used in section 1.

3. Here we test the preorder condition, using the fact that $stack[0] = 0$ and that l is initially 0.

```
⟨ Abort if j is not a legal value for j_k 3 ⟩ ≡
  if (j ≥ k) {
    fprintf(stderr, "Parsing␣error:␣'%d%s'␣should␣start␣", j, c);
    abort("with␣a␣number␣less␣than␣%d!\n", k, 4);
  }
  while (j < stack[l]) l−−;
  if (j ≠ stack[l]) {
    fprintf(stderr, "Parsing␣error:␣'%d%s'␣shouldn't␣start␣", j, c);
    fprintf(stderr, "with␣a␣number␣between␣%d␣", stack[l]);
    abort("and␣%d!\n", stack[l + 1], 5);
  }
  stack[++l] = k;
```

This code is used in section 2.

4.

```
⟨ Global variables 4 ⟩ ≡
  char jj[maxn];      /* the vertex j_k */
  char js[maxn];      /* 0 if j_k → k,  1 if j_k ← k */
  char stack[maxn];   /* values that are legitimate for the next j_k */
```

See also sections 8, 14, 24, 30, 32, 37, 47, 57, and 60.

This code is used in section 1.

5.

```
⟨ Local variables 5 ⟩ ≡
  register int j, k, l = 0;   /* heavily-used miscellaneous indices */
  int n;   /* size of the input graph */
```

See also sections 9 and 45.

This code is used in section 1.

6. Consider the example

in which all arcs are directed upward; it would be written `+0+1-2+1+0-5-0+7` in the notation above. A nonroot vertex k is called *positive* if $j_k \to k$ and *negative* if $j_k \leftarrow k$; thus $\{1, 2, 4, 5, 8\}$ are positive in this example, and $\{3, 6, 7\}$ are negative.

We write $x \preceq y$ if there is a directed path from x to y. Removing all vertices x such that $x \preceq 0$ disconnects the graph into a number of pieces having positive roots; in our example, removing $\{0, 7\}$ leaves three components rooted at $\{1, 5, 8\}$. We call these roots the set of *positive vertices near* 0, and denote that set by A. Similarly, the *negative vertices near* 0 are obtained when we remove all y such that $0 \preceq y$; the set of resulting roots, denoted by B, is $\{3, 6, 7\}$ in our example.

Why are the sets A and B so important? Because the labelings for which $bit[0] = 0$ are precisely those that we obtain by setting $bit[x] = 0$ for all $x \preceq 0$ and then labeling the subgraphs rooted at a for $a \in A$. Similarly, all labelings for which $bit[0] = 1$ are obtained by setting $bit[y] = 1$ for all $0 \preceq y$ and labeling the subgraphs rooted at b for $b \in B$.

Thus if n_k is the number of labelings of the subgraph rooted at k, the total number of labelings of the whole graph is $\prod_{a\in A} n_a + \prod_{b\in B} n_b$.

For each subgraph rooted at k, we define the positive and negative vertices near k in the same way as we did for the case $k = 0$, calling those sets A_k and B_k.

Every positive vertex adjacent to 0 appears in A, and every negative vertex adjacent to 0 appears in B. These are called the *principal* elements of A and B. Every nonprincipal member of A is a member of A_b for some unique principal vertex $b \in B$. Similarly, every nonprincipal member of B is a member of B_a for some unique principal vertex $a \in A$. For example, 8 belongs to A_7, and $3 \in B_1$.

```
⟨ Type definitions 6 ⟩ ≡
  typedef struct info_struct {
    char id;      /* name of vertex in an A or B set */
    ⟨ Other fields of an info record 12 ⟩;
    struct info_struct *sib;      /* previous element in the set */
    struct info_struct *ref;      /* further info about nonprincipal element */
  } info;
```

See also sections 29 and 55.

This code is used in section 1.

7. ⟨ Initialize the data structures 7 ⟩ ≡

```
  list[0][0] = list[0][1] = Λ;
  for (k = 1, i = &infonode[0]; k ≤ n; k++, i++)
    for (j = jj[k], ii = Λ; ; ii = i++, j = jj[j]) {
      i→id = k, i→ref = ii;
      i→sib = list[j][js[k]], list[j][js[k]] = i;
      if (j ≡ 0 ∨ js[j] ≡ js[k]) break;
    }
```

See also sections 13, 16, 23, 31, and 61.

This code is used in section 1.

8. ⟨ Global variables 4 ⟩ +≡

```
  info infonode[((maxn * maxn) ≫ 2) + (maxn ≫ 1)];      /* elements of A_k and B_k */
  info *list[maxn][2];      /* heads of those sets */
```

9. ⟨ Local variables 5 ⟩ +≡

```
  info *i, *ii;      /* pointers to info nodes */
```

10. Aha! We can begin to see how to get the desired Gray path. The well-known reflected Gray code for mixed-radix number systems tells us how to obtain a path P_0 of length $\prod_{a\in A} n_a$ for the labelings with $bit[0] = 0$ as well as a path P_1 of length $\prod_{b\in B} n_b$ for the labelings with $bit[0] = 1$. All we have to do is figure out a way to end P_0 with a labeling that differs only in $bit[0]$ from the starting point of P_1.

Suppose P_0 begins with $bit[a] = \alpha_a$ for each $a \in A$, and P_1 ends with $bit[b] = \beta_b$ for each $b \in B$. Then P_0 ends with $bit[a] = \alpha'_a = (\alpha_a + \delta_a) \bmod 2$ and P_1 begins with $bit[b] = \beta'_b = (\beta_b + \epsilon_b) \bmod 2$, where

$$\delta_a = \prod_{\substack{a'<a\\ a'\in A}} n_{a'}, \qquad \epsilon_b = \prod_{\substack{b'<b\\ b'\in B}} n_{b'}.$$

These equations have a unique solution such that the final labeling of P_0 is adjacent to the first labeling of P_1. Namely, we set $\alpha'_a = 1$ for all principal elements $a \in A$, and $\beta'_b = 0$ for all principal elements $b \in B$. And if a is nonprincipal but $a \in A_b$, where b is principal, we let $\alpha'_a = \alpha_{ba}$, the value of $bit[a]$ at the beginning of the Gray path for the subgraph rooted at b. Similarly, if b is nonprincipal but $b \in B_a$, where a is principal, we let $\beta'_b = \beta_{ab}$. These formulas and those of the previous paragraph determine the values of α_a and β_b for all $a \in A$ and $b \in B$.

For example, the calculations yield the following numbers when we apply them to the smaller subgraphs, from the bottom up:

$A_8 = \emptyset$		$B_8 = \emptyset$		$n_8 = 1 + 1 = 2$
$A_7 = \{8\}$	$\alpha'_{78} = 1;\ \alpha_{78} = 0$	$B_7 = \emptyset$		$n_7 = 2 + 1 = 3$
$A_6 = \emptyset$		$B_6 = \emptyset$		$n_6 = 1 + 1 = 2$
$A_5 = \emptyset$		$B_5 = \{6\}$	$\beta'_{56} = 0;\ \beta_{56} = 1$	$n_5 = 1 + 2 = 3$
$A_4 = \emptyset$		$B_4 = \emptyset$		$n_4 = 1 + 1 = 2$
$A_3 = \emptyset$		$B_3 = \emptyset$		$n_3 = 1 + 1 = 2$
$A_2 = \emptyset$		$B_2 = \{3\}$	$\beta'_{23} = 0;\ \beta_{23} = 1$	$n_2 = 1 + 2 = 3$
$A_1 = \{2, 4\}$	$\alpha'_{12} = 1, \alpha'_{14} = 1;\ \alpha_{12} = 0, \alpha_{14} = 0$	$B_1 = \{3\}$	$\beta'_{13} = 1;\ \beta_{13} = 0$	$n_1 = 6 + 2 = 8$

For the graph as a whole, therefore, which has $A_0 = \{1, 5, 8\}$ and $B_0 = \{3, 6, 7\}$, we have $\alpha'_1 = 1$, $\alpha'_5 = 1$, $\alpha'_8 = \alpha_{78} = 0$; $\alpha_1 = 0$, $\alpha_5 = 1$, $\alpha_8 = 0$; $\beta'_3 = \beta_{13} = 0$, $\beta'_6 = \beta_{56} = 1$, $\beta'_7 = 0$; $\beta_3 = 1$, $\beta_6 = 1$, $\beta_7 = 0$. There are $n_0 = 48 + 12 = 60$ possible labelings altogether.

The Gray path for a trivial subgraph like G_8, G_6, G_4, or G_3 is simply '0, 1'. For G_7 on bits 7 and 8, the path is '00, 01, 11'. For G_5 on bits 5 and 6, or G_2 on bits 2 and 3, the path is '00, 10, 11'. And for G_1 on bits 1234, the path of length 8 is

$$0000,\ 0001,\ 0101,\ 0100,\ 0110,\ 0111,\ 1111,\ 1101.$$

(Notice that it ends with $bit[3] = \beta_{13} = 0$.)

11. The overall Gray path for our example starts out with $bit[1] = \alpha_1 = 0$, $bit[5] = \alpha_5 = 1$, and $bit[8] = \alpha_8 = 0$, so the first $n_1 n_5 n_8 = 48$ labelings $bit[0] \ldots bit[8]$ are

$$\begin{array}{l} 0\,0000\,11\,0\,0 \\ 0\,0000\,11\,0\,1 \\ 0\,0000\,10\,0\,1 \\ 0\,0000\,10\,0\,0 \\ 0\,0000\,00\,0\,0 \\ 0\,0000\,00\,0\,1 \\ 0\,0001\,00\,0\,1 \\ 0\,0001\,00\,0\,0 \\ \quad\vdots \\ 0\,1101\,11\,0\,0\,. \end{array}$$

Then comes the change to $bit[0]$, and we finish up with $n_3 n_6 n_7 = 12$ more:

$$\begin{array}{l} 1\,11\,0\,11\,1\,00 \\ 1\,11\,0\,11\,1\,01 \\ 1\,11\,0\,11\,1\,11 \\ 1\,11\,0\,11\,0\,11 \\ 1\,11\,0\,11\,0\,01 \\ 1\,11\,0\,11\,0\,00 \\ 1\,11\,1\,11\,0\,00 \\ \quad\vdots \\ 1\,11\,1\,11\,1\,00\,. \end{array}$$

12. ⟨ Other fields of an **info** record 12 ⟩ ≡

```
char alfprime;      /* α'_ka or β'_kb */
char del;           /* δ_ka mod 2 or ε_kb mod 2 */
```

This code is used in section 6.

13. We need not actually calculate the values n_k exactly; we only need to know if n_k is even or odd. However, this program does compute the exact values (unless they exceed our computer's word size).

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = n; k ≥ 0; k−−) {
    register int s, t;
    for (s = 1, i = list[k][0], ii = Λ; i; s *= nn[i→id], i = i→sib) {
      i→del = 1;
      if ((nn[i→id] & 1) ≡ 0) ii = i;
      if (i→ref) i→alfprime = i→ref→alfprime ⊕ i→ref→del;
      else i→alfprime = 1;
    }
    for (i = list[k][0]; ii; i = i→sib)
      if (i ≡ ii) ii = Λ; else i→del = 0;
    for (t = 1, i = list[k][1], ii = Λ; i; t *= nn[i→id], i = i→sib) {
      i→del = 1;
      if ((nn[i→id] & 1) ≡ 0) ii = i;
      if (i→ref) i→alfprime = i→ref→alfprime ⊕ i→ref→del;
      else i→alfprime = 0;
    }
    for (i = list[k][1]; ii; i = i→sib)
      if (i ≡ ii) ii = Λ; else i→del = 0;
    nn[k] = s + t;
  }
```

14.

```
⟨ Global variables 4 ⟩ +≡
  int nn[maxn];      /* n_k, the number of labelings of G_k */
```

15. Here are two subroutines that I used when debugging.

```
⟨ Subroutines 15 ⟩ ≡
  void print_info(int k)
  {
    register info *i;
    printf("Info␣for␣vertex␣%d:␣A␣=", k);
    for (i = list[k][0]; i; i = i→sib) printf("␣%d", i→id);
    if (list[k][0]) printf(",␣B␣="); else printf("␣(),␣B␣=");
    for (i = list[k][1]; i; i = i→sib) printf("␣%d", i→id);
    if (list[k][1]) printf("\n"); else printf("␣()\n");
    for (i = list[k][0]; i; i = i→sib)
      printf("␣alf%d=%d,␣alf%d'=%d\n", i→id, i→alfprime ⊕ i→del, i→id, i→alfprime);
    for (i = list[k][1]; i; i = i→sib)
      printf("␣bet%d=%d,␣bet%d'=%d\n", i→id, i→alfprime ⊕ i→del, i→id, i→alfprime);
  }
  void print_all_info(int n)
  {
    register int k;
    for (k = 0; k ≤ n; k++) print_info(k);
  }
```

See also sections 58 and 59.

This code is used in section 1.

16. ⟨ Initialize the data structures 7 ⟩ +≡

```
if (verbose) {
  print_all_info(n);
  printf("(Altogether␣%d␣solutions.)\n", nn[0]);
}
```

17. Coroutines. As in the program KODA-RUSKEY, we can represent the path-generation process by considering a system of cooperating programs, each of which has the same basic structure. There is one coroutine for each pair of nodes (k,l) such that $k \in A_l$ or $k \in B_l$, and it looks essentially like this:

```
coroutine p( )
  {
    while (1) {
       while (p→child_A( )) return true;
      p→bit = 1; return true;
       while (p→child_B( )) return true;
      return p→sib( );
       while (p→child_B( )) return true;
      p→bit = 0; return true;
       while (p→child_A( )) return true;
      return p→sib( );
    }
  }
```

Here p→*child_A* is a pointer to the rightmost element of A_k, and p→*child_B* points to the rightmost element of B_k; p→*sib* points to the nearest sibling of p to the left, in A_l or B_l. If any of these pointers is Λ, the corresponding coroutine $\Lambda(\,)$ is assumed to simply return *false*.

The reader is urged to compare this coroutine with the analogous one in KODA-RUSKEY, which is the special case where p→*child_A* $= \Lambda$ for all p.

Suppose p→*child_A*( ) first returns *false* after it has been called α times; thus p→*child_A*( ) generates α different labelings, including the initial one. Similarly, suppose that p→*child_B*( ) and p→*sib*( ) generate β and σ different labelings, respectively, before first returning *false*. Then the coroutine $p(\,)$ itself will generate $\sigma(\alpha+\beta)$ labelings between the times when it returns *false*. The final labeling for p will be the final labeling for p→*sib*, together with either the initial or final labeling of the subtree rooted at k, depending on whether σ is even or odd, respectively.

After the coroutine has first returned *false*, invoking it again will cause it to generate the labelings in reverse order before returning *false* a second time. Then the process repeats.

18. How many coroutines can there be? Our example graph defines a total of 13 coroutines (including a coroutine 0, which corresponds to the special case where $k = 0$ and there are no siblings).

It isn't difficult to prove that the worst case, about $n^2/4$, occurs in graphs that have a specification like `+0+1+2+3+4-5-5-5-5-5`. Such graphs have roughly n times $2^{n/2}$ labelings, so we do not have to be embarrassed about the nonlinear initialization time needed to set up data structures and to compute the values α_{ka}, α'_{ka}, β_{kb}, and β'_{kb}.

Indeed, if k is a positive vertex, it appears in r coroutines (k,l_1), $\ldots$, (k,l_r) if and only if the path from k to 0 has the form $k \leftarrow l_1 \rightarrow \cdots \rightarrow l_r = 0$ or begins with $k \leftarrow l_1 \rightarrow \cdots \rightarrow l_r \leftarrow l_{r+1}$. If k is a negative vertex, it appears in r coroutines (k,l_1), $\ldots$, (k,l_r) if and only if the path from k to 0 has the form $k \rightarrow l_1 \leftarrow \cdots \leftarrow l_r = 0$ or begins with $k \rightarrow l_1 \leftarrow \cdots \leftarrow l_r \rightarrow l_{r+1}$. Only the coroutine (k,l_1) is principal. Notice that if $r > 1$, nodes $(l_1,\ldots,l_{r-1})$ appear only in their principal coroutine, because a nonprincipal coroutine arises only at points where the path to 0 switches directions.

If there are s direction-switching vertices k such that the path to 0 begins $k \rightarrow l_1 \leftarrow l_2$, the 2^s independent settings of their bits all appear in at least one labeling. And if there are t direction-switching vertices k such that the path to 0 begins $k \leftarrow l_1 \rightarrow l_2$, the same remark applies to the 2^t independent settings of *their* individual bits. Therefore if the number of coroutines (k,l) exceeds $(c+1)n$, the number of labelings must exceed $2^{\max(s,t)} \ge 2^{c/2}$.

19. In Section 1.4.2 of *Fundamental Algorithms*, I wrote, "Initialization of coroutines tends to be a little tricky, although not really difficult." Perhaps I should reconsider that statement in the light of the present program, because the initialization of all the coroutines considered here is more than a little tricky.

In fact, I've decided not to implement the coroutines directly, although a simulation along the lines of KODA-RUSKEY would make a nice exercise. My real goal is to come up with a loopless implementation, because I was told that no such implementation (nor even an implementation with constant *amortized* time) was presently known.

[Readers who do take the time to work out the exercise suggested in the previous paragraph will notice a interesting difference with respect to the previous case: The *parent* pointers in the coroutine implementation of KODA-RUSKEY are essentially static, but now they must in general be dynamic. For example, in a graph like $0 \leftarrow 1 \rightarrow 2 \leftarrow 3$, coroutine $(3, 2)$ is called by both $(2, 0)$ and $(2, 1)$. This is related to the key difficulty we will face in coming up with a loopless way to solve our more general problem.]

20. A generalized fringe. The loopless implementation in KODA-RUSKEY is based on a notion called the *fringe*, which is a linear list containing nodes that are either *active* or *passive*. We are dealing here with a generalization of the problem solved there, so we naturally seek an extension of the fringe concept in hopes that fringes will help us again.

Each loopless KODA-RUSKEY step consists of four basic operations:

1) Find the rightmost active node, p;
2) Complement $bit[p]$;
3) Insert or delete appropriate fringe nodes at the right of p;
4) Make p passive and activate all nodes to its right.

Suitable data structures make all these operations efficient. Fortunately, the same scheme does in fact handle our more general problem, except that operation (3) must now involve both insertion and deletion.

The root node 0 is always present in the fringe. And if k is any node in the fringe, it is immediately followed by its current *descendants*, which are defined as follows: If $bit[k] = 0$, the descendants of k are the nodes a for $a \in A_k$ and their descendants. If $bit[k] = 1$, the descendants of k are the nodes b for $b \in B_k$ and their descendants.

21. An example will make this clear; for simplicity, we will consider only the subgraph G_1 on the vertices $\{1, 2, 3, 4\}$ of our larger example. The initial labeling 0000 means that the fringe begins with three nodes

$$1_0 \quad 2_0 \quad 4_0\,,$$

where the subscript on k indicates the value of $bit[k]$. Nodes 2_0 and 4_0 have no descendants, because $A_2 = A_4 = \emptyset$. All nodes are initially active.

Since 4_0 is the rightmost active node, we complement $bit[4]$, and the fringe becomes

$$1_0 \quad 2_0 \quad \overline{4}_1\,.$$

The bar over 4 indicates that this node is now passive. Again, $\overline{4}_1$ has no descendants, this time because $B_4 = \emptyset$.

The next step complements $bit[2]$, and $B_2 = \{3\}$ now enters the fringe:

$$1_0 \quad \overline{2}_1 \quad 3_0 \quad 4_1\,.$$

The initial value of $bit[3]$ is $\beta'_{23} = 0$.

22. Proceeding in this way, the first eight steps can be summarized as follows:

$$\begin{array}{llll}
1_0 & 2_0 & 4_0 & \\
1_0 & 2_0 & \overline{4}_1 & \\
1_0 & \overline{2}_1 & 3_0 & 4_1 \\
1_0 & \overline{2}_1 & 3_0 & \overline{4}_0 \\
1_0 & \overline{2}_1 & \overline{3}_1 & 4_0 \\
1_0 & \overline{2}_1 & \overline{3}_1 & \overline{4}_1 \\
\overline{1}_1 & 3_1 & & \\
\overline{1}_1 & \overline{3}_0 & &
\end{array}
\begin{array}{l}
\cdots \text{ complement } bit[4] \\
\cdots \text{ complement } bit[2] \\
\cdots \text{ complement } bit[4] \\
\cdots \text{ complement } bit[3] \\
\cdots \text{ complement } bit[4] \\
\cdots \text{ complement } bit[1] \\
\cdots \text{ complement } bit[3]
\end{array}$$

and at this point all of the labelings have been generated. Restarting the process causes them to be regenerated in reverse order, stopping again when the original starting point is reached:

$$\begin{array}{llll}
1_1 & 3_0 & & \\
1_1 & \overline{3}_1 & & \\
\overline{1}_0 & 2_1 & 3_1 & 4_1 \\
\overline{1}_0 & 2_1 & 3_1 & \overline{4}_0 \\
\overline{1}_0 & 2_1 & \overline{3}_0 & 4_0 \\
\overline{1}_0 & 2_1 & \overline{3}_0 & \overline{4}_1 \\
\overline{1}_0 & \overline{2}_0 & 4_1 & \\
\overline{1}_0 & \overline{2}_0 & \overline{4}_0 &
\end{array}
\begin{array}{l}
\cdots \text{ complement } bit[3] \\
\cdots \text{ complement } bit[1] \\
\cdots \text{ complement } bit[4] \\
\cdots \text{ complement } bit[3] \\
\cdots \text{ complement } bit[4] \\
\cdots \text{ complement } bit[2] \\
\cdots \text{ complement } bit[4]
\end{array}$$

23. Let's look more closely at what happens when $bit[k]$ changes its state. Suppose the rightmost active node is k_0. Then k_0 is immediately followed in the fringe by its descendants; those descendants were all passive, but we can reactivate them in anticipation of step (4). The current descendants of k_0 are the elements of A_k and *their* descendants, and each element $a \in A_k$ has $bit[a] = \alpha'_{ka}$.

We will say that the *entourage* of k_0 is the contents of the fringe at the beginning of the process that would generate the Gray path for the subgraph rooted at k; and the entourage of k_1 is the contents of the fringe at the end of that process. Thus, the small example we've just seen shows us that the entourage of 1_0 is $1_0\,2_0\,4_0$, and the entourage of 1_1 is $1_1\,3_0$.

In general, the entourage of k_0 consists of k_0 itself followed by the consecutive entourages of a_t for each $a \in A_k$, with $t = \alpha_{ka}$, in order of increasing a. We find therefore that the entourage of 0_0 in our main example is

$$0_0 \quad 1_0 \quad 2_0 \quad 4_0 \quad 5_1 \quad 6_1 \quad 8_0\,;$$

this is the initial contents of the fringe. The corresponding entourage of 0_1—the final fringe—is

$$0_1 \quad 3_1 \quad 6_1 \quad 7_0 \quad 8_0\,.$$

The final element of an entourage always has the form j_s where either $s = 0$ and $A_j = \emptyset$ or $s = 1$ and $B_j = \emptyset$. Here is a short program that locates the final j value at the end of the entourage for k_t:

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = n; k ≥ 0; k−−)
    for (j = 0; j ≤ 1; j++) {
      i = list[k][j];
      if (i) fj[k][j] = fj[i→id][i→alfprime ⊕ i→del];
      else fj[k][j] = k;
    }
```

24.

```
⟨ Global variables 4 ⟩ +≡
  char fj[maxn][2];      /* final vertices in the entourages of k0, k1 */
```

25. At the point where $bit[k]$ is about to change from 0 to 1, the fringe following k_0 contains k's *transition string* τ_{k0}, which consists again of the consecutive entourages a_t of all $a \in A_k$, but this time with $t = \alpha'_{ka}$ instead of $t = \alpha_{ka}$. The fringe is then modified from '$\ldots\, k_0\, \tau_{k0}\, \ldots$' to '$\ldots\, k_1\, \tau_{k1}\, \ldots$', where the transition string τ_{k1} consists of the consecutive entourages b_t of all $b \in B_k$, with $t = \beta'_{kb}$.

Thus the two transition strings for vertex 0 in our main example are $\tau_{00} = 1_1\, 3_0\, 5_1\, 6_1\, 8_0$ and $\tau_{01} = 3_0\, 6_1\, 7_0\, 8_0$. At the moment $bit[0]$ changes from 0 to 1, the fringe changes from $0_0\, \overline{1}_1\, \overline{3}_0\, \overline{5}_1\, \overline{6}_1\, \overline{8}_0$ to $\overline{0}_1\, 3_0\, 6_1\, 7_0\, 8_0$; and if we run the sequence backwards, it will go from $0_1\, \overline{3}_0\, \overline{6}_1\, \overline{7}_0\, \overline{8}_0$ to $\overline{0}_0\, 1_1\, 3_0\, 5_1\, 6_1\, 8_0$ when $bit[0]$ becomes 0.

26. Now comes a key observation: Recall that some elements of A_k and B_k are called "principal," namely the vertices adjacent to k (the "children" of k in tree terminology). *The nonprincipal vertices of A_k and B_k appear in both transition strings τ_{k0} and τ_{k1}, with their entourages in the same states.* Indeed, the entourage of each principal vertex a of A_k is a_1 followed by the entourages of all $b \in B_a$; and all such b are nonprincipal, because $\alpha'_{ka} = 1$ and $B_a \subseteq B_k$ when a is principal. Similarly, the entourage of each principal vertex b of B_k is b_0 followed by the entourages of all $a \in A_b$; and all such a are nonprincipal, because $\beta'_{kb} = 0$ and $A_b \subseteq A_k$ when b is principal.

Therefore, if we maintain the fringe as a doubly linked list—which we definitely want to do—the task of replacing one of k's transition strings by the other is fairly easy to describe at link level: When $bit[k]$ changes from 0 to 1, we remove the existing substring τ_{k0} from the fringe and replace it by the entourages of each $b \in B$, in increasing order of the b's. If b is principal, the entourage of b consists of b_0 followed by the entourages of all $a \in A_b$, and the latter vertices are nonprincipal; therefore the entourages of every such a are already properly linked within themselves, because they appear as substrings of τ_{k0}. Moreover, our assumption that the tree vertices were labeled in preorder guarantees that the entourages of all $a \in A_b$ appear consecutively in τ_{k0}; thus they are properly linked to each other, and all we need to do when forming the entourage of a principal vertex $b \in B$ is link it to the leftmost element of A_b. If on the other hand b is nonprincipal, none of its internal links need to be changed at all, because its entourage was already present as a substring of τ_{k0}.

All these entourages appear in preorder, in both τ_{k0} and τ_{k1}. Therefore, to change τ_{k0} to τ_{k1} we simply need to remove all of the principal members of A_k and insert all of the principal members of B_k, in their appropriate preorder position. For example, we saw a moment ago that the task of going from τ_{00} to τ_{01} consists of removing 1_1 and 5_1, then inserting 7_0. The total number of link-pairs we need to change is at most twice the number of elements of A_k plus twice the number of elements of B_k.

27. The preorder assumption makes still further simplification possible. Indeed, we know that the special case in KODA-RUSKEY requires at most two splices per transition, so we can expect to discover a generalization of the principle that led to such an improvement.

Suppose all of the elements of $A_k \cup B_k$ have been merged into preorder, thus giving us the "union" of τ_{k0} and τ_{k1}. Suppose further that two vertices $b\, b'$ of B_k are adjacent in this union, where b is principal. It follows that b' is also principal; otherwise some element of A_k would intervene. When τ_{k0} is being replaced by τ_{k1} in the fringe, we therefore need to insert both b and b'. Fortunately, they were properly linked to each other when they last left the fringe, and neither one has entered the fringe since then; so they still are properly linked. (We also need to be sure that they were linked properly to each other during the initialization, before processing starts.) The same argument applies when two vertices $a\, a'$ of A_k are adjacent and a is principal.

One can often, but not always, avoid some of the link-updating with respect to τ_{k0} and τ_{k1} by cleverly reordering the subtrees of k.

28. An algorithm that changes transition strings as just described will run in constant amortized time per output, because the cost of changing links in the fringe can be charged to the processing time when the corresponding fringe element next becomes passive.

But it looks like bad news for our goal of loopless computation, because we might need to do $\Omega(n)$ operations when $bit[k]$ changes.

No, all is not lost: We can leave "stale" links in the fringe, fixing them one at a time, just before the algorithm needs to look at them!

Namely, suppose the transition string τ_{k1} consists of substrings $\sigma_1\sigma_2\ldots\sigma_s$ that are properly linked within themselves but not necessarily to each other. We can prepare a list of "fixup instructions" $(x_1,\ldots,x_s)$, where x_j says that the first element of σ_j should be joined to the last element of σ_{j-1}, or to element k_1 if $j=1$. And we can plant a flag that will cause x_j to be performed just as the first element of σ_j becomes passive. The execution of x_j will then move the flag in preparation for x_{j-1}, or it will remove the flag if $j=1$.

29. Data structures. Now that we know basically what needs to be done, we are ready to design the records that will be linked together to form the fringe. These records, which we will call fnodes, naturally contain *left* and *right* fields, because the fringe is doubly linked (modulo stale pointers). An fnode also contains a *fixup* field, which (if non-Λ) points to information that will correct a stale *left* pointer. And of course there's also a *bit* field, giving the status of $bit[k]$ in fnode number k.

Each fnode also has a *focus* pointer, which implicitly classifies fringe elements as active or passive, just as it did in the KODA-RUSKEY program: Usually $p\neg focus = p$, except when p is a passive node to the immediate left of an active node. In the latter case, $p\neg focus$ points to the first active node to the left of p.

Besides these dynamically changing fields, each fnode also contains two static fields that are precomputed during the initialization, namely $tau[0]$ and $tau[1]$. These fields refer to sequential lists that specify the transition strings τ_{k0} and τ_{k1} for fnode number k.

```
⟨ Type definitions 6 ⟩ +≡
  typedef struct fnode_struct {
    char bit;      /* either 0 or 1; always 0 when an A-child's bit is 0, always 1 when a B-child's bit is 1 */
    struct fnode_struct *left, *right;      /* neighbors in the fringe */
    struct fnode_struct **fixup;      /* remedy for a stale left link */
    struct fnode_struct *focus;      /* red-tape cutter for efficiency */
    struct fnode_struct **tau[2];      /* list of transition string link points */
  } fnode;
```

30. The fringe is extended to contain a special fnode called *head*, which makes the list circular. In other words, $head\neg right$ and $head\neg left$ are the leftmost and rightmost elements of the fringe. Also, $head\neg left\neg focus$ is the rightmost active element.

```
#define head (vertex + 1 + n)
⟨ Global variables 4 ⟩ +≡
  fnode vertex[maxn + 1];      /* the collection of all potential fringe nodes */
```

31. And how should we store the *tau* information? Let's "compile" the set of necessary link changes into a sequential list $(r_1, l_1, \ldots, r_t, l_t, \Lambda)$ with the meaning that we want to set $l_j\neg right = r_j$ and $r_j\neg left = l_j$. The case $j = 1$ is, however, an exception, because it refers to a potentially unknown part of the fringe (lying to the right of τ_{k0} and τ_{k1}); in that case we want to set we want to set $l_j\neg right = r_j\neg right$ and $r_j\neg right\neg left = l_j$.

```
⟨ Initialize the data structures 7 ⟩ +≡
  tau_ptr = tau_table;
  for (k = 0; k ≤ n; k++) {
    ⟨ Merge the sets A_k and B_k, retaining preorder 33 ⟩;
    ⟨ Compile the link-change strings vertex[k].tau[ ] 34 ⟩;
  }
```

32.
```
⟨ Global variables 4 ⟩ +≡
  fnode *tau_table[((maxn * maxn) ≫ 1) + maxn + maxn];      /* elements of tau fields */
  fnode **tau_ptr;      /* the first unused place in tau_table */
  char ct[maxn];      /* ct[l] = 0 if ab[l] ∈ A_k, otherwise ct[l] = 1 */
  char verbose = 1;      /* should details of tau compilation be printed? */
  info *ab[maxn];      /* an element of A_k or B_k */
```

33. It is most convenient to sort in *decreasing* order here, so that the rightmost link-updates are listed first, because we have stored them in that order and we will need them in that order.

This part of the program is the most subtle, so I've included a provision for verbose printing during debugging sessions. A simple case in which $A_k = \{5\}$ and $B_k = \emptyset$ will be printed as `(a5)`. A slightly more complicated case in which $A_k = \{2\}$ and $B_k = \{4\}$ would be `(a2)(b4)` if both are principal, or `(a2b4)` if only 2 is principal. A more complicated case, used as an example below, has $A_k = \{1, 2, 5, 8, 11\}$ and $B_k = \{3, 4, 6, 7, 9, 10\}$, with elements $\{1, 2, 5, 7, 9, 10\}$ principal; this would be printed as `(a1)(a2b3b4)(a5b6)(b7a8)(b9)(b10a11)`.

Recall that an element j of $A_k \cup B_k$ is principal if and only if it is a child of k. The parentheses in the verbose printout are used to group k's children with their nonprincipal descendants.

```
#define principal(l)  (jj[ab[l]→id] ≡ k)
⟨ Merge the sets A_k and B_k, retaining preorder 33 ⟩ ≡
  for (i = list[k][0], ii = list[k][1], l = 0; i ∨ ii; l++) {
    if (¬ii ∨ (i ∧ i→id > ii→id)) ab[l] = i, ct[l] = 0, i = i→sib;
    else ab[l] = ii, ct[l] = 1, ii = ii→sib;
  }
  if (l ∧ verbose) {
    printf("Union%d:␣(", k);
    for (j = l − 1; ; ) {
      printf("%c%d", 'a' + ct[j], ab[j]→id);
      if (j ≡ 0) break;
      j−−;
      if (principal(j)) printf(")(");
    }
    printf(")\n");
  }
```

This code is used in section 31.

34. ```⟨ Compile the link-change strings vertex[k].tau[ ] 34 ⟩ ≡
  if (¬l) vertex[k].tau[0] = vertex[k].tau[1] = Λ;
  else {
    ab[l] = Λ;   /* sentinel at end of list */
    vertex[k].tau[1] = tau_ptr;
    ⟨ Compile vertex[k].tau[1] 35 ⟩;
    vertex[k].tau[0] = tau_ptr;
    ⟨ Compile vertex[k].tau[0] 40 ⟩;
    ⟨ Link siblings together 44 ⟩;
  }
```

This code is used in section 31.
```

35. Our job here is to specify the links that must change when the principal elements of A_k are deleted and the principal elements of B_k are inserted. In the simple example `(a5)` mentioned above, we simply set $r_1 = \&vertex[5]$ and $l_1 = vertex[k]$. The verbose printout will say '`k:5+`', meaning that $vertex[k]$ is to be followed in the fringe by the vertex that currently follows $vertex[5]$.

In the next simplest example, `(a2)(b4)`, we will set $r_1 = \&vertex[2]$, $l_1 = \&vertex[4]$, $r_2 = \&vertex[4]$, $l_2 = \&vertex[k]$; the verbose printout will symbolize this by '`4:2+ k:4`'. The result of `(a2b4)` would, on the other hand, be '`k:2+`', because only $vertex[2]$ needs to be removed from the fringe when $vertex[4]$ is nonprincipal.

Finally, the instructions compiled from the complex example `(a1)(a2b3b4)(a5b6)(b7a8)(b9)(b10a11)` would be

```
10:8r+  8r:9  7:8  6r:7  4r:6  k:3.
```

The '`r`' here denotes the rightmost element of a nonprincipal vertex's entourage. (As explained earlier, the instruction '`9:10`' need not be compiled.)

The reader might be able to understand from these examples why the author took time out to think before writing this part of the code.

```
⟨ Compile vertex[k].tau[1] 35 ⟩ ≡
  ⟨ Find r_1 for tau[1] 36 ⟩;
  ⟨ Find l_1 for tau[1] 38 ⟩;
  while (ab[l]) ⟨ Find the next r_j and l_j for tau[1] 39 ⟩;
  tau_ptr++;      /* the tau list terminates with Λ */
```

This code is used in section 34.

36.

```
⟨ Find r_1 for tau[1] 36 ⟩ ≡
  for (l = 0; ¬principal(l); l++) ;      /* nonprincipals at the end can be ignored */
  if (ct[l]) {      /* principal b */
    for (j = l + 1; ab[j] ∧ ct[j]; j++) ;
    if (ab[j]) {
      if (verbose) sprintf(rbuf, "%dr+", ab[j]→id);
      *tau_ptr++ = vertex + fj[ab[j]→id][ab[j]→alfprime];
    } else {
      if (verbose) sprintf(rbuf, "%d+", k);
      *tau_ptr++ = vertex + k;
    }
  } else {      /* principal a */
    if (verbose) sprintf(rbuf, "%d+", ab[l]→id);
    *tau_ptr++ = vertex + ab[l]→id;
  }
```

This code is used in section 35.

37.

```
⟨ Global variables 4 ⟩ +≡
  char rbuf[8];      /* symbolic form of r_1 for verbose printing */
```

38. Once we've found r_1, principal a's at the end can be ignored.

```
⟨ Find l_1 for tau[1] 38 ⟩ ≡
  while (ab[l] ∧ ¬ct[l] ∧ principal(l)) l++;
  if (¬ab[l]) {
    if (verbose) printf("␣%d:%s\n", k, rbuf);
    *tau_ptr++ = vertex + k;
  } else if (principal(l)) {
    if (verbose) printf("␣%d:%s", ab[l]→id, rbuf);
    *tau_ptr++ = vertex + ab[l]→id;
  } else {
    if (verbose) printf("␣%dr:%s", ab[l]→id, rbuf);
    *tau_ptr++ = vertex + fj[ab[l]→id][ab[l]→alfprime];
  }
```

This code is used in section 35.

39. At this point $ab[l]$ is the **info** node from which we produced l_{j-1}. If it refers to a principal element, it's an element b that is absent from τ_{k0} but present in τ_{k1}. Otherwise it's a nonprincipal element, which appears in both τ_{k0} and τ_{k1}.

```
⟨ Find the next r_j and l_j for tau[1] 39 ⟩ ≡
  {
    if (principal(l))
      for (l++; ab[l] ∧ ct[l] ∧ principal(l); l++) ;
    else for (l++; ¬principal(l); l++) ;
    *tau_ptr++ = vertex + ab[l − 1]→id;
    if (ab[l] ∧ ¬ct[l] ∧ principal(l))
      for (l++; ab[l] ∧ ¬ct[l] ∧ principal(l); l++) ;
    if (¬ab[l]) {
      if (verbose) printf("␣%d:%d\n", k, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + k;
    } else if (principal(l)) {
      if (verbose) printf("␣%d:%d", ab[l]→id, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + ab[l]→id;
    } else {
      if (verbose) printf("␣%dr:%d", ab[l]→id, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + fj[ab[l]→id][ab[l]→alfprime];
    }
  }
```

This code is used in section 35.

40. The next few sections are identical to the previous ones, with a's and b's swapped. (Unless I have erred.) The compiled instructions for $tau[0]$ in the complex example are

```
8r:10+  6r:8  5:6  4r:5  2:3  k:1.
```

```
⟨ Compile vertex[k].tau[0] 40 ⟩ ≡
  ⟨ Find r_1 for tau[0] 41 ⟩;
  ⟨ Find l_1 for tau[0] 42 ⟩;
  while (ab[l]) ⟨ Find the next r_j and l_j for tau[0] 43 ⟩;
  tau_ptr++;      /* the tau list terminates with Λ */
```

This code is used in section 34.

41. ⟨ Find r_1 for $tau[0]$ 41 ⟩ ≡

```
for (l = 0; ¬principal(l); l++) ;    /* nonprincipals at the end can be ignored */
if (¬ct[l]) {    /* principal a */
  for (j = l + 1; ab[j] ∧ ¬ct[j]; j++) ;
  if (ab[j]) {
    if (verbose) sprintf(rbuf, "%dr+", ab[j]→id);
    *tau_ptr++ = vertex + fj[ab[j]→id][ab[j]→alfprime];
  } else {
    if (verbose) sprintf(rbuf, "%d+", k);
    *tau_ptr++ = vertex + k;
  }
} else {    /* principal b */
  if (verbose) sprintf(rbuf, "%d+", ab[l]→id);
  *tau_ptr++ = vertex + ab[l]→id;
}
```

This code is used in section 40.

42. ⟨ Find l_1 for $tau[0]$ 42 ⟩ ≡

```
while (ab[l] ∧ ct[l] ∧ principal(l)) l++;
if (¬ab[l]) {
  if (verbose) printf("␣%d:%s\n", k, rbuf);
  *tau_ptr++ = vertex + k;
} else if (principal(l)) {
  if (verbose) printf("␣%d:%s", ab[l]→id, rbuf);
  *tau_ptr++ = vertex + ab[l]→id;
} else {
  if (verbose) printf("␣%dr:%s", ab[l]→id, rbuf);
  *tau_ptr++ = vertex + fj[ab[l]→id][ab[l]→alfprime];
}
```

This code is used in section 40.

43. At this point $ab[l]$ is the **info** node from which we produced l_{j-1}. If it refers to a principal element, it's an element a that is absent from τ_{k1} but present in τ_{k0}. Otherwise it's a nonprincipal element, which appears in both τ_{k0} and τ_{k1}.

```
⟨ Find the next r_j and l_j for tau[0] 43 ⟩ ≡
  {
    if (principal(l))
      for (l++; ab[l] ∧ ¬ct[l] ∧ principal(l); l++) ;
    else for (l++; ¬principal(l); l++) ;
    *tau_ptr++ = vertex + ab[l − 1]→id;
    if (ab[l] ∧ ct[l] ∧ principal(l))
      for (l++; ab[l] ∧ ct[l] ∧ principal(l); l++) ;
    if (¬ab[l]) {
      if (verbose) printf("␣%d:%d\n", k, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + k;
    } else if (principal(l)) {
      if (verbose) printf("␣%d:%d", ab[l]→id, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + ab[l]→id;
    } else {
      if (verbose) printf("␣%dr:%d", ab[l]→id, *(tau_ptr − 1) − vertex);
      *tau_ptr++ = vertex + fj[ab[l]→id][ab[l]→alfprime];
    }
  }
```

This code is used in section 40.

44. We need to link the siblings of a family together if they are adjacent and of the same type, because of one of the optimization we're doing. So we might as well link *all* siblings together.

```
⟨ Link siblings together 44 ⟩ ≡
  for (l = 0, p = Λ; ab[l]; l++)
    if (principal(l)) {
      q = vertex + ab[l]→id;
      if (p) q→right = p, p→left = q;
      p = q;
    }
```

This code is used in section 34.

45. ⟨ Local variables 5 ⟩ +≡

```
  fnode *p, *q, *r;
```

46. Doing it. The time has come to construct the loopless implementation in practice, as we have been doing so far in theory.

```
⟨ Generate the answers 46 ⟩ ≡
  while (1) {
    ⟨ Print out all the current bits 53 ⟩;
    ⟨ Set p to the rightmost active node of the fringe, and activate everything to its right 48 ⟩;
    if (p ≠ head) {
      if (p→fixup) ⟨ Repair the stale pointer p→left 50 ⟩;
      if (p→bit ≡ 0) ⟨ Move forward, setting p→bit = 1 51 ⟩
      else ⟨ Move backward, setting p→bit = 0 52 ⟩
    } else if (been_there_and_done_that) break;
    else {
      printf("...␣and␣now␣we␣generate␣in␣reverse:\n");
      been_there_and_done_that = 1; continue;
    }
    ⟨ Make node p passive 49 ⟩;
  }
```

This code is used in section 1.

47.
```
⟨ Global variables 4 ⟩ +≡
  char been_there_and_done_that;      /* have we completed a cycle already? */
```

48. The nice thing is that the algorithm not only is loopless, its operations are simple.

```
⟨ Set p to the rightmost active node of the fringe, and activate everything to its right 48 ⟩ ≡
  q = head→left;
  p = q→focus;
  q→focus = q;
```

This code is used in section 46.

49. At this point we know that $p\rightarrow right$ (more precisely, the node that would be $p\rightarrow right$ if links weren't stale) is active. And we also know that $p\rightarrow left$ is not stale.

```
⟨ Make node p passive 49 ⟩ ≡
  q = p→left;
  p→focus = q→focus;
  q→focus = q;
```

This code is used in section 46.

50.
```
⟨ Repair the stale pointer p→left 50 ⟩ ≡
  {
    q = *(p→fixup), r = *(p→fixup + 1);
    p→left = q, q→right = p;
    if (r) r→fixup = p→fixup + 2;
    p→fixup = Λ;
  }
```

This code is used in section 46.

51. ⟨ Move forward, setting $p\neg bit = 1$ 51 ⟩ ≡

```
{
  p¬bit = 1;
  if (p¬tau[1]) {
    q = (*(p¬tau[1]))¬right;
    r = *(p¬tau[1] + 1);
    q¬left = r, r¬right = q;
    r = *(p¬tau[1] + 2);
    if (r) r¬fixup = p¬tau[1] + 3;
  }
}
```

This code is used in section 46.

52. ⟨ Move backward, setting $p\neg bit = 0$ 52 ⟩ ≡

```
{
  p¬bit = 0;
  if (p¬tau[0]) {
    q = (*(p¬tau[0]))¬right;
    r = *(p¬tau[0] + 1);
    q¬left = r, r¬right = q;
    r = *(p¬tau[0] + 2);
    if (r) r¬fixup = p¬tau[0] + 3;
  }
}
```

This code is used in section 46.

53. ⟨ Print out all the current bits 53 ⟩ ≡

```
for (k = 0; k ≤ n; k++) putchar('0' + vertex[k].bit);
if (verbose) ⟨ Print the fringe in symbolic form 54 ⟩;
putchar('\n');
```

This code is used in section 46.

54. Once again, I'm writing optional code that should help me gain confidence (and pinpoint errors) while debugging. Such code also ought to help an observer understand what the program thinks it is doing, or at least what I thought it should be doing when I wrote it.

The fringe is printed right-to-left, because that's the way the implicit parts of the data need to be interpreted (namely, the active/passive bits and the stale-pointer corrections). I could, of course, take the trouble to reverse the order and make the printout more intuitive; but hey, this is for educated eyes only.

Passive vertices are shown in parentheses.

⟨ Print the fringe in symbolic form 54 ⟩ ≡

```
for (l = 0, q = head¬left, p = q¬focus; q ≠ head; q = r) {
  printf("␣%s%d_%d%s", q ≠ p ? "(" : "", q − vertex, q¬bit, q ≠ p ? ")" : "");
  ⟨ Set r to the non-stale form of q¬left 56 ⟩;
  if (q ≡ p) p = r¬focus;
}
```

This code is used in section 53.

55. Here we need to maintain a stack of currently active fixups.

```
⟨ Type definitions 6 ⟩ +≡
  typedef struct {
    fnode *vert;      /* vertex with a stale left pointer */
    fnode **ref;      /* tau string with info about future vertices */
  } fstack_node;
```

56.
```
⟨ Set r to the non-stale form of q→left 56 ⟩ ≡
  if (q→fixup) fstack[++l].vert = q, fstack[l].ref = q→fixup;
  if (q ≡ fstack[l].vert) {
    putchar('!');      /* indicate that a stale link is being fixed */
    r = *(fstack[l].ref);
    if (*(fstack[l].ref + 1)) fstack[l].vert = *(fstack[l].ref + 1), fstack[l].ref += 2;
    else l−−;
  } else r = q→left;
```
This code is used in section 54.

57.
```
⟨ Global variables 4 ⟩ +≡
  fstack_node fstack[maxn];      /* stack used in verbose printout */
```

58. Priming the pump. The program is complete except for one niggling detail: We need to set up the initial contents of the fringe. Everything will maintain itself, once the whole structure is up and running, but how do we get chickens before we have eggs?

Here are two recursive subroutines that come to the rescue. The first one sets $vertex[j].bit = 1$ in all vertices j such that $k \preceq j$. The second one contributes the entourage of given vertex in a given state to the current fringe.

```
⟨ Subroutines 15 ⟩ +≡
  void setbits(char k)
  {
    register info *i;
    vertex[k].bit = 1;
    for (i = list[k][0]; i; i = i→sib)
      if (jj[i→id] ≡ k) setbits(i→id);
  }
```

59. A global variable called *cur_vert* points to the left end of the fringe-so-far. We construct the entourages from right to left, because that's the way the *list* info is set up.

All bits are assumed to be zero initially; therefore we don't need a routine to set them zero, only the *setbits* routine to make some of them nonzero.

```
⟨ Subroutines 15 ⟩ +≡
  void entourage(char k, char t)
  {
    register fnode *p = vertex + k;
    register info *i;
    if (t) setbits(k);
    for (i = list[k][t]; i; i = i→sib) entourage(i→id, i→alfprime ⊕ i→del);
    cur_vert→left = p, p→right = cur_vert, cur_vert = p;
  }
```

60.
```
⟨ Global variables 4 ⟩ +≡
  fnode *cur_vert;      /* front of a doubly linked list under construction */
```

61. And now we're done!

```
⟨ Initialize the data structures 7 ⟩ +≡
  for (k = 0; k ≤ n + 1; k++) vertex[k].focus = &vertex[k];
  cur_vert = head;
  entourage(0, 0);
  cur_vert→left = head, head→right = cur_vert;
```

62. Comment about looplessness. In practice, the algorithm would run a bit faster if the *fixup* links were dispensed with and all splicing were done directly at transition time. Constant *amortized* time—or, rather, total execution time in general—is the main criterion in most applications; there usually is no reason to care whether some generation steps are slow and others are fast, as long as the entire job is done quickly.

However, the construction of loopless algorithms carries an academic cachet: "Look at how clever I am, I can even do it looplessly!" That's why this program has gone the extra mile to assure constant time per generation step, even though the added complications may well have been a step backwards from a practical standpoint. [Excuse me, I'm thinking only of sequential computation here; loopless algorithms can have definite advantages with respect to parallel processing.]

On the other hand, many academic purists may reasonably claim that I have not actually reached my stated goal. When Gideon Ehrlich introduced the notion of loopless implementations in *JACM* **20** (1973), 500–513, he insisted that the initialization time be $O(n)$; the program above can only guarantee an initialization time that is $O\bigl(\sum_k(\|A_k\|+\|B_k\|)\bigr)$, and that sum can be $\sim n^2/4$. Clearly the initialization time ought to be substantially smaller than the number of outputs, or a loopless algorithm would be trivial. But I think it's fair to allow initialization to take as long as, say, $O\bigl(\min(m,n^2)\bigr)$ when there are m outputs, especially when m is usually (but not always) exponential in n.

63. Index.

LI-RUSKEY