November 24, 2020 at 13:24

1. Introduction. This program does calculations with skew ternary trees, and exhibits the corresponding nonseparable planar graphs. It implements some simple algorithms that I discovered in November, 2013, based on ideas of Alberto Del Lungo, Francesco Del Ristoro, and Jean-Guy Penaud [*Theoretical Computer Science* **233** (2000), 201–215]. I wrote it in order to learn more about the seemingly magical properties of this amazing correspondence.

I apologize for having no time to provide a better user interface, or to give more extensive commentary. Ideally an interactive system should be written, with excellent graphics to display and manipulate the trees and graphs in an intuitive fashion; color should be used to exhibit at least some of the fascinating patterns that are present, etc. I'm hoping that some reader will be motivated to write such an "app," because it will certainly be a fabulously instructive toy.

I will at least try to define and explain the basics in this document. A ternary tree is either empty, or it consists of a root node and three ternary trees called the left, middle, and right subtrees of the root. The roots of those subtrees are called the left, middle, and right children of the root node. (This definition is strictly analogous to the familiar definition of binary trees, in Section 2.3 of my book *Fundamental Algorithms*.)

Furthermore we extend the ternary tree by placing "buds" in the positions of empty subtrees. And we also introduce a bud at the top, attached to the root node. In this way every node, including the root, is attached to exactly four other nodes or buds; and every bud is attached to exactly one other node or bud. We will give labels to each node and to each bud, in order to exhibit fine details of the structure.

An extended ternary tree with n nodes always has $2n+2$ buds. (Notice that this result, which is easily proved by induction on n, holds in particular when the ternary tree is empty. In that case, $n=0$ and there simply are two buds joined to each other.)

The embedding of such a tree in the plane leads to a family of $2n+2$ extended ternary trees that are "cyclically equivalent," as illustrated below. There's one such tree for each bud, obtained by placing that bud at the top and letting everything else "hang down" from it in. Each of these trees has a different root bud, but not necessarily a different root, because different buds can lead to distinct trees with the same root node.

The $2n+2$ buds can always be paired up into $n+1$ groups of two. Indeed, we can find the mate of any bud by proceeding on a unique path away from that bud, always taking the middle branch whenever there are three choices for the next step, and continuing until another bud is encountered.

Every node and every bud is assigned a *rank* in the following natural way: The root node and root bud have rank zero; and the left, middle, and right children of a rank r node have ranks $r-1$, r, and $r+1$, respectively.

A *skew ternary tree* is a ternary tree for which all nodes have nonnegative rank.

For example, the skew ternary tree shown here has 6 nodes and 7 pairs of buds. Ranks are shown in red.

Notice that there's one bud of rank -1 for every node of rank 0.

2. Fact: *Every family of* $2n+2$ *cyclically equivalent ternary trees includes exactly four skew ternary trees.*

Moreover, this theorem—which is the main reason for the existence of this program—has an astonishingly simple proof. The idea is to consider the $n-1$ edges that go between nodes of the ternary, and to treat each edge `U` — `V` as a pair of arcs `U` $\longrightarrow$ `V` and `V` $\longrightarrow$ `U`. That gives us $2n-2$ arcs. And there's a natural way to

match those arcs to $2n-2$ of the $2n+2$ buds, by means of $2n-2$ noncrossing filaments as illustrated here:

More precisely, imagine an ant, named Alice, who crawls around the periphery of the tree. Alice starts in state -2, just to the right of bud number 0. She increases her state by 1 whenever she passes a bud; and she decreases her state by 1 whenever she passes an arc. Then she will be in state $-2+(2n+2)-(2n-2)=+2$

when she returns to its starting point:

And if she keeps on crawling, she will repeat the same pattern, but with her state increased by 4.

The key fact is that Alice is in state k whenever she reaches a bud of rank k—except for the starting bud, when she's in state ± 2. Thus the unmatched buds correspond to the skew ternary trees; in this example the trees whose roots hang down from buds 0, 4, 5, and 6 will have no nodes of negative rank. Conversely, a ternary tree that begins at a matched bud will have at least one buds of rank < -1, so it will have at least one node of rank < 0.

QED.

(A reader who understands this proof will also be able to show that *every family of* $4n + 2$ *cyclically equivalent quinary trees includes exactly six skew quinary trees.* And so on.)

3. The four skew ternary trees of a cyclic family turn out to have remarkable properties. Let's look at

the state transitions that Alice would encounter by starting at each of the four unmatched buds:

The corresponding skew ternary trees, which we might as well show without their buds, are

$T =$; $T^+ =$

Notice the notation used here, based on a well-defined operator $T \mapsto T^+$ that takes one skew ternary tree to another. Since $T^{++++} = T$, we also abbreviate T^{+++} as T^-; and T^{++} can also be called T^{--}.

4. One of the first goals of this program will be to compute the "conjugates" T^+, T^{++}, and $T^{+++} = T^-$, given a skew ternary tree T. That tree is specified on the command line, as a sequence of four-character arguments `abcd`: The first character, `a`, names a node; the next three characters name that node's children, using '`-`' for an empty child. For example, the tree T above could be specified by the six command-line arguments

```
A-BD    B--C    C---    DE-F    E---    F---
```

in some order. There should be one argument for each node. The program parses the arguments and checks to make sure that they actually do define a skew ternary tree.

5. OK, we now know the definition of skew ternary trees, and it's time to begin coding. Here's the structure of the program as a whole:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  ⟨ Type definitions 7 ⟩
  ⟨ Global variables 8 ⟩
  ⟨ Assertion failure subroutine 66 ⟩
  ⟨ Subroutines 11 ⟩
  main(int argc, char *argv[])
  {
    register int c, i, j, k, p;
    ⟨ Process the command line 6 ⟩;
    ⟨ Find and print the three conjugates of T 16 ⟩;
    ⟨ Find and print the corresponding planar maps 49 ⟩;
  }
```

6. We don't deal with the empty tree; there must be at least one node.

```
⟨ Process the command line 6 ⟩ ≡
  if (argc ≡ 1) {
    fprintf(stderr, "Usage: %s node_1 node_2 ... node_n\n", argv[0]);
    exit(−1);
  }
  ⟨ Parse the arguments; report a problem and exit if they don't define a skew tree 9 ⟩;
```

This code is used in section 5.

7. Parsing. First things first: We gotta get the tree into memory in a convenient form. The basic data structure has *left*, *middle*, *right*, and *parent* fields in each node, and a few other things we'll need as we go along. Buds are represented by negative integers; other links are to a node's 8-bit character code.

(At most 64 different visible character codes are used in this implementation, namely `'*'` and the 63 from `'@'` to `'~'`.)

```
#define sentinel 999
#define maxcodes 64
```

⟨ Type definitions 7 ⟩ ≡

```
typedef struct node_struct {
  int left;      /* the left child */
  int middle;    /* the middle child */
  int right;     /* the right child */
  int parent;    /* the parent */
  int rank;      /* set to sentinel at input time; later is the actual rank */
} node;
typedef struct bud_struct {
  int parent;    /* the parent */
  int rank;      /* the actual rank */
  int stepno;    /* step number in the state chart (see below) */
} bud;
```

See also section 14.

This code is used in section 5.

8. ⟨ Global variables 8 ⟩ ≡

```
node inputnode[256];    /* actually only nodes '@' thru '~' are used */
bud inputbud[512];      /* data for bud number k is stored in inputbud[−k] */
int buds;      /* this many buds have been created so far */
int n;      /* the number of nodes in the tree */
```

See also sections 15, 30, and 41.

This code is used in section 5.

9. The initial setup is straightforward, although a bit tedious.

```
#define abort0(message, code)
        { fprintf(stderr, "%s!\n", message);
          exit(code); }
#define abort1(message, j, code)
        { fprintf(stderr, "Bad arg (%s): %s!\n", argv[j], message);
          exit(code); }
#define abort2(message, j, c, code)
        { fprintf(stderr, "Bad arg (%s): Node '%c' %s!\n", argv[j], c, message);
          exit(code); }
⟨ Parse the arguments; report a problem and exit if they don't define a skew tree 9 ⟩ ≡
  for (j = 1; j < argc; j++) {
    if (strlen(argv[j]) ≠ 4) abort1("Must be four characters long", j, −10);
    c = argv[j][0];
    if (c < '@' ∨ c > '~') abort2("is not permitted", j, c, −15);
    if (inputnode[c].rank) abort2("has already been defined", j, c, −11);
    inputnode[c].rank = sentinel;
    p = argv[j][1];
    if (p ≠ '-') {
      if (p < '@' ∨ p > '~') abort2("is not permitted", j, p, −16);
      if (inputnode[p].parent) abort2("already has a parent", j, p, −12);
      inputnode[c].left = p, inputnode[p].parent = c;
    }
    p = argv[j][2];
    if (p ≠ '-') {
      if (p < '@' ∨ p > '~') abort2("is not permitted", j, p, −17);
      if (inputnode[p].parent) abort2("already has a parent", j, p, −13);
      inputnode[c].middle = p, inputnode[p].parent = c;
    }
    p = argv[j][3];
    if (p ≠ '-') {
      if (p < '@' ∨ p > '~') abort2("is not permitted", j, p, −18);
      if (inputnode[p].parent) abort2("already has a parent", j, p, −14);
      inputnode[c].right = p, inputnode[p].parent = c;
    }
  }
  n = argc − 1;
  ⟨ Introduce the buds and compute the ranks 10 ⟩;
```

This code is used in section 6.

10. We need to locate the root, which should be the unique input node that has no parent. Then we'll attach bud -1 to it.

Buds $-2k-1$ and $-2k-2$ are mates; hence the mate of bud x is bud $x \oplus 1$.

```
#define root inputbud[1].parent
⟨ Introduce the buds and compute the ranks 10 ⟩ ≡
  for (j = '@'; j ≤ '~'; j++) {
    if (inputnode[j].rank ∧ ¬inputnode[j].parent) {
      if (root) {
        fprintf(stderr, "Nodes␣'%c'␣and␣'%c'␣cannot␣both␣be␣roots!\n", root, j);
        exit(−20);
      }
      root = j;
    }
    if (inputnode[j].parent ∧ ¬inputnode[j].rank) {
      fprintf(stderr, "No␣data␣was␣supplied␣for␣node␣`%c'!\n", j);
      exit(−21);
    }
  }
  if (¬root) abort0("There's␣no␣root", −21);
  inputbud[1].rank = −2;      /* bud number 1 is the "root bud," above the root node */
  setmate(root);      /* locate and define its mate */
  fillbuds(root, 0);
  ⟨ Check that we've filled out the whole tree 13 ⟩;
```

This code is used in section 9.

11. The *setmate* subroutine allocates two new buds. Its parameter names the node where we discovered the existence of such buds.

In most cases, the mate is reached by going upward through middle links, then crossing from left to right (or vice versa) and going downward through middle links.

```
⟨ Subroutines 11 ⟩ ≡
  void setmate(int p)
  {
    register int q, d;
    buds += 2, q = 1 − buds;
    if (inputbud[buds − 1].parent) {
      if (buds > 2) confusion("bud␣parent␣already␣set");
      goto downward_mid;
    }
    inputbud[buds − 1].parent = p;
  upward: if (inputnode[p].middle ≡ q) {
      q = p, p = inputnode[p].parent;
      goto upward;
    }
    if (inputnode[p].left ≡ q) {
      q = p, p = inputnode[p].right, d = 1;
      goto downward;
    }
    if (inputnode[p].right ≡ q) {
      q = p, p = inputnode[p].left, d = −1;
      goto downward;
    }
    confusion("supposed␣parent␣node␣not␣apparent");
  downward_mid: q = p, p = inputnode[p].middle, d = 0;
  downward: if (p < 0) abort0("Mate␣mixup", −25);
    if (p > 0) goto downward_mid;
    if (d > 0) inputnode[q].right = −buds;
    else if (d < 0) inputnode[q].left = −buds;
    else inputnode[q].middle = −buds;
    inputbud[buds].parent = q;
  }
```

See also sections 12, 17, 18, 19, 21, 24, 32, 33, 42, and 55.

This code is used in section 5.

12. The main work of filling buds and setting ranks is done by a straightforward recursive procedure *fillbuds*, which traverses the ternary tree in preorder.

```
⟨ Subroutines 11 ⟩ +≡
  void fillbuds(int p, int r)
  {
    if (r < 0) {
      fprintf(stderr, "Not␣properly␣skewed:␣rank(%c)=-1!\n", p);
      exit(-30);
    }
    inputnode[p].rank = r;
    if (inputnode[p].left > 0) fillbuds(inputnode[p].left, r − 1);
    else {
      if (inputnode[p].left ≡ 0) inputnode[p].left = −buds − 1, setmate(p);
      inputbud[−inputnode[p].left].rank = r − 1;
    }
    if (inputnode[p].middle > 0) fillbuds(inputnode[p].middle, r);
    else {
      if (inputnode[p].middle ≡ 0) inputnode[p].middle = −buds − 1, setmate(p);
      inputbud[−inputnode[p].middle].rank = r;
    }
    if (inputnode[p].right > 0) fillbuds(inputnode[p].right, r + 1);
    else {
      if (inputnode[p].right ≡ 0) inputnode[p].right = −buds − 1, setmate(p);
      inputbud[−inputnode[p].right].rank = r + 1;
    }
  }
```

13. We've prepared a skewed ternary tree by filling in all the missing fields. But if the input is, say, '`A---B-B-`', the tree we've prepared won't contain all of the given nodes, because of a cycle. Thus we must make sure that the number of buds found is $2n + 2$.

```
⟨ Check that we've filled out the whole tree 13 ⟩ ≡
  if (buds ≠ n + n + 2) abort0("The␣input␣contains␣a␣cycle", −66);
```

This code is used in section 10.

14. The state chart. Now that we've got the tree in memory, we can emulate Alice's moves. This program gathers more information than is absolutely needed, just in case the extra data will help me psych out some structural properties.

```
⟨ Type definitions 7 ⟩ +≡
  typedef struct step_struct {
    int rank;      /* rank on entry */
    int first;     /* bud being passed, or arc's initial node */
    int second;    /* arc's final node (in the second case) */
    int match;     /* bud being matched (in the second case) */
  } step;
```

15.
```
⟨ Global variables 8 ⟩ +≡
  step chart[4 * maxcodes];     /* the state chart, of length 4n */
  int steps;     /* the current number of entries in chart */
  int stack[256];     /* buds currently unmatched */
  int stacked;     /* the number of such buds */
```

16.
```
⟨ Find and print the three conjugates of T 16 ⟩ ≡
  ⟨ Create the state chart 20 ⟩;
  ⟨ Print the tree with all buds shown 23 ⟩;
  ⟨ Print the conjugates from the state chart 22 ⟩;
```
This code is used in section 5.

17. The state chart is created from a recursive routine *createsteps*, analogous to *fillbuds*.

```
⟨ Subroutines 11 ⟩ +≡
  void branch(int, int);     /* see below */
  void budstep(int);     /* see below */
  void createsteps(int p)
  {
    register int q;
    q = inputnode[p].left;
    if (q > 0) branch(p, q);
    else budstep(−q);
    q = inputnode[p].middle;
    if (q > 0) branch(p, q);
    else budstep(−q);
    q = inputnode[p].right;
    if (q > 0) branch(p, q);
    else budstep(−q);
  }
```

18.
```
#define offset 2     /* difference between stacked and the current rank */
⟨ Subroutines 11 ⟩ +≡
  void budstep(int b)
  {     /* chart gains a bud */
    chart[steps].first = b, chart[steps].rank = stacked − offset;
    if (chart[steps].rank ≠ inputbud[b].rank) confusion("rank␣offense␣b");
    inputbud[b].stepno = steps;
    steps++, stack[stacked++] = b;
  }
```

19. ⟨ Subroutines 11 ⟩ +≡

```
void branch(int p, int q)
{   /* chart passes from one arc to its dual */
  chart[steps].first = p, chart[steps].second = q, chart[steps].rank = stacked − offset;
  if (chart[steps].rank ≠ inputnode[q].rank) confusion("rank offense q");
  chart[steps].match = stack[−−stacked];
  steps++;
  createsteps(q);
  chart[steps].first = q, chart[steps].second = p, chart[steps].rank = stacked − offset;
  if (chart[steps].rank ≠ inputnode[q].rank + 2) confusion("rank offense p");
  chart[steps].match = stack[−−stacked];
  steps++;
}
```

20. ⟨ Create the state chart 20 ⟩ ≡

```
chart[0].rank = −2, chart[0].first = 1, steps = 1;
stack[0] = 1, stacked = offset − 1;
createsteps(root);
if (stacked ≠ 2 + offset) confusion("mismatched");
if (steps ≠ 4 ∗ n) confusion("total steps");
```

This code is used in section 16.

21. Conversely, given the state chart, there's a simple recursive routine that prints a tree beginning after an unmatched bud.

Interestingly, the nodes of the tree are reported in postorder although they are encountered in preorder.

⟨ Subroutines 11 ⟩ +≡

```
void printfam(int p)
{
  register int q;
  int l, m, r;
  if (steps ≡ 4 ∗ n) steps = 0;
  q = chart[steps++].second;
  if (q ≡ 0) l = '-';
  else l = q, printfam(q), steps++;
  if (steps ≡ 4 ∗ n) steps = 0;
  q = chart[steps++].second;
  if (q ≡ 0) m = '-';
  else m = q, printfam(q), steps++;
  if (steps ≡ 4 ∗ n) steps = 0;
  q = chart[steps++].second;
  if (q ≡ 0) r = '-';
  else r = q, printfam(q), steps++;
  printf("%c%c%c%c", p, l, m, r);
}
```

22. The algorithm here is very cute, so I let the reader have the fun of deciphering it.

```
⟨ Print the conjugates from the state chart 22 ⟩ ≡
  chart[4 * n].second = sentinel, chart[4 * n].first = root;
  for (j = 1; j < 4; j++) {
    for (i = 0; i < j; i++) printf("+");
    printf(":");
    for (k = steps = inputbud[stack[j + offset − 2]].stepno + 1; chart[k].second ≡ 0; k++) ;
    printfam(chart[k].first);
    printf("\n");
    if (chart[steps].first ≠ stack[j + offset − 2]) confusion("bad␣end␣of␣cycle");
  }
```

This code is used in section 16.

23.

```
⟨ Print the tree with all buds shown 23 ⟩ ≡
  print_tree(root);
  printf("\n");
```

This code is used in section 16.

24.

```
⟨ Subroutines 11 ⟩ +≡
  void print_tree(int p)
  {    /* prints node p's subtree in preorder */
    register int i;
    for (i = 0; i < inputnode[p].rank + 8; i++) printf(".");
    printf("␣%c:", p);
    if (inputnode[p].left < 0) printf("%3d", −inputnode[p].left);
    else printf("␣␣%c", inputnode[p].left);
    if (inputnode[p].middle < 0) printf("%3d", −inputnode[p].middle);
    else printf("␣%c␣", inputnode[p].middle);
    if (inputnode[p].right < 0) printf("%3d\n", −inputnode[p].right);
    else printf("␣%c\n", inputnode[p].right);
    if (inputnode[p].left > 0) print_tree(inputnode[p].left);
    if (inputnode[p].middle > 0) print_tree(inputnode[p].middle);
    if (inputnode[p].right > 0) print_tree(inputnode[p].right);
  }
```

25. The quad-edge data structure for planar maps. Turning from trees to more complex graphs drawn in the plane, we now implement some beautiful data structures that were introduced by Leo Guibas and Jorge Stolfi in *ACM Transactions on Graphics* **4** (1985), 74–123.

The best way to understand their "quad-edge structure" is to consider a small example. The normal way to draw a planar graph with, say, vertices $\{1, 2, 3, 4\}$ and edges $\{a, b, c, d, e, f\}$ and faces $\{\mathrm{I}, \mathrm{II}, \mathrm{III}, \mathrm{IV}\}$ is to

connect the vertices by lines for the edges, and to name the faces in the enclosed regions:

(∗)

Inside a computer, however, the best way to represent the topology of this diagram is to construct a more

elaborate structure, which can be regarded as annotating the graph (∗) and embedding it in a richer graph:

Vertices (red) and faces (green) have been replaced by oriented cycles, which all travel counterclockwise, except that the outermost cycle runs clockwise. (That cycle would run the other way if we drew it on the equator of a sphere and looked at it from the south pole, while viewing the rest of the map from the north pole.)

The oriented cycles in (∗∗) have little connectors that we shall call "pips." The cycle for a vertex v of degree d has d pips, which indicate all of the edges adjacent to v, in counterclockwise order. Similarly, the cycle for a face indicates all of the edges surrounding that face, as we march counterclockwise around it.

One advantage of a representation like (∗∗) is the fact that it nicely represents also the *dual* graph, in which vertices become faces, faces become vertices, and edges "rotate" by 90°. For example, the dual of (∗)

is the planar graph

(***)

26. Notice that each of the edges $\{a, b, c, d, e, f\}$ in $(*)$ appears as a vertex in $(**)$. Every such vertex has degree 4, and it connects to four pips via lines numbered 0, 1, 2, and 3 in clockwise order. The pips on lines 0 and 2 always belong to vertex cycles; the pips on lines 1 and 3 always belong to face cycles. We could change the numbers $(0, 1, 2, 3)$ to $(2, 3, 0, 1)$, respectively, at each edge-vertex, without changing the meaning of this diagram; the actual numbering of these lines isn't important. But their cyclic ordering is crucial, and so is their evenness or oddness.

I should point out that two planar graphs are considered to be essentially the same if they are topologically equivalent when drawn on the surface of a sphere. They should represent the same decomposition of the sphere's surface into regions delimited by the given edges, in the sense that we could transform one drawing into the other, smoothly and without trickery. In particular, we could redraw $(*)$ in three equivalent ways

by choosing any of the other faces to be exterior:

Each of these graphs corresponds precisely to the vertices, edges, faces, and pips of (∗∗); and so are three variants of (∗∗)! Butleft-right reflection would give a different graph in this case, because (∗) has no symmetry.

27. Suppose there are m edges. Diagram $(**)$ can also be regarded as a *permutation* of the $4m$ pips, expressed in cycle form, namely

$$\alpha = (a_2d_2c_2b_2)(d_0e_2)(c_0e_0f_0)(a_0b_0f_2)(a_1f_1e_3d_3)(c_3d_1e_1)(b_3c_1f_3)(a_3b_1).$$

These cycles correspond to vertices (1), (2), (3), (4) and faces (I), (II), (III), (IV); for instance, '$(a_0b_0f_2)$' describes the cycle $a_0 \longrightarrow b_0 \longrightarrow f_2 \longrightarrow a_0$ for vertex 4 in $(**)$.

Guibas and Stolfi noted that the permutations α obtained from planar maps in this way have a very important property called the backup axiom: *If α takes $u_{i+1} \mapsto v_j$, where u and v are edges of the planar graph being represented and where their subscripts are treated modulo 4, then α also takes $v_{j+1} \mapsto u_i$.* For example, $a_2 \mapsto d_2$ and $d_3 \mapsto a_1$ in α. Notice that a_2 and d_2 are vertex pips, but d_3 and a_1 are face pips.

The backup axiom can be formulated in terms of permutations, using the special "rotation" permutation

$$\rho = (a_0a_1a_2a_3)(b_0b_1b_2b_3)(c_0c_1c_2c_3)(d_0d_1d_2d_3)(e_0e_1e_2e_3)(f_0f_1f_2f_3);$$

namely, it is equivalent to saying that $\alpha\rho\alpha\rho$ is the identity permutation. After moving from any pip by applying α and then ρ, we can back up to our original state by applying α and ρ again. This principle underlies the efficiency of a quad-edge structure, because we can easily move in the clockwise or counterclockwise direction around any vertex or around any face; we needn't traverse the whole cycle to find our predecessor.

Continuing our example, we have

$$\alpha\rho = (a_0b_1)(a_1f_2)(a_2d_3)(a_3b_2)(b_0f_3)(b_3c_2)(c_0e_1)(c_1f_0)(c_3d_2)(d_0e_3)(d_1e_2)(e_0f_1).$$

In general $\alpha\rho$ will always be a permutation of order 2, which takes every vertex pip into some face pip. Therefore $\alpha\rho$ consists entirely of two-cycles; it's a matching between the $2m$ vertex pips and the $2m$ face pips.

Similarly, $\rho\alpha$ always consists of two-cycles; in our case it's

$$\rho\alpha = (a_0f_1)(a_1d_2)(a_2b_1)(a_3b_0)(b_3f_2)(b_2c_1)(c_3e_0)(c_0f_3)(c_2d_1)(d_3e_2)(d_0e_1)(e_3f_0).$$

We'll have (u_iv_j) in $\rho\alpha$ if and only if $(u_{i+1}v_{j+1})$ is in $\alpha\rho$, because of the backup axiom. For example, $\rho\alpha$ contains (a_1d_2) which $\alpha\rho$ contains (a_2d_3).

The regions outside the cycles in $(**)$ all have four sides. For example, there's a region near the top right whose corner pips in counterclockwise order are (d_3, d_2, a_2, a_1). The backup axiom explains this fact. Moreover, it tells us that the pips at opposite corners, such as $\{d_3, a_2\}$ and $\{d_2, a_1\}$, are the pips matched by $\alpha\rho$ and $\rho\alpha$.

28. Thus the quad-edge data structure for a planar graph with m edges essentially consists of $4m$ pointers, which tell us how to move from one pip to another. These pointers form a permutation of the pips, where the permutation takes vertex pips into vertex pips and face pips into face pips. It also satisfies the backup axiom.

Guibas and Stolfi also observed that every planar graph without isolated vertices can be constructed by repeatedly performing a single primitive operation. This operation, called *splice*, exchanges two vertex pips and two face pips. (Well, there's also a primitive operation that initializes the entire data structure. To get things rolling, we begin with a set of m edges that define m two-vertex components; thus we have $2m$ vertices initially, each of degree 1. After the initialization, we can proceed to splice until we've got the graph we want.)

The best way to understand splicing is—guess what—to look at a small example. So let's construct the pip-permutation α above, and the planar graph above, by a sequence of splices.

We begin with the initial permutation

$$\alpha_0\beta_0\gamma_0\delta_0\varepsilon_0\varphi_0;$$

here $\alpha_0 = (a_0)(a_2)(a_1a_3)$, $\beta_0 = (b_0)(b_2)(b_1b_3)$, $\ldots$, and $\varphi_0 = (f_0)(f_2)(f_1f_3)$ each specify a two-vertex graphs K_2 disjoint from the others. The sub-permutation α_0 corresponds to the two-vertex graph whose edge is named a, and the other sub-permutations are similar.

Two edges a and b belong to the same component of a graph if and only if there's a sequence $(d_1, d_2, \ldots, d_r)$, for some r, such that $\alpha\rho^{d_1}\alpha\rho^{d_2}\ldots\alpha\rho^{d_r}$ takes $a_0 \mapsto b_0$. If the graph has c components, we can best think of it as a set of c connected graphs, each of which is drawn on the surface of a separate sphere; thus each component has its own "exterior face." According to a famous theorem of Euler, the number of vertices plus the number of faces is always equal to the number of edges plus $2c$.

A splice σ consists of applying two swap operations; in other words, σ has the form $(u_iv_j)(r_ks_l)$. Here i and j are even (hence u_i and v_j are vertex pips), while k and l are odd (hence r_k and s_l are face pips). If u_i and v_j lie in different cycles of α, then they lie in the same cycle of $\alpha\sigma$; in fact, they are obtained by pasting the cycles together in a straightforward way:

$$(u_ix_1\ldots x_p)(v_jy_1\ldots y_q)(u_iv_j) = (u_ix_1\ldots x_pv_jy_1\ldots y_q).$$

For example, if u and v are edges of different components, it's clear that u_i and v_j must lie in different cycles; and in that case the net effect is to paste two disconnected components of a graph together, with two vertices coalescing into one.

If u and v are edges of the same component, but u_i and v_j aren't in the same cycle, then we're allowed to splice them together only if u_{i+1} and v_{j+1} are pips of the same face. Otherwise we couldn't merge the vertices of u_i and v_j without making lines cross. (That's a consequence of Euler's theorem, mentioned above; we'd be drawing the component on a torus, not a sphere.)

Going the other way, suppose u_i and v_j do lie in the same cycle of α. Then the splice operation splits the graph into two parts, making two copies of the vertex whose pips included u_i and v_j; one copy gets some of the adjacent edges, the other copy gets the rest. Actually, however, we won't need to do splices of this kind; it's possible to form any desired planar graph without isolated vertices merely by pasting vertices together, decreasing the number of vertices with each splice.

Similar remarks apply to the cycles of face pips, depending on whether or not r_k and s_l belong to the same cycle of α. A swap operation between elements of different cycles always merges the cycles; a swap operation between elements of a single cycle always splits that cycle.

The value of (r_ks_l) is completely determined by the value of (u_iv_j) by the following important *splice rule*: If α takes $u_{i+1} \mapsto x_p$ and $v_{j+1} \mapsto y_q$, then $r_k = x_p$ and $s_l = y_q$. This rule is necessary so that the backup axiom is preserved; we can't paste vertices together or split them apart unless we do an exactly consistent thing with respect to the faces. And fortunately, the splice rule is also sufficient for maintaining the backup condition.

29. Armed with all this information, we're ready at last to construct the example map $(*)$ from the initial permutation $\alpha_0\beta_0\gamma_0\delta_0\varepsilon_0\varphi_0$ in a sensible way, without resorting to trial and error.

That map has both a and b attached to vertex 4, with pips a_0 and b_0 in the vertex ring for 4 in $(**)$. So we can start by splicing α_0 and β_0 together; that means $u_i = a_0$ and $v_j = b_0$. The splice rule now tells us that $\sigma = (a_0b_0)(a_3b_3)$, because $\alpha_0\beta_0$ takes $a_1 \mapsto a_3$ and $\beta_1 \mapsto b_3$. Thus we obtain

$$\alpha_1 = \alpha_0\beta_0\sigma_1 = \alpha_0\beta_0(a_0b_0)(a_3b_3) = (a_0b_0)(a_2)(b_2)(a_1b_3b_1a_3).$$

This permutation represents a path of length 2, consisting of edges a and b joined by a common vertex whose pips are a_0 and b_0. The other two vertices, which have degree 1 because they're endpoints of the path, have the respective pips a_2 and b_2. The reader is encouraged to draw the corresponding graph of vertices, edges, faces, and pips, so that these ideas become crystal clear. (This graph will be analogous to $(**)$, but it will be considerably simpler because there are only three vertices, two edges, and one face.)

Next let's join *those* two vertices together; we're allowed to do that, even though they belong to the same component, because a_3 and b_3 belong to the same cycle. The result, with $\sigma_2 = (a_2b_2)(a_1b_1)$, is

$$\alpha_2 = \alpha_1\sigma_2 = (a_0b_0)(a_2b_2)(a_1b_3)(a_3b_1),$$

representing a cycle of length 2 between the vertices (a_0b_0) and (a_2b_2). There are two faces, (a_1b_3) and (a_3b_1), either of which can be considered to be the exterior face.

In a similar way we can build up a *three*-cycle from c, d, and e: Letting $\sigma_3 = (c_2d_2)(c_1d_1)$, $\sigma_4 = (d_0e_2)(d_3e_1)$, and $\sigma_5 = (c_0e_0)(c_3e_3)$, we obtain

$$\begin{aligned}
\gamma_1 &= \gamma_0\delta_0\sigma_3 = (c_0)(c_2d_2)(d_0)(c_1c_3d_1d_3);\\
\gamma_2 &= \gamma_1\varepsilon_0\sigma_4 = (c_0)(c_2d_2)(d_0e_2)(e_0)(c_1c_3d_1e_1e_3d_3);\\
\gamma_3 &= \gamma_2\varphi\sigma_5 = (c_0e_0)(c_2d_2)(d_0e_2)(c_1e_3d_3)(c_3d_1e_1).
\end{aligned}$$

Now we can hook f to vertex (c_0e_0), using $\sigma_6 = (c_0f_0)(e_3f_3)$:

$$\gamma_4 = \gamma_3\sigma_6 = (c_0e_0f_0)(c_2d_2)(d_0e_2)(f_2)(c_1f_3f_1e_3d_3)(c_3d_1e_1).$$

The two remaining components can be joined together, merging vertices (a_2b_2) and (c_2d_2) appropriately with $\sigma_7 = (b_2d_2)(a_1c_1)$:

$$\alpha_3 = \alpha_2\gamma_4\sigma_7 = (a_2d_2c_2b_2)(a_0b_0)(c_0e_0f_0)(d_0e_2)(f_2)(a_1b_3c_1f_3f_1e_3d_3)(a_3b_1)(c_3d_1e_1).$$

And the final coup de grâce hooks (f_2) to (a_0b_0), with $\sigma_8 = (f_2a_0)(f_1b_3)$:

$$\alpha_4 = \alpha_3\sigma_8 = (a_0b_0f_2)(a_2d_2c_2b_2)(c_0e_0f_0)(d_0e_2)(a_1f_1e_3d_3)(a_3b_1)(b_3c_1f_3)(c_3d_1e_1).$$

Yes, this is indeed the permutation of $(**)$. We have

$$\alpha = \alpha_0\beta_0\gamma_0\delta_0\varepsilon_0\varphi_0\,\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6\sigma_7\sigma_8.$$

One of the main reasons I wrote this program was because I knew that a computer could do these calculations almost instantly, without making silly mistakes.

30. So let's write that code. Each pip is conveniently represented by its subscript plus four times the ASCII code of the edge name. For example, a_3 would be $(\texttt{'a'} \ll 2) + 3$, which is 391 because $\texttt{'a'} = 97$.

We maintain both α and its inverse α^- in memory, because both representations are useful. If p is a pip with $alpha[p] = q$, then $q = alphainv[p]$. (There isn't really a need for both representations, however, because the backup axiom $\alpha^- = \rho\alpha\rho$ always holds.)

```
#define pip(u,i) ((u) << 2) + (i)
#define pip_edge(p) ((p) >> 2)
#define pip_sub(p) ((p) & 3)
#define rot(p) (((p) + 1) ^ (((p) ^ ((p) + 1)) & -4))   /* ρ */
#define irot(p) (((p) - 1) ^ (((p) ^ ((p) - 1)) & -4))   /* ρ^- */
```

⟨ Global variables 8 ⟩ +≡

```
int alpha[4 * 256];   /* the permutation of pips describing the current planar map */
int alphainv[4 * 256];   /* its inverse */
int verts;   /* the current number of vertices */
```

31. We'll permute only the pips for a special *root edge* and for edges that correspond to nodes in the skew ternary tree that was input. The latter nodes are identifiable because they have left children. (They also have middle and root children. But we don't really care about the children's identities, only their existence.) The root edge is called `*`.

⟨ Create the initial permutation 31 ⟩ ≡

```
for (k = '*', verts = 0; k ≤ '~'; k++)
  if (k ≡ '*' ∨ inputnode[k].left) {
    alpha[pip(k,0)] = alphainv[pip(k,0)] = pip(k,0);
    alpha[pip(k,1)] = alphainv[pip(k,1)] = pip(k,3);
    alpha[pip(k,2)] = alphainv[pip(k,2)] = pip(k,2);
    alpha[pip(k,3)] = alphainv[pip(k,3)] = pip(k,1);
    verts += 2;
  }
if (verts ≠ 2 * (n + 1)) confusion("initial␣vertex␣count");
```

This code is used in sections 49 and 62.

32. The *splice* subroutine is given the addresses of two vertex pips that are supposed to be interchanged. It figures out the two corresponding face pips, using the splice rule mentioned above.

We do not worry about the "legality" of a splice, in the sense of preserving planarity, because we'll use *splice* only to reduce the number of vertices. Any illegal usage would cause the final number of faces to be incompatible with Euler's criterion.

(Well, there's an exception: In one place below I will splice two pips apart that are adjacent in their vertex cycle. To compensate, I'll increase *verts* by 2.)

```
⟨ Subroutines 11 ⟩ +≡
  void splice(int p, int q)
  {
    register int r, s;
    if ((p & 1) + (q & 1)) confusion("attempt to splice face pips");
    r = alphainv[p], s = alphainv[q];
    alphainv[p] = s, alphainv[q] = r;
    alpha[s] = p, alpha[r] = q;
    p = alpha[rot(p)], q = alpha[rot(q)];      /* now swap the appropriate faces */
    r = alphainv[p], s = alphainv[q];
    alphainv[p] = s, alphainv[q] = r;
    alpha[s] = p, alpha[r] = q;
    verts--;
  }
```

33. Here's a cute subroutine that displays all relevant information about the planar graph by printing α's cycles. First come the pips of the vertex cycles, then the pips of the face cycles, one line at a time.

The program also counts the number of vertices and faces, so that it can use Euler's formula to report the number of components (assuming planarity).

```
⟨ Subroutines 11 ⟩ +≡
  void print_alpha(void)
  {
    register int c, f, p, q, r, t, v;
    ⟨ Print and count the vertex cycles 34 ⟩;
    if (v ≠ verts) confusion("vertex count");
    ⟨ Print and count the face cycles 35 ⟩;
    c = (v − (n + 1) + f) ≫ 1;
    printf("(Altogether %d vertices, %d edges, %d faces, %d component%s.)\n",
        v, n + 1, f, c, c ≡ 1 ? "" : "s");
  }
```

34. The idea is to find a cycle leader (the least p whose cycle hasn't already been printed), and to print its cycle, until all cycles for even-numbered pips have been found.

```
⟨ Print and count the vertex cycles 34 ⟩ ≡
  printf("Vertices:\n");
  v = 0, p = pip('*', 0), t = 2 * (n + 1);
  while (1) {
    for ( ; alpha[p] ≤ 0 ∧ t; p += 2) {
      if (alpha[p] < 0) {      /* we've temporarily negated it, see Algorithm 1.3.3I */
        alpha[p] = −alpha[p], t−−;
      }
    }
    if (t ≡ 0) break;      /* t unprocessed pips remain */
    for (q = p, r = alpha[q]; r > 0; q = r, r = alpha[q]) {
      printf("␣%c%d", pip_edge(r), pip_sub(r));
      alpha[q] = −r;
    }
    printf("\n");
    v++;
  }
```

This code is used in section 33.

35. Exactly the same idea works for odd-numbered pips, of course.

```
⟨ Print and count the face cycles 35 ⟩ ≡
  printf("Faces:\n");
  f = 0, p = pip('*', 1), t = 2 * (n + 1);
  while (1) {
    for ( ; alpha[p] ≤ 0 ∧ t; p += 2) {
      if (alpha[p] < 0) {      /* we've temporarily negated it, see Algorithm 1.3.3I */
        alpha[p] = −alpha[p], t−−;
      }
    }
    if (t ≡ 0) break;      /* t unprocessed pips remain */
    for (q = p, r = alpha[q]; r > 0; q = r, r = alpha[q]) {
      printf("␣%c%d", pip_edge(r), pip_sub(r));
      alpha[q] = −r;
    }
    printf("\n");
    f++;
  }
```

This code is used in section 33.

36. The building blocks of planar graphs. Any connected multigraph is built up in a straightforward treelike way from so-called blocks (aka biconnected components or nonseparable graphs), attached together via so-called articulation points (aka cut vertices). We exclude the trivial cases where a block has fewer than two vertices; in other words, we exclude the empty graph, the one-vertex graph K_1, and the multigraph that consists of a single self-loop.

A nontrivial biconnected planar graph is said to be *rooted* when we place an arrow on one of its edges, converting that edge to a directed arc from u to v called the root edge. Vertex u is called the root; and we draw the graph so that the root edge lies on the path that travels counterclockwise around the exterior face. (In other words, the exterior face lies on your right, if you move from u to v.)

A rooted, nontrivial biconnected planar map (henceforth "RNBPM") is an equivalence class of rooted, nontrivial biconnected planar graphs, where two such graphs are said to be equivalent if they're topologically the same when drawn on a sphere as discussed earlier. Thus, each RNBPM can be characterized by its α permutation, except for renaming of the edges and except for adding 2 (mod 4) to the subscripts of any selected subset of the edges.

Suppose r is the root edge, and suppose the other edges of the exterior cycle are e^1, e^2, $\ldots$, e^p. We will define things so that the root vertex cycle contains the pip r_0, hence the exterior face cycle contains the pip r_3. We will also define pip numbers so that α takes $r_0 \mapsto e_2^1$, $e_0^1 \mapsto e_2^2$, $\ldots$, $e_0^p \mapsto r_2$, so that the exterior face cycle is $(e_3^p \ldots e_3^2 e_3^1 r_3)$. Exception: If $p = 0$, that cycle is of course $(r_1 r_3)$.

37. The simplest RNBPM consists of just the root edge. Otherwise we can build up any RNBPM recursively in a simple way: Removal of the root edge leaves a graph with $m \geq 1$ blocks (hence $m-1$ articulation points); consequently each of those blocks becomes an RNBPM once we identify its root edge. We choose to take the root as the first edge encountered on the exterior face of the full graph, in counterclockwise order. We also take note of the first edge that is *not* exterior in the full graph, thereby making the block "doubly rooted." Then the original RNBPM is easily reconstructed from its doubly rooted blocks.

Once again we crave an example. Our previous graph $(*)$ will be an RNBPM if we choose any edge u as a designated root edge, and if we consider the pip u_3 to be on its exterior face. But that example is too simple

to reveal the general situation; so let's consider something a bit more complex:

(†)

Here $m = 4$ and the exterior face has $p = 7$ other edges; its cycle is therefore $(r_3 d_3^2 d_3^1 c_3^3 c_3^2 c_3^1 b_3^1 a_3^1)$. Furthermore the interior face touching r is $(r_1 a_3^3 a_3^2 b_1^1 c_3^7 c_3^6 c_3^5 c_3^4 d_3^3)$. If we remove edge r, three articulation points spring up that subdivide the remaining graph into four blocks, having exterior edges identified by the letters $\{a, b, c, d\}$. Block b is just an isthmus, but the other blocks have been built up in turn from smaller constituents. Those larger blocks have been shaded in this diagram, because they may contain complicated interior structure that is invisible from the outside.

The four blocks can be regarded as RNBPMs, having the respective root edge pips a_3^1, b_3^1, c_3^1, and d_3^1. And they're also doubly rooted, because we specify nonroot exterior vertex pips a_2^2, b_0^1, c_2^4, d_2^3 that tell us how to hook them together. If α, β, γ, and δ are the permutations corresponding to those blocks, and if $\omega = (r_0)(r_2)(r_1 r_3)$ is the permutation for edge r, the permutation for the whole RNBPM is $\alpha\beta\gamma\delta\omega\,\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5$, where

$$\sigma_1 = (d_2^3 r_2)(d_3^2 r_1), \quad \sigma_2 = (c_2^4 d_2^1)(c_3^3 d_3^3), \quad \sigma_3 = (b_0^1 c_2^1)(b_3^1 c_3^7), \quad \sigma_4 = (a_2^2 b_2^1)(a_3^1 b_1^1), \quad \sigma_5 = (a_2^1 r_0)(r_3 a_3^3)$$

are the appropriate splicings.

38. Here, for handy reference, are the smallest RNBPMs and their canonical permutations:

$(r_0)(r_2)(r_3r$

39. Planar maps, conformément à Jacquard et Schaeffer. We return now to our main theme of skew ternary trees.

At the very beginning I mentioned that Del Lungo et al found an intriguing correspondence between skew ternary trees and RNBPMs. They found it after first having invented the idea of skew ternary trees, and conjecturing that the number of such trees with n nodes is precisely the number of RNBPMs with n nonroot edges.

Benjamin Jacquard and Gilles Schaeffer responded to that conjecture by finding an ingenious correspondence that is quite different from the one discovered almost simultaneously by Del Lungo et al. [See *Journal of Combinatorial Theory* **A83** (1998), 1–20.] Naturally I wondered if the two correspondences are somehow related, so I decided to implement both of them in this program.

According to their construction, an RNBPM such as (†) is represented by a skew ternary tree of the form

where A', B', C', and D' represent the doubly rooted RNBPMs of the $m = 4$ blocks that arise when edge r

is removed. Thus the chart corresponding to their representation will have the form

where A^*, B^*, C^*, and D^* represent the subtrees A', B', C', and D' in some fashion.

In this particular example B' is empty, because component b has only a single edge in (†); thus B^* is simply a "+1 step" for the bud $\overline{2}$. But the subtrees A', C', and D' are nonempty (and they might in fact be extremely complicated).

The Jacquard–Schaeffer construction also has the property that the total number of rank 0 nodes is always exactly p, the number of nonroot edges on the exterior face of the given RNBPM. Consequently the subtrees D' and C' will contain nodes d^2, c^3, and c^2 of rank 0; but A' won't contain any such nodes.

40. To complete the construction, we need to explain how to represent a doubly rooted RNBPM. Consider, for example, the skew ternary tree T that appeared in the introductory sections at the very beginning of this program: The RNBPM corresponding to T can be used to build larger RNBPMs in three different ways, because T has three nodes `A`, `B`, and `E` of rank 0.

It turns out that the "buds and charts" method discussed above provides a nice way to encode the second root. The idea is to use one of the three cyclic variants that begin at a bud of rank -1 (namely at bud 1, 2, or 4). Those trees have respectively 2, 1, and 0 nodes of rank -1, and no nodes of rank -2; so they can safely be used as the right subtree T' of a node that has rank 0.

For example, the three possibilities for T^* in this example have the following respective charts:

One way to form this is to start at the bud in question and continue creating the chart cyclically until that bud occurs again. The we delete both appearances, and replace it by matching arcs from an assumed parent node `T` to the new subtree root and back. (It follows that a subtree T' of n nodes has a chart T^* of length $4n+1$, even when $n=0$.)

Conversely, it's easy to reconstruct T from any of these shifted variants T^*, by undoing the process: First we delete the matching arcs that enclose the whole; then we replace the bud that was deleted. Finally we wind back the cycle until creating a bud with rank -2 for the first time. (That bud will be cloned from the rightmost bud of rank $+2$.)

Incidentally, one can show by induction that the number of nodes of odd rank in the skew ternary tree is equal to the number of faces in the corresponding RNBPM, minus 2, according to this construction. And the number of nodes of even rank is the number of nonroot vertices.

41. Our principal goal is to take a given skew ternary tree and to construct the corresponding RNBPM, but computing the quad-tree permutation of that planar map. The tree will be given in chart form. I won't be stingy with memory, so I'll keep a stack of the various charts that arise during the recursion.

A tree with n nodes will produce an RNBPM with n edges in addition to the root edge.

```
⟨ Global variables 8 ⟩ +≡
  step chartstack[maxcodes][4*maxcodes];      /* (only the first and second fields of these entries are used) */
  step tmpchart[4 * maxcodes];
  int stk[maxcodes * maxcodes];      /* stack of subtrees waiting to be processed */
  int curbud;      /* the bud whose tree is being mapped (see below) */
```

42. The *rnbpm_js* routine constructs the Jacquard–Schaeffer RNBPM for *chartstack*[*s*] with root edge *r*. A third parameter, *h*, tells the current height of the auxiliary stack *stk*.

The value of *stk*[*h*] is also supposed to identify the root of the skew ternary tree whose chart is in *chartstack*[*s*]. (The name of the root doesn't appear in the chart when the tree has only one node, hence we need this extra contextual information.)

```
⟨ Subroutines 11 ⟩ +≡
  void rnbpm_js(int s, int r, int h)
  {
    register int i, j, k, l, m, p, q, t, tt, apip, steps;
    ⟨ Determine the number m of initial rank 0 nodes, and stack them 43 ⟩;
    apip = pip(r, 2);      /* pip for attaching blocks */
    while (m) {
      m--, t = stk[h + m];
      ⟨ Copy the tree T underlying the next T* to chartstack[s + 1] 44 ⟩;
      if (m ∧ l ≥ 0 ∧ (chartstack[s][steps].first ≠ t ∨ chartstack[s][steps].second ≠ stk[h + m − 1]))
        confusion("arc␣bracketing");
      steps++;
      if (l < 0) ⟨ Handle the case of an empty tree 45 ⟩
      else ⟨ Build the RNBPM for chartstack[s + 1] 46 ⟩;
      ⟨ Splice the new RNBPM to the previous fragment 47 ⟩;
    }
    ⟨ Splice everything into a cycle 48 ⟩;
  }
```

43. ⟨ Determine the number m of initial rank 0 nodes, and stack them 43 ⟩ ≡

```
if (chartstack[s][0].second) confusion("no␣root␣bud");
for (m = 1, steps = 1; ; m++) {
  if (chartstack[s][steps++].second) confusion("non␣skew");
  stk[h + m] = chartstack[s][steps++].second;
  if (stk[h + m] ≡ 0) break;
}
```

This code is used in section 42.

44. At this point we're poised to look at the steps of T^*, where T is the subtree that corresponds to edge $t = stk[h + m]$. If T^* is the trivial one-edge RNBPM, we set $l = -1$; otherwise we set l to the number of nodes in T^* that have rank 0 (which is also the number of buds that have rank -1).

In the second case, the subchart T^* is easily identified because it begins with a downward step from t to tt and ends with a downward step from tt to t. (These downward steps occur first from rank 1 down to rank 0, then from rank 3 down to rank 2; so they are reminiscent of the German text for quoted text, which begins with „ and ends with “!) We delete those steps and shift the others cyclically backward, in order to deduce T from T^* as explained above.

⟨ Copy the tree T underlying the next T^* to $chartstack[s + 1]$ 44 ⟩ ≡

```
if (chartstack[s][steps].second ≡ 0) l = -1;
else {
  tt = chartstack[s][steps].second;
  if (chartstack[s][steps++].first ≠ t) confusion("wrong␣parent");
  tmpchart[0].first = tmpchart[0].second = 0;      /* dummy bud */
  for (j = 1, q = l = 0; chartstack[s][steps].second ≠ t; steps++, j++) {
    tmpchart[j] = chartstack[s][steps];
    if (q ≡ 2) k = j;      /* remember the location of the last bud with rank 2 */
    else if (q ≡ -1) l++;      /* count the buds of rank -1 */
    if (tmpchart[j].second) q--; else q++;      /* q is the rank */
  }
  if (chartstack[s][steps].first ≠ tt) confusion("right␣bracket");
  for (i = k; i < j; i++) chartstack[s + 1][i - k] = tmpchart[i];
  for (i = 0; i < k; i++) chartstack[s + 1][i + j - k] = tmpchart[i];
}
steps++;
```

This code is used in section 42.

45. The RNBPM for an empty tree is simply the unadorned root edge r. We've initialized that edge already (although I could have initialized it here, instead).

⟨ Handle the case of an empty tree 45 ⟩ ≡

```
p = pip(t, 1);
```

This code is used in section 42.

46. There's a better way to do this step, because we can identify the pip p directly while copying the chart. But I didn't have time to stop and figure it out.

⟨ Build the RNBPM for $chartstack[s+1]$ 46 ⟩ ≡

```
{
  stk[h+m] = (chartstack[s+1][2].second ? chartstack[s+1][2].first : chartstack[s+1][3].second ?
      chartstack[s+1][3].first : tt);
  rnbpm_js(s+1, t, h+m);
  for (p = alphainv[pip(t,3)]; l; l--) p = alphainv[p];
}
```

This code is used in section 42.

47. ⟨ Splice the new RNBPM to the previous fragment 47 ⟩ ≡

```
splice(irot(p), apip);
apip = pip(t, 2);
```

This code is used in section 42.

48. ⟨ Splice everything into a cycle 48 ⟩ ≡

```
splice(pip(r, 0), apip);
```

This code is used in section 42.

49. Okay, *rnbpm_js* is finished. Here's how we apply it to each of the four skew ternary trees of interest.

⟨ Find and print the corresponding planar maps 49 ⟩ ≡

```
for (j = 0; j < 4; j++) {
  printf("---␣JS␣map␣for␣T");
  for (i = 0; i < j; i++) printf("+");
  printf("␣---\n");
  ⟨ Create the initial permutation 31 ⟩;
  for (i = inputbud[stack[j + offset − 2]].stepno, k = 0; i < 4 * n; i++, k++) chartstack[0][k] = chart[i];
  for (i = 0; i < inputbud[stack[j + offset − 2]].stepno; i++, k++) chartstack[0][k] = chart[i];
  stk[0] = inputbud[stack[j + offset − 2]].parent;
  rnbpm_js(0, '*', 0);
  print_alpha( );
}
```

See also section 62.

This code is used in section 5.

50. Unfortunately, I must report serious disappointment with this correspondence between RNBPMs and skew ternary trees, because its "T^* method" of keeping l exterior nonroot edges of a sub-RNBPM in the larger RNBPM actually keeps the *last* l such edges, not the edges that follow the root! Therefore the correspondence between nodes and edges is extremely weird, and it doesn't have any apparent significance for understanding the graph structure.

For example, the tree that corresponds to (†) turns out to be quite crazy:

The nodes of rank zero are a^1, b^1, c^1, d^1, d^3, c^5, c^6, c^7! They're equinumerous with the nonroot edges of (†), but that's almost the only good thing we can say about them.

The problem appears to be unfixable, because the rank 0 nodes in A^* are widely separated from node a^1.

51. Planar maps, conformemente a Del Lungo et al. The paper by Del Lungo, Del Ristoro, and Penaud presented a completely different way to associate RNBPMs with skew ternary trees, based on a completely different recursive decomposition. We shall call it the DDP correspondence.

Instead of building an RNBPM from m other doubly labeled RNBPMs, as in (†), the DDP correspondence relies on an interesting *binary* operation '$\overset{c}{\bowtie}$', which forms an RNBPM from just two others, S and T, where

S is doubly rooted but T is just singly rooted. The following picture illustrates this operation:

Here S has three edges a^1, a^2, a^3 on its exterior face, besides its root edge; and edge a^2 is the second root (distinguishable by the fact that the first root edge has no name). Similarly, T has four exterior edges b^1, b^2, b^3, b^4, and it is singly rooted. If the root edge of T runs from vertex u to vertex v, and if S's main root points to vertex s while its second root points to vertex w, the operation attaches the two RNBPMs by (i) making u and w coincide; (ii) erasing T's root edge, and (iii) introducing a new edge c from s to v.

The smallest cases of $\overset{c}{\bowtie}$ need special care: If T consists simply of its root edge, we simply add a new edge c from s to w. If S consists simply of its root edge, we consider that it is doubly rooted with the second root the reverse of the first. In the latter case the net effect is to take T and split its root into two pieces, the second of which is c.

The green shading in (‡), as in (†), indicates that complicated structure might exist within the interiors of S and T. Such structure is, however, irrelevant as we continue to build larger structures. Notice that when S isn't simply a root edge, one face of T gains one or more edges when T's root edge is removed; but T's interior structure doesn't "leak out."

52. This construction is equivalent to making three splices in the the quad-edge permutations that correspond to S and T. Let's assume that S's root edge is called r, so that its exterior face in this example is the cycle $(r_3a_3^3a_3^2a_3^1)$. The auxiliary root edge is a^2; hence w is the vertex cycle that contains a_0^2. (If S were simply the root edge r, its exterior face would be (r_3r_1), and w would be (r_2).) We may also assume that T's root edge is called c; hence its exterior face cycle is $(c_3b_3^4b_3^3b_3^2b_3^1)$, $u = (c_2 \dots)$, and $v = (c_0 \dots)$.

Step (i) of the operation corresponds to splicing with $(c_2a_2^3)$; this attaches S to T. Step (ii) then corresponds to splicing with (c_2x), where $x = c_2\alpha$ is the first pip counterclockwise from c in the cycle for $u = w$.

This leaves edge c "dangling":

(i) ; (ii)

Finally, a splice with $(c_2 a_2^1)$ produces $S \overset{c}{\bowtie} T$.

53. The canonical permutation representations are now different, because edge labels are assigned in a different order. Here, again for handy reference, is the new list for all cases with at most three nonroot

edges—showing also the skew ternary trees that we are about to construct for them:

$(r_0)(r_2)(r_3r$

54. The DDP correspondence between an RNBPM T and its skew ternary tree $\widehat{T}$ is designed to have two key properties, both of which can be observed in the examples just shown: (1) The nodes of rank 0, from top to bottom, correspond to the nonroot edges that touch the root vertex, in counterclockwise order. (2) The buds of rank 0 that follow the last node of rank 0, in preorder, correspond to the nonroot edges of the exterior cycle, in counterclockwise order.

And the recursive rule to define the correspondence is amazingly simple: The simplest RNBPM (which has nothing but a root edge) corresponds to the empty tree (which has two buds, both of rank 0, only one of which is actually considered to be meaningful in property (2)). Otherwise the tree corresponding to $S \overset{c}{\bowtie} T$ is obtained by (i) finding $\widehat{S}$ and $\widehat{T}$; (ii) computing $\widehat{T}^+$ using the cyclic rotation operation on skew ternary trees at the beginning of this program; (iii) replacing the rank 0 bud of $\widehat{S}$ that corresponds to S's second root by a new node c whose right child is $\widehat{T}^+$.

To invert this rule, notice that we can recover S, T, and c from the resulting skew ternary tree, because c is the last node of rank 0 (in preorder); then if c's right subtree is R, we have $R = \widehat{T}^+$, hence $\widehat{T} = R^-$; and $\widehat{S}$ is obtained by removing c and R. From $\widehat{S}$ and $\widehat{T}$ we know S and T, recursively. And the second root in S corresponds to the parent node of c.

55. Here then is a subroutine that implements what was just said. The *rnbpm_ddp* routine constructs the RNBPM for *chartstack*[s] with root edge r. The chart is followed by a special step for which the *second* field is *sentinel* and the *first* field is the name of the root.

```
⟨ Subroutines 11 ⟩ +≡
  void rnbpm_ddp(int s, int r)
  {
    register int c, i, j, jj, k, p, q, rr, t, steps, parent;
    ⟨ Find the last node, c, in preorder that has rank 0, and its parent p 56 ⟩;
    ⟨ Copy R⁻ into chartstack[s + 1], where R is the right subtree of c 57 ⟩;
    ⟨ Recursively build the RNBPM for T 58 ⟩;
    ⟨ Copy the rest of the tree into chartstack[s + 1] 59 ⟩;
    ⟨ Recursively build the RNBPM for S 60 ⟩;
    ⟨ Hook everything together with three magic splices 61 ⟩;
  }
```

56. The steps of the chart follow preorder.

```
⟨ Find the last node, c, in preorder that has rank 0, and its parent p 56 ⟩ ≡
  if (chartstack[s][0].second) confusion("no␣root␣bud");
  for (steps = 1, q = −1; chartstack[s][steps].second ≠ sentinel; steps++) {
    if (q ≡ −1) j = steps;
    if (chartstack[s][steps].second ≡ 0) q++; else q−−;
  }
  if (q ≠ 2) confusion("bad␣rank␣at␣end");
  c = chartstack[s][j − 1].second;
  if (c ≡ 0) {      /* c is the root of the charted tree */
    if (j ≠ 1) confusion("parentless␣rank␣-1␣bud␣not␣at␣beginning");
    c = chartstack[s][steps].first, p = 0;
  } else p = chartstack[s][j − 1].first;
  if (chartstack[s][j + 1].second) confusion("not␣the␣last␣zero");
```

This code is used in section 55.

57. If c's right child is just a bud, the subtree R is empty and it corresponds to the empty tree. Otherwise R is bracketed in the chart by arcs from c to its root node and back again, just as the subtree T^* was bracketed in the procedure *rnbpm_js*. In this case the copying task is simpler than it was before, because we needn't count zeros.

```
⟨ Copy R⁻ into chartstack[s+1], where R is the right subtree of c 57 ⟩ ≡
  jj = j − 1, steps = j + 2;
  rr = chartstack[s][steps].second;
  if (rr) {      /* rr is the root of a nonempty subtree R */
    if (chartstack[s][steps++].first ≠ c) confusion("wrong␣parent");
    tmpchart[0].first = tmpchart[0].second = 0;      /* dummy bud */
    for (j = 1, q = 0; chartstack[s][steps].second ≠ c; steps++, j++) {
      tmpchart[j] = chartstack[s][steps];
      if (q ≡ 2) k = j;      /* remember the location of the last bud with rank 2 */
      if (tmpchart[j].second) q−−; else q++;      /* q is the rank */
    }
    if (chartstack[s][steps].first ≠ rr) confusion("right␣bracket");
    for (i = k; i < j; i++) chartstack[s+1][i−k] = tmpchart[i];      /* shift to R⁻ */
    for (i = 0; i < k; i++) chartstack[s+1][i+j−k] = tmpchart[i];
    chartstack[s+1][j].second = sentinel;
    chartstack[s+1][j].first = (chartstack[s+1][2].second ? chartstack[s+1][2].first :
        chartstack[s+1][3].second ? chartstack[s+1][3].first : rr);
  }
  steps++;
```

This code is used in section 55.

58.

```
⟨ Recursively build the RNBPM for T 58 ⟩ ≡
  if (rr) rnbpm_ddp(s + 1, c);
```

This code is used in section 55.

59. If c is the root of the tree, then p is zero, subtree $\widehat{S}$ is empty, and nothing needs to be done. Otherwise the tree that corresponds to $\widehat{S}$ is obtained by simply leaving out c and R. In the latter case, jj points to the arc from p to c, and *steps* points to the arc from c back to p.

```
⟨ Copy the rest of the tree into chartstack[s+1] 59 ⟩ ≡
  if (p) {
    for (i = 0; i < jj; i++) chartstack[s+1][i] = chartstack[s][i];
    chartstack[s+1][i].first = chartstack[s+1][i].second = 0, i++;      /* bud for c */
    for (steps++; ; steps++, i++) {
      chartstack[s+1][i] = chartstack[s][steps];
      if (chartstack[s+1][i].second ≡ sentinel) break;
    }
  }
```

This code is used in section 55.

60.

```
⟨ Recursively build the RNBPM for S 60 ⟩ ≡
  if (p) rnbpm_ddp(s + 1, r);
```

This code is used in section 55.

61. Finally we obey the three-step splicing protocol for $\overset{c}{\bowtie}$ that was described above. Some tricky maneuvering is necessary in the degenerate cases.

```
⟨ Hook everything together with three magic splices 61 ⟩ ≡
  if (rr ≡ 0) {      /* T̂ is empty */
    if (p) splice(pip(c,0), alpha[pip(p,0)]);
    else splice(pip(c,0), pip(r,2));
  } else {
    if (p) splice(pip(c,2), alpha[pip(p,0)]);
    else splice(pip(c,2), pip(r,2));
    splice(pip(c,2), alpha[pip(c,2)]);
    verts += 2;      /* because we spliced two pips from the same vertex */
  }
  splice(pip(c,2), irot(alphainv[pip(r,3)]));
```

This code is used in section 55.

62. Okay, *rnbpm_ddp* is finished. Here's how we apply it to each of the four skew ternary trees of interest.

```
⟨ Find and print the corresponding planar maps 49 ⟩ +≡
  for (j = 0; j < 4; j++) {
    printf("---␣DDP␣map␣for␣T");
    for (i = 0; i < j; i++) printf("+");
    printf("␣---\n");
    ⟨ Create the initial permutation 31 ⟩;
    for (i = inputbud[stack[j + offset − 2]].stepno, k = 0; i < 4 ∗ n; i++, k++) chartstack[0][k] = chart[i];
    for (i = 0; i < inputbud[stack[j + offset − 2]].stepno; i++, k++) chartstack[0][k] = chart[i];
    chartstack[0][k].first = inputbud[stack[j + offset − 2]].parent;
    chartstack[0][k].second = sentinel;
    rnbpm_ddp(0, '*');
    print_alpha( );
  }
```

63. Here, for instance, is the RNBPM that corresponds to the example skew ternary tree T at the very

beginning of this program, according to the DDP correspondence:

The α permutation is

$$(e_2b_2a_2r_0)(f_0d_0e_0r_2)(c_0d_2f_2a_0)(c_2b_0)(a_1f_1r_1)(e_3r_3)(b_1c_1a_3)(e_1d_3c_3b_3)(f_3d_1).$$

64. One can show that the number of nodes of even rank is the number of interior faces in the DDP correspondence; the number of nodes of odd rank is the number of vertices, minus 2.

(The actual rank of each vertex node and each face node can in fact be "read off" from the RNBPM, if it is examined in an appropriate depth-first search order, because of results mentioned below.)

65. Indeed, this program led to a huge surprise, because much, much more is true. In every case that I had examined by hand, in my first explorations of the DDP correspondence, I noticed that *the four* RNBPM*s obtained from* T, T^+, T^{++}, *and* T^{+++} *are dual graphs!* I wrote this program in order to check that conjecture on examples that were too large to study reliably by hand; and I found that the conjecture was always verified, even when I looked at large random instances.

More precisely, if we call the four pip permutations α_0, α_1, α_2, and α_3, when the DDP correspondence has been applied to four conjugate skew ternary trees, I found that α_k was equal to $\alpha_0\hat{\rho}^k$ in every case that I computed. Here $\hat{\rho}$ is the permutation $(r_1r_3)\rho(r_1r_3)$; it's like ρ except that it *decreases* subscripts of r while increasing the subscripts of all the other edges (modulo 4). For example, the alpha permutations for T^+, T^{++}, and T^{+++} are the following "clones" of the alpha permutation for T given above:

$$(e_1b_1a_1r_1)(f_3d_3e_3r_3)(c_3d_1f_1a_3)(c_1b_3)(a_0f_0r_2)(e_2r_0)(b_0c_0a_2)(e_0d_2c_2b_2)(f_2d_0)$$
$$(e_0b_0a_0r_2)(f_2d_2e_2r_0)(c_2d_0f_0a_2)(c_0b_2)(a_3f_3r_3)(e_1r_1)(b_3c_3a_1)(e_3d_1c_1b_1)(f_1d_3)$$
$$(e_3b_3a_3r_3)(f_1d_1e_1r_1)(c_1d_3f_3a_1)(c_3b_1)(a_2f_2r_0)(e_0r_2)(b_2c_2a_0)(e_2d_0c_0b_0)(f_0d_2)$$

Evidence for that conjecture was overwhelming, so I asked several experts for help. And it turned out, by extraordinary luck, that I had chosen exactly the right person to ask, namely Gilles Schaeffer. He told me that, after writing the paper with Jacquard that was cited above, he continued to do research about connections between trees and planar maps. The result was his Ph.D. dissertation, *Conjugation d'arbres et cartes combinatoires aléatores* (l'Université Bordeaux I, 1998); and when I downloaded that thesis I found it to be an amazingly rich compendium of deep new results, covering many topics in addition to RNBPMs (most of which he chose not to publish elsewhere). In particular, on pages 65–67 he sketched a $(2n+2)$-to-4 correspondence between ternary trees and RNBPMs, which amounts to an independent discovery of the DDP correspondence, although he did not explicitly mention any connection with skew ternary trees.

Schaeffer's remarkable construction on those three pages explains everything: It can be used to show that *Alice can actually construct the corresponding planar graph "online" as she is walking around the tree!*

Namely, we can add four downward steps to the state chart, and assign new labels to the steps, as follows

(illustrated for the skew ternary tree in the introduction):

An upward step is labeled x_i, where i is the rank (modulo 4) at the beginning of the step and x is the name of the node attached to this bud. A downward step is labeled y_j, where j is the rank plus two (modulo 4) at the beginning of the step and y is the name of the node to which we are descending. The final four steps are labeled r_2, r_3, r_0, and r_1. Thus we've assigned $4n + 4$ labels, one for each pip in the permutation; and the pips have been matched up in pairs $\{x_i, y_j\}$. Every such pair has the meaning that α maps $x_i \mapsto y_{j-1}$ and $y_j \mapsto x_{i-1}$, where the subscripts are treated modulo 4.

The validity of this rule is readily proved by induction, because it is an *extremely* strong induction hypothesis. And my conjecture about the way duality affects the permutations α_k is an immediate consequence.

66. Hey, we're finished—except for a final routine, which we fondly hope will never be invoked.

```
⟨ Assertion failure subroutine 66 ⟩ ≡
  void confusion(char *id)
  {     /* an assertion has failed */
    fprintf(stderr, "This␣can't␣happen␣(%s)!\n", id);
    exit(-666);
  }
```

This code is used in section 5.

67. Index.

SKEW-TERNARY-CALC