1*. Intro. This program determines whether a given free tree, S, is isomorphic to a subtree of another free tree, T, using an algorithm published by David W. Matula [*Annals of Discrete Mathematics* **2** (1978), 91–106]. His algorithm is quite efficient; indeed, it runs even faster than he thought it did! If S has m nodes and T has n nodes, the running time is at worst proportional to mn times the square root of the maximum inner-degree of any node in S, where the inner degree of a node is the number of its nonleaf neighbors.

In this version, tree T is specified in a text file, using the "rectree format" defined in MATULA-BIG. Tree S is obtained from T by deleting d leaves, one at a time, where each deletion is chosen uniformly from among the existing leaves. (Thus S is definitely a subtree; I just want to see how long it takes for this algorithm to find it.) The value of d is given on the command line.

I hacked this code from MATULA-BIG.

2*. The program is instrumented to record the number of mems, namely the number of times it accesses an octabyte of memory. (Most of the memory accesses are actually to tetrabytes (**int**s), because this program rarely deals with two tetrabytes that are known to be part of the same octabyte.)

```
#define maxn 2000
#define o mems++   /* count one mem */
#define oo mems += 2   /* count two mems */
#define ooo mems += 3   /* count three mems */
#define oooo mems += 4   /* count four mems */
#define suboverhead 10   /* mems charged per subroutine call */
#define decode(c) ((c) ≥ '0' ∧ (c) ≤ '9' ? (c) − '0' : (c) ≥ 'a' ∧ (c) ≤ 'z' ? (c) − 'a' + 10 :
          (c) ≥ 'A' ∧ (c) ≤ 'Z' ? (c) − 'A' + 36 : −1)
#define encode(p) ((p) < 10 ? (p) + '0' : (p) < 36 ? (p) − 10 + 'a' : (p) < 62 ? (p) − 36 + 'A' : '?')
#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h"
  int del, seed;   /* command-line parameters */
  ⟨Type definitions 5*⟩;
  ⟨Global variables 6*⟩;
  unsigned long long mems;   /* memory references */
  unsigned long long imems;   /* mems during the input phase */
  ⟨Subroutines 7*⟩;
  main(int argc, char *argv[])
  {
    register int d, e, g, i, j, k, m, n, p, q, r, s, v, z;
    ⟨Process the command line 3*⟩;
    imems = mems, mems = 0;
    if (m > n) fprintf(stderr, "There's␣no␣solution,␣because␣m>n!\n");
    else {
      ⟨Solve the problem 28⟩;
      ⟨Report the solution 47⟩;
    }
    fprintf(stderr, "Altogether␣%lld+%lld␣mems.\n", imems, mems);
  }
```

3*. ⟨Process the command line 3*⟩ ≡

```
if (argc ≠ 4 ∨ sscanf(argv[2], "%d", &del) ≠ 1 ∨ sscanf(argv[3], "%d", &seed) ≠ 1) {
  fprintf(stderr, "Usage: %s T_rectree deletions seed\n", argv[0]);
  exit(−1);
}
gb_init_rand(seed);
⟨Input the tree S 14*⟩;
⟨Input the tree T 22*⟩;
```

This code is used in section 2*.

4*. Recursive tree format. The definition of a free tree in rectree format has many redundancy checks to keep you honest; so it is probably best to let a computer prepare it.

The opening lines of that file may contain optional comments, indicated by a '`%`' sign at the very left. Then comes the "main line," which defines the overall context. The main line can have one of three forms:

a) '`T`n`_0.`' This means an n-node tree, beginning with node 0.
b) '`T`m`_0,T`m`_`m`.`' This means a $2m$-node tree, formed from the m-node trees `T`m`_0` and `T`m`_`m, with their nodes made adjacent.
c) '`2T`m`_0.`' This means a $2m$-node tree, formed from two identical copies of the m-node tree `T`m`_0`, with their nodes made adjacent.

Options (b) and (c) are created by the Mathematica program `randomfreetree.m` when it's creating a free tree with two centroids. (But the trees in rectree format are allowed to have their centroids anywhere.)

All lines after the main line contain definitions of the subtrees of size 3 or more, *always proceeding in strictly last-in-first-out order.* Subtree names have the form `T`k`_`o, where k is the number of nodes and o is the offset (the number of the root node). Each definition of a k-node subtree appears on a separate line, beginning with the subtree name immediately followed by '`=`', and a sum of terms that collectively define subtrees totalling $k-1$ nodes (and followed by '`.`'). Each subtree name in these terms is preceded by an integer j, meaning that j copies are to appear. The offset after a term 'j`T`k`_`o' should therefore by $o+jk$.

5*. Data structures for the trees. A **node** record is allocated for each node of a tree. It has four fields: *child* (the index of its most recent child, if any), *sib* (the index of its parent's previous child, if any), *deg* (the number of neighbors), and *arc* (the number of the arc to its parent). The *deg* and *arc* fields aren't actually used for S, but we need them for T. Reference to the *deg* and *arc* fields in the same node counts as only one mem.

```
⟨Type definitions 5*⟩ ≡
  typedef struct node_struct {
    int child;   /* who is my first child, if any? */
    int sib;   /* who is the next child of my parent, if any? */
    int deg;   /* how many neighbors do I have, including my parent (if any)? */
    int arc;   /* which arc corresponds to the link from me to my parent? */
  } node;
```

This code is used in section 2*.

6*. ⟨Global variables 6*⟩ ≡

```
  node snode[maxn + 1];   /* the m nodes of S, and one more */
  node tnode[maxn + 1];   /* the n nodes of T, and one more */
```

See also sections 27, 33, 41, 46, 50, and 52*.

This code is used in section 2*.

7*. Here's a subroutine that reads a rectree file and puts the associated tree into the *tnode* array.

```
#define bufsize 1 ≪ 12
⟨Subroutines 7*⟩ ≡
  FILE *infile;
  char buf[bufsize];
  int read_rectree(char *filename)
  {
    register i, j, k, p, q, r, s, typ, stack, n, size, off, rightoff, rep;
    mems += suboverhead;
    infile = fopen(filename, "r");
    if (¬infile) {
      fprintf(stderr, "I␣can't␣open␣'%s'␣for␣reading!\n", filename);
      exit(−99);
    }
    while (1) {
      if (¬fgets(buf, bufsize, infile)) {
        fprintf(stderr, "Rectree␣file␣'%s'␣ended␣before␣the␣main␣line\n", filename);
        exit(−98);
      }
      if (o, buf[0] ≠ '%') break;
    }
    ⟨Process the main line 8*⟩;
    while (stack ≥ 0) ⟨Process the top subtree definition on the stack 10*⟩;
    ⟨Bring on the clones 12*⟩;
    ⟨Adjust the tree if bicentroidal 13*⟩;
    return o, tnode[0].deg;
  }
```

See also sections 16*, 17*, 21*, 23, and 29.

This code is used in section 2*.

8* While the tree is being installed, we use the *child* field to link items in the stack of nodes to be finished. We also use the *deg* field to record the subtree size, and the *arc* field to record the number of clones that should be made.

⟨Process the main line 8*⟩ ≡

```
if (buf[0] ≡ '2') o, typ = tnode[0].arc = 2, p = 1; else o, typ = p = tnode[0].arc = 0;
⟨Scan a subtree name 9*⟩;
if (off) {
  fprintf(stderr, "The␣main␣subtree␣must␣start␣at␣0!\n...%s", buf + q);
  exit(−104);
}
oo, n = tnode[0].deg = size, tnode[0].child = −1, tnode[0].sib = 0, stack = 0;
if (typ ≡ 2) n += n;
else if (o, buf[p] ≡ ',') {
  typ = 1, p++;
  ⟨Scan a subtree name 9*⟩;
  if (off ≠ n) {
    fprintf(stderr, "The␣second␣main␣subtree␣should␣start␣at␣%d!\n...%s", off, buf + q);
    exit(−105);
  }
  o, tnode[0].sib = n, stack = n, n += size;
}
if (n > maxn) {
  fprintf(stderr, "Tree␣too␣big,␣because␣maxn=%d!\n...%s", maxn, buf + q);
  exit(−102);
}
for (k = 1; k ≤ n; k++) oo, tnode[k].child = tnode[k].sib = tnode[k].deg = tnode[k].arc = 0;
if (typ ≡ 1) o, tnode[stack].deg = n − stack;    /* tnode[stack].child = 0 */
if (o, buf[p] ≠ '.') {
  fprintf(stderr, "The␣main␣line␣didn't␣end␣with␣`.'!\n%s", buf);
  exit(−106);
}
```

This code is used in section 7*.

9* ⟨Scan a subtree name 9*⟩ ≡

```
if (o, buf[p] ≠ 'T') {
  fprintf(stderr, "Subtree␣name␣doesn't␣start␣with␣T!\n...%s", buf + p);
  exit(−100);
}
for (q = p++, size = 0; o, (buf[p] ≥ '0' ∧ buf[p] ≤ '9'); p++) size = 10 * size + buf[p] − '0';
if (size ≡ 0) {
  fprintf(stderr, "Subtree␣size␣is␣missing␣or␣zero!\n...%s", buf + q);
  exit(−101);
}
if (buf[p++] ≠ '_') {
  fprintf(stderr, "Subtree␣name␣missing␣`_'!\n...%s", buf + q);
  exit(−103);
}
for (off = 0; o, (buf[p] ≥ '0' ∧ buf[p] ≤ '9'); p++) off = 10 * off + buf[p] − '0';
```

This code is used in sections 8*, 10*, and 11*.

10*. ⟨Process the top subtree definition on the stack 10*⟩ ≡

```
{
  oo, k = stack, stack = tnode[k].child, s = tnode[k].deg;
  o, tnode[k].child = (s ≥ 2 ? k + 1 : 0);
  if (s > 2) {
    if (¬fgets(buf, bufsize, infile)) {
      fprintf(stderr, "Rectree file `%s' ended before defining T%d_%d!\n", filename, s, k);
      exit(−107);
    }
    p = 0; ⟨Scan a subtree name 9*⟩;
    if (size ≠ s ∨ off ≠ k) {
      fprintf(stderr, "Rectree file `%s' doesn't define T%d_%d!\n %s", filename, s, k, buf);
      exit(−108);
    }
    if (o, buf[p++] ≠ '=') {
      fprintf(stderr, "Missing `=' in definition of T%d_%d!\n %s", s, k, buf);
      exit(−109);
    }
    rightoff = k + 1;
    ⟨Define the subtrees of node k 11*⟩;
    if (buf[p] ≠ '.') {
      fprintf(stderr, "Missing `.' after definition of T%d_%d!\n %s", s, k, buf);
      exit(−112);
    }
    if (rightoff ≠ k + s) {
      fprintf(stderr, "The definition of T%d_%d has %d nodes!\n %s", s, k, rightoff − k, buf);
      exit(−113);
    }
  }
}
```

This code is used in section 7*.

11*. ⟨Define the subtrees of node k 11*⟩ ≡

```
while (o, buf[p] ≡ '+') {
  for (q = p++, rep = 0; o, (buf[p] ≥ '0' ∧ buf[p] ≤ '9'); p++) rep = 10 ∗ rep + buf[p] − '0';
  if (rep ≡ 0) {
    fprintf(stderr, "Replication count missing or zero!\n...%s", buf + q);
    exit(−110);
  }
  ⟨Scan a subtree name 9*⟩;
  if (off ≠ rightoff) {
    fprintf(stderr, "That subtree should start at %d!\n...%s", rightoff, buf + q);
    exit(−111);
  }
  oo, j = rightoff, tnode[j].deg = size, tnode[j].child = stack;
  if (rep > 1) tnode[j].arc = rep;     /∗ that mem was already charged ∗/
  stack = j;
  rightoff += rep ∗ size;
  if (buf[p] ≡ '+') tnode[j].sib = rightoff;
}
```

This code is used in section 10*.

12* At this point the entire tree is in place, except that no cloning has yet been done to copy the subtrees that are supposed to be repeated.

Clones can appear inside of clones. But everything is easily patched up, if we look at the tree from bottom up when finding work to do, then clone from top down. (Kind of cute.) And we know that the total work will take linear time, because nothing is done twice.

⟨ Bring on the clones 12* ⟩ ≡
```
for (p = n − 1; p ≥ 0; p−−)
  if (o, tnode[p].arc) {
    s = tnode[p].deg, j = s ∗ tnode[p].arc;
    o, tnode[p].arc = 0;    /∗ erase tracks ∗/
    oo, i = tnode[p].sib, tnode[p].sib = p + s;    /∗ the clone will be a sibling ∗/
    for (k = p + s; k < p + j; k++) {
      o, q = tnode[k − s].child, r = tnode[k − s].sib;
      if (q) o, tnode[k].child = q + s;
      if (r) o, tnode[k].sib = r + s;
    }
    o, tnode[k − s].sib = i;    /∗ the rightmost sibling is original sibling of p ∗/
  }
```
This code is used in section 7*.

13* ⟨ Adjust the tree if bicentroidal 13* ⟩ ≡
```
if (typ) {
  o, p = tnode[0].sib;    /∗ sibling of the root will become its child ∗/
  oo, tnode[p].sib = tnode[0].child, tnode[0].child = p, tnode[0].sib = 0;
  o, tnode[0].deg = n;
}
```
This code is used in section 7*.

14* To get tree S, we start by constructing tree T, so that we can chop some of its leaves away.

⟨ Input the tree S 14* ⟩ ≡
```
n = read_rectree(argv[1]);
if (n ≤ del + 2) {
  fprintf(stderr, "I␣don't␣want␣to␣delete␣%d␣nodes␣from␣a␣tree␣of␣size␣%d!\n", del, n);
  exit(−200);
}
⟨ Remove del leaves, one by one 15* ⟩;
```
See also section 18*.

This code is used in section 3*.

15*. The following approach to leaf deletion keeps the rooted tree structure, but will optionally remove the root if it becomes a leaf. First we identify the nonroot leaves, by calculating the degree of all nodes (not including their parents), simultaneously identifying all parents and putting all leaves into a sequential list. Then we remove random elements from that list; the removal of a leaf decreases the degree of its parent, so that its parent might become a leaf for the next round.

The parents are temporarily stored in *arc* fields of *tnode*, and the leaves are temporarily stored in *arc* fields of *snode* (because those fields are currently unused).

In this process, z is the current root, and gg is the current number of leaves. When a leaf has been deleted, we reset its parent to -1.

```
#define leaf(k) snode[k].arc
#define parent(k) tnode[k].arc

⟨Remove del leaves, one by one 15*⟩ ≡
  z = gg = 0; leafprep(0);
  m = n − del;
  for ( ; del; del−−) {
    d = (o, tnode[z].deg ≡ 1);    /* is the root also a leaf? */
    r = gb_unif_rand(gg + d);    /* choose a random leaf */
    if (r ≡ gg) ooo, z = tnode[z].child;    /* delete the root */
    else {
      o, q = leaf(r);
      oo, p = parent(q), parent(q) = −1;
      oo, tnode[p].deg−−;
      if (tnode[p].deg) o, p = leaf(−−gg);
      o, leaf(r) = p;    /* replace q by another leaf */
    }
  }
  restructure(z);    /* now really remove the nodes with negative parent */
```

This code is used in section 14*.

16*. ⟨Subroutines 7*⟩ +≡

```
  int gg;    /* a global counter */

  int leafprep(int p)
  {
    register int d, q;

    mems += suboverhead;
    for (o, d = 0, q = tnode[p].child; q; o, q = tnode[q].sib) {
      d++, parent(q) = p;
      if (leafprep(q) ≡ 0) o, leaf(gg++) = q;
    }
    o, tnode[p].deg = d;
    return d;
  }
```

17* The *restructure* routine makes the *child* and *sib* fields great again.

⟨ Subroutines 7* ⟩ +≡

```
void restructure(int p)
{
  register int q;
  mems += suboverhead;
  o, q = tnode[p].child;
  while (q ∧ (o, parent(q) < 0))  o, q = tnode[q].sib;
  o, tnode[p].child = q;
  while (q) {
    if (o, tnode[q].child)  restructure(q);
    o, p = q, q = tnode[q].sib;
    while (q ∧ (o, parent(q) < 0))  o, q = tnode[q].sib;
    o, tnode[p].sib = q;
  }
}
```

18* OK, we've pruned away the desired number of leaves; but we're still not done, because this program also wants the root of S to be a leaf. All the nodes must be renumbered internally.

So we transform the *tnode* array so that the root is a leaf. Then we copy *tnode* to *snode*, remapping all node numbers as we go.

⟨ Input the tree S 14* ⟩ +≡
  ⟨ Make the root of *tnode* into a leaf 19* ⟩;
  ⟨ Copy and remap *tnode* into *snode* 20* ⟩;

19* I thought this would be easier than it has turned out to be. Did I miss something? It's a nice little exercise in datastructurology.

Node z moves to node n, so that it can become a child or a sibling (in case $z = 0$).

⟨ Make the root of *tnode* into a leaf 19* ⟩ ≡

```
oo, r = n, p = tnode[z].child, tnode[r].child = p;
while (o, q = tnode[p].child) {      /* make p the root, retaining its child q */
  o, k = tnode[p].sib, s = tnode[q].sib;
  o, tnode[p].sib = 0;
  o, tnode[q].sib = r;
  o, tnode[r].child = k, tnode[r].sib = s;
  r = p, p = q;
}
ooo, s = tnode[p].sib, tnode[p].sib = 0, tnode[p].child = r, tnode[r].child = s;      /* now p is the root */
```

This code is used in section 18*.

20* ⟨ Copy and remap *tnode* into *snode* 20* ⟩ ≡

```
gg = 0; copyremap(p);
if (gg ≠ m) {
  fprintf(stderr, "I'm␣confused!\n");
  exit(−666);
}
oo, snode[z].arc = snode[n].arc;
```

This code is used in section 18*.

21*. This recursion is a bit tricky, and I wonder what's the best way to explain it. (An exercise for the reader.)

⟨ Subroutines 7* ⟩ +≡

```
void copyremap(int r)
{
  register int p, q;
  mems += suboverhead;
  gg++;
  o, p = tnode[r].child;
  if (¬p) return;
  o, snode[gg − 1].child = gg;      /* copy a (remapped) child pointer */
  while (1) {
    q = gg;      /* the future interior name of p */
    copyremap(p);
    o, p = tnode[p].sib;
    if (¬p) return;
    o, snode[q].sib = gg;      /* copy a (remapped) sibling pointer */
  }
}
```

22*. ⟨ Input the tree T 22* ⟩ ≡

```
n = read_rectree(argv[1]);
⟨ Allocate the arcs 24 ⟩;
fprintf(stderr, "OK,␣I've␣got␣%d␣nodes␣for␣S␣and␣%d␣nodes␣for␣T,␣max␣degree␣%d.\n", m, n,
    maxdeg);
```

This code is used in section 3*.

23. The target tree T has $2(n-1)$ arcs, from each nonroot node to its parent and vice versa. The arcs from u to v are assigned consecutive integers, from 0 to $2n-3$, in lexicographic order of $(\deg(v), v, u)$. (Well, the second and third components might not be in numerical order; but all d arcs from a vertex of degree d are consecutive, beginning with the arc to the parent.)

In order to assign these numbers, we keep lists of all nodes having a given degree, using the *arc* fields temporarily to link them together.

⟨ Subroutines 7* ⟩ +≡

```
void fixdeg(int p)
{
  register int d, q;
  mems += suboverhead;
  for (o, d = 1, q = tnode[p].child; q; o, d++, q = tnode[q].sib) fixdeg(q);
  if (p) ooo, tnode[p].arc = head[d], tnode[p].deg = d, head[d] = p;
      /* p is not the root; it has d neighbors including its parent */
  else ooo, tnode[0].arc = head[d − 1], tnode[0].deg = d − 1, head[d − 1] = −1;
      /* root is temporarily renamed −1 */
}
```

24. We set *thresh*[*d*] to the number of the first arc for a node of degree d or more.

⟨ Allocate the arcs 24 ⟩ ≡

```
fixdeg(0);
for (d = 1, e = 0; e < 2 * n - 2; d++) {
  o, thresh[d] = e;
  for (o, p = head[d]; p; e += d, p = q) {
    if (p < 0) p = 0;
    oo, q = tnode[p].arc, tnode[p].arc = e;
  }
}
for (maxdeg = d - 1, emax = e; d < m; d++) o, thresh[d] = emax;
⟨ Allocate the dual arcs 26 ⟩;
```

This code is used in section 22*.

25. The arc from u to v has a dual, namely the arc from v to u. (And conversely.) We've assigned numbers to the arcs that go to a parent; the other arcs are their duals.

26. ⟨ Allocate the dual arcs 26 ⟩ ≡

```
for (p = 0; p < n; p++) {
  for (oo, e = (p ? tnode[p].arc : tnode[p].arc - 1), q = tnode[p].child; q; o, q = tnode[q].sib) {
    ooo, dual[tnode[q].arc] = ++e, dual[e] = tnode[q].arc;
    oooo, uert[dual[e]] = vert[e] = p, uert[e] = vert[dual[e]] = q;
  }
}
```

This code is used in section 24.

27. ⟨ Global variables 6* ⟩ +≡

```
int head[maxn];          /* heads of lists by degree */
int maxdeg;              /* maximum degree seen */
int thresh[maxn];        /* where the arcs from large degree nodes start */
int vert[maxn + maxn];   /* the source vertex of each arc */
int uert[maxn + maxn];   /* the target vertex of each arc */
int dual[maxn + maxn];   /* the dual of each arc */
int emax;                /* the total number of arcs */
```

28. The master control. There's a two-dimensional array called *sol* that pretty much governs the computation. The first index, p, is a node of S; the second index, e, is an arc of T. If e is the arc from u to v, consider the subtree of T that's rooted at u and includes v; we call it "subtree e." If there's no way to embed the subtree of S rooted at p to subtree e, by mapping p to v, then we'll set $sol[p][e]$ to zero. Otherwise we'll set $sol[p][e]$ to a nonzero value, with which we could deduce such an embedding if called on to do so.

The basic idea is simple, working recursively up from small subtrees to larger ones: Suppose p has r children, $q_1, \ldots, q_r$; and suppose v has $s+1$ neighbors, $u_0, \ldots, u_s$. Suppose further that we've already computed $sol[q_i][e_j]$, for $1 \le i \le r$ and $0 \le j \le s$, where e_j is the arc from v to u_j. Matula's algorithm will tell us how to compute $sol[p][dual[e_j]]$ for $0 \le j \le s$. Thus we can fill in the rows of *sol* from bottom to top; eventually $sol[1]$ will tell us if we can embed *all* of S.

Let's look closely at that crucial subproblem: How, for example, do we know if $sol[p][dual[e_0]]$ should be zero or nonzero? That subproblem means that we want to embed subtree p into the subtree below the arc from u_0 to v. And the subproblem is clearly solvable if and only if we can match up each child q_i of p with a distinct child u_j of v, in such a way that $sol[p_i][q_j]$ is nonzero. Aha, yes: It's a bipartite matching problem! And there are good algorithms for bipartite matching!

More generally, consider the subproblem in which u_j is a parent of v in T, while $u_0, \ldots, u_{j-1}, u_{j+1}, \ldots, u_s$ are children. Matula discovered that these subproblems are essentially the same, for all j between 0 and s. It's a beautiful way to save a factor of n by combining similar subproblems.

So that's what we'll do, with a recursive procedure called *solve*.

⟨Solve the problem 28⟩ ≡
  $z = solve(1)$;

This code is used in section 2*.

29. The task of *solve*, given a node p of S, is to set the values of $sol[p][e]$ for each arc e.

The base case of this recursion occurs when p is a leaf; a leaf can be embedded anywhere.

Another easy case occurs when subtree e of T has too small a degree to support any embedding.

If some descendant d of p can't be embedded, *solve* returns $-d$. Otherwise *solve* returns the number of 1s in $sol[p]$.

```
⟨Subroutines 7*⟩ +≡
  int solve(int p)
  {
    register int e, m, n, q, r, z;
    mems += suboverhead;
    o, q = snode[p].child;
    if (q ≡ 0) {
      for (e = 0; e < emax; e++) o, sol[p][e] = 1;
      return emax;
    }
    for (r = 0; q; o, r++, q = snode[q].sib) {
      z = solve(q);
      if (z ≤ 0) return (z ? z : −q);     /* if we can't embed a subtree, we can't embed S */
    }     /* now sol[q][e] is known for all children q of p and all arcs e */
    for (o, z = e = 0; e < thresh[r + 1]; e++) o, sol[p][e] = 0;     /* degree too small */
    for (n = r + 1; e < emax; e += n) {
      ⟨Local variables for the HK algorithm 35⟩;
      while (o, e ≡ thresh[n + 1]) n++;     /* advance n to the degree of vert[e] */
      ⟨Set up Matula's bipartite matching problem for p and e 30*⟩;
      ⟨Solve that problem and update sol[p][e . . e + n − 1] 43⟩;
    }
    return z;
  }
```

30*. Bipartite matching chez Hopcroft and Karp. Now we implement the classic HK algorithm for bipartite matching, stealing most of the code from the program HOPCROFT-KARP. (The reader should consult that program for further remarks and proofs.) The children of p play the role of "boys" in that algorithm, and the arcs for neighbors of v play the role of "girls." That algorithm is slightly simplified here, because we are interested only in cases where all the boys can be matched. (There always are more girls than boys, in our case.)

In Matula's matching problem, p is a vertex of S that has children $q_1, \ldots, q_r$; e is an arc of T from $v = vert[e]$ to $u = uert[e]$, where v has $s+1$ neighbors $u_0, \ldots, u_s$. The matching problem will have $m \le r$ boys and $n = s+1$ girls.

We use a simple data structure to represent the bipartite graph: The potential partners for girl j are in a linked list beginning at *glink*[j], linked in *next*, and terminated by a zero link. The partner at link l is stored in *tip*[l].

⟨ Set up Matula's bipartite matching problem for p and e 30* ⟩ ≡
```
⟨ Initialize the tables needed for n girls 32 ⟩;
for (o, t = m = 0, b = snode[p].child; b; o, b = snode[b].sib) ⟨ Record the potential matches for boy b 31 ⟩;
if (m ≡ 0) goto yes_sol;     /* every boy matches every girl */
if (m * n > record) {
  record = m * n;
  fprintf(stderr, "␣...matching␣%d␣boys␣to␣%d␣girls\n", m, n);
}
```
This code is used in section 29.

31. If b is matched to every girl, we needn't include him in the bipartite graph. (This situation happens rather often, for example whenever b is a leaf, so it's wise to test for it.) On the other hand, if some boy isn't matched to any girl, we know in advance that there will be no bipartite matching.

The HK algorithm uses a *mate* table, to indicate the current mate of every boy as it constructs tentative matchings. There's also an inverse table, *imate*, for the girls. If b has no mate, *mate*[b] = 0; if g has no mate, *imate*[g] = 0. But if b is tentatively matched to g, we have *mate*[b] = g and *imate*[g] = b.

⟨ Record the potential matches for boy b 31 ⟩ ≡
```
{
  for (g = e; g < e + n; g++)
    if (oo, sol[b][dual[g]] ≡ 0) break;
  if (g ≡ e + n) continue;     /* boy b fits anywhere, so omit him */
  oo, m++, mate[b] = mark[b] = 0;
  for (k = t, gg = e; gg < g; gg++) oooo, tip[++t] = b, next[t] = glink[gg], glink[gg] = t;
  for (g++; g < e + n; g++)
    if (oo, sol[b][dual[g]]) oooo, tip[++t] = b, next[t] = glink[g], glink[g] = t;
  if (k ≡ t) goto no_sol;     /* boy b fits nowhere, so give up */
}
```
This code is used in section 30*.

32. We've now created a bipartite graph with m boys, n girls, and t edges.

The HK algorithm proceeds in *rounds*, where each round finds a maximal set of so-called SAPs, which are are vertex-disjoint augmenting paths of the shortest possible length. If a round finds k such paths, it reduces the number of free boys (and free girls) by k. Eventually, after at most $2\sqrt{n}$ rounds, we reach a state where no more SAPs exist. And then we have a solution, if and only if no boys are still free (hence $n-m$ girls are still free).

Variable f in the algorithm denotes the current number of free girls. They all appear in the first f positions of any array called *queue*, which governs a breadth-first search. This array has an inverse, *iqueue*: If g is free, we have $queue[iqueue[g]] = g$.

⟨ Initialize the tables needed for n girls 32 ⟩ ≡

```
for (g = e; g < e + n; g++) oooo, glink[g] = 0, imate[g] = 0, queue[g - e] = g, iqueue[g] = g - e;
f = n;
```

This code is used in section 30*.

33. The key idea of the HK algorithm is to create a directed acyclic graph in which the paths from a dummy node called $\top$ to a dummy node called $\bot$ correspond one-to-one with the augmenting paths of minimum length. Each of those paths will contain *final_level* existing matches.

This dag has a representation something like our representation of the girls' choices, but even sparser: The first arc from boy i to a suitable girl is in *blink*[i], with *tip* and *next* as before. Each girl, however, has exactly one outgoing arc in the dag, namely her *imate*. An *imate* of 0 is a link to $\bot$. The other dummy node, $\top$, has a list of free boys, beginning at *dlink*.

An array called *mark* keeps track of the level (plus 1) at which a boy has entered the dag. All marks must be zero when we begin.

The *next* and *tip* arrays must be able to accommodate $2t+m$ entries: t for the original graph, t for the edges at round 0, and m for the edges from $\top$.

```
#define maxg (2 * maxn)  /* upper limit on the number of girls */
#define maxt (maxn * maxg)  /* upper limit on the number of bipartite edges */
```

⟨ Global variables 6* ⟩ +≡

```
int blink[maxn], glink[maxg];  /* list heads for potential partners */
int next[maxt + maxt + maxn], tip[maxt + maxt + maxn];  /* links and suitable partners */
int mate[maxn], imate[maxg];
int queue[maxg];  /* girls seen during the breadth-first search */
int iqueue[maxg];  /* inverse permutation, for the first f entries */
int mark[maxn];  /* where boys appear in the dag */
int marked[maxn];  /* which boys have been marked */
int dlink;  /* head of the list of free boys in the dag */
```

34. ⟨ Build the dag of shortest augmenting paths (SAPs) 34 ⟩ ≡

```
final_level = -1, tt = t;
for (marks = l = i = 0, q = f; ; l++) {
  for (qq = q; i < qq; i++) {
    o, g = queue[i];
    for (o, k = glink[g]; k; o, k = next[k]) {
      oo, b = tip[k], pp = mark[b];
      if (pp ≡ 0) ⟨ Enter b into the dag 36 ⟩
      else if (pp ≤ l) continue;
      oooo, tip[++tt] = g, next[tt] = blink[b], blink[b] = tt;
    }
  }
  if (q ≡ qq) break;    /* nothing new on the queue for the next level */
}
```

This code is used in section 43.

35. ⟨ Local variables for the HK algorithm 35 ⟩ ≡

```
register int b, f, g, i, j, k, l, t, gg, pp, qq, tt, final_level, marks;
```

This code is used in section 29.

36. Once we know we've reached the final level, we don't allow any more boys at that level unless they're free. We also reset q to qq, so that the dag will not reach a greater level.

⟨ Enter b into the dag 36 ⟩ ≡

```
{
  if (final_level ≥ 0 ∧ (o, mate[b])) continue;
  else if (final_level < 0 ∧ (o, mate[b] ≡ 0)) final_level = l, dlink = 0, q = qq;
  ooo, mark[b] = l + 1, marked[marks++] = b, blink[b] = 0;
  if (mate[b]) oo, queue[q++] = mate[b];
  else oo, tip[++tt] = b, next[tt] = dlink, dlink = tt;
}
```

This code is used in section 34.

37. We have no SAPs if and only no free boys were found.

⟨ If there are no SAPs, **break** 37 ⟩ ≡

```
if (final_level < 0) break;
```

This code is used in section 43.

38. ⟨ Reset all marks to zero 38 ⟩ ≡

```
while (marks) oo, mark[marked[--marks]] = 0;
```

This code is used in section 39.

39. We've just built the dag of shortest augmenting paths, by starting from dummy node ⊥ at the bottom and proceeding breadth-first until discovering *final_level* and essentially reaching the dummy node ⊤. Now we more or less reverse the process: We start at ⊤ and proceed *depth*-first, harvesting a maximal set of vertex-disjoint augmenting paths as we go. (Any maximal set will be fine; we needn't bother to look for an especially large one.)

The dag is gradually dismantled as SAPs are removed, so that their boys and girls won't be reused. A subtle point arises here when we look at a girl g who was part of a previous SAP: In that case her mate will have been changed to a boy whose *mark* is negative. This is true even if $l = 0$ and g was previously free.

⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 39 ⟩ ≡

```
  while (dlink) {
    oo, b = tip[dlink], dlink = next[dlink];
    l = final_level;
  enter_level: o, boy[l] = b;
  advance: if (o, blink[b]) {
      ooo, g = tip[blink[b]], blink[b] = next[blink[b]];
      if (o, imate[g] ≡ 0) ⟨ Augment the current matching and continue 40 ⟩;
      if (o, mark[imate[g]] < 0) goto advance;
      b = imate[g], l−−;
      goto enter_level;
    }
    if (++l > final_level) continue;
    o, b = boy[l];
    goto advance;
  }
  ⟨ Reset all marks to zero 38 ⟩;
```

This code is used in section 43.

40. At this point $g = g_0$ and $b = boy[0] = b_0$ in an augmenting path. The other boys are $boy[1]$, $boy[2]$, and so on.

⟨ Augment the current matching and **continue** 40 ⟩ ≡

```
  {
    if (l) fprintf(stderr, "I'm␣confused!\n");     /* a free girl should occur only at level 0 */
    ⟨ Remove g from the list of free girls 42 ⟩;
    while (1) {
      o, mark[b] = −1;
      ooo, j = mate[b], mate[b] = g, imate[g] = b;
      if (j ≡ 0) break;     /* b was free */
      o, g = j, b = boy[++l];
    }
    continue;
  }
```

This code is used in section 39.

41. ⟨ Global variables 6* ⟩ +≡

```
  int boy[maxn];     /* the boys being explored during the depth-first search */
```

42. ⟨ Remove g from the list of free girls 42 ⟩ ≡

```
  f−−;     /* f is the number of free girls */
  o, j = iqueue[g];     /* where is g in queue? */
  ooo, i = queue[f], queue[j] = i, iqueue[i] = j;     /* OK to clobber queue[f] */
```

This code is used in section 40.

43. Hey folks, we've now got all the infrastructure and machinery of the HK algorithm in place. It only remains to actually perform the algorithm.

⟨ Solve that problem and update $sol[p][e\,..\,e+n-1]$ 43 ⟩ ≡
  while (1) {
    ⟨ Build the dag of shortest augmenting paths (SAPs) 34 ⟩;
    ⟨ If there are no SAPs, **break** 37 ⟩;
    ⟨ Find a maximal set of disjoint SAPs, and incorporate them into the current matching 39 ⟩;
  }
  if $(f \equiv n - m)$ ⟨ Store the solution in $sol[p]$ 44 ⟩
  else
  no_sol: **for** $(k = 0;\ k < n;\ k{+}{+})$ $o, sol[p][e + k] = 0$;
  continue;   /* resume the loop on e */
yes_sol: **for** $(k = 0;\ k < n;\ k{+}{+})$ $o, sol[p][e + k] = 1$;
  $z \mathrel{+}= n$;

This code is used in section 29.

44. The climax. But it's still necessary to don our thinking cap and figure out exactly what we've got, when the HK algorithm has found a perfect matching of m boys to $n > m$ girls.

Our job is to update n entries of *sol*, one for each girl. That entry should be 0 if and only if the girl has a mate in *every* perfect match. (Because the subgraph isomorphism will assign her to the parent of v in T, while the mated girls will be assigned to some of v's children in the embedding.)

Suppose, for example, that the bipartite matching is unique. In that case we'll want to set $sol[p][g] = 0$ if and only if $imate[g] \neq 0$.

Usually, however, there will be a number of perfect matchings, involving different sets of girls. Matula noticed, in Theorem 3.4 of his paper, that it's actually easy to distinguish the forcibly matched girls from the others. Moreover — fortunately for us — the necessary information is sitting conveniently in the dag, when the HK algorithm ends!

Indeed, it's not difficult to verify that every perfect matching either includes g or corresponds to a path from g to $\perp$ in the dag. Therefore — ta da — the freeable girls are precisely the girls in the first q positions of *queue*!

⟨ Store the solution in $sol[p]$ 44 ⟩ ≡
```
{
  for (k = 0; k < n; k++) o, sol[p][e + k] = 0;
  for (k = 0; k < q; k++) ooo, z++, sol[p][queue[k]] = 1;
  ⟨ Store the mate information too 45 ⟩;
}
```
This code is used in section 43.

45. If we're interested only in whether or not an embedding of S into T exists, the *sol* array tells us everything we need to know.

But if we want to actually see an embedding, we might wish to store the solutions to the matching problems we've solved, so that we don't need to repeat those calculations later.

In a way that's foolish: Only a small number of matching problems will need to be redone. So we're wasting space by storing this extra information — which doesn't fit in *sol*. And we're gaining only an insignificant amount of time.

Still, the details are interesting, so I'm plunging ahead. Let *solx* and *soly* be arrays, such that the solution to the bipartite matching problem in $sol[p][e \mathinner{..} e + n - 1]$ is recorded in $solx[p][e \mathinner{..} e + n - 1]$ and $soly[p][e \mathinner{..} e + n - 1]$. (Both *solx* and *soly* are arrays of **int**, while *sol* itself could have been an array of single bits.)

It suffices to store the final *imate* table in *solx*, and to store links of a path from g to $\perp$ in *soly*.

⟨ Store the mate information too 45 ⟩ ≡
```
for (g = e; g < e + n; g++) oo, solx[p][g] = imate[g];
for (k = 0; k < q; k++) {
  o, g = queue[k];
  if (o, imate[g]) oooo, soly[p][g] = tip[blink[imate[g]]];
}
```
This code is used in section 44.

46. ⟨ Global variables 6* ⟩ +≡
```
int sol[maxn][maxg];   /* the master control matrix */
int solx[maxn][maxg];  /* imate info for bipartite solutions */
int soly[maxn][maxg];  /* final dag info for bipartite solutions */
```

47. The anticlimax. When all has been done but not yet said, we want to tell the user what happened. At this point z holds the value of $solve(1)$. It's negative, say $-d$, if the subtree of S rooted at node d and its parent cannot be isomorphically embedded in T. Otherwise z is zero if S itself cannot be embedded, although every subtree of node 1 is embeddable. Otherwise z is the number of arcs e of T for which there's an embedding with node 0 of S mapped into the root of subtree e.

(In the latter case, notice that z is probably *not* the actual total number of embeddings. It's just the number of places where we could start an embedding and obtain at least one success.)

```
⟨Report the solution 47⟩ ≡
  if (z < 0)
    fprintf(stderr, "Failure;␣We␣can't␣even␣embed␣node␣%d␣and␣its␣parent.\n", encode(−z));
  else {
    fprintf(stderr, "There␣%s␣%d␣place%s␣to␣anchor␣an␣embedding␣of␣node␣1.\n",
        z ≡ 1 ? "is" : "are", z, z ≡ 1 ? "" : "s");
    if (z) ⟨Print a solution 49*⟩;
  }
```

This code is used in section 2*.

48. Our final task is to harvest the information in *sol*, *solx*, and *soly*, in order to present the user with the images of nodes 0, 1, ... of S, in one of the possible embeddings found.

To do this, we assign an edge called $solarc[p]$ to each nonroot vertex p of S. If this arc runs from v to u, it means that the embedding maps p to v and p's parent to u. These arcs are assigned top-down, starting with the rightmost e such that $sol[1][e] = 1$.

49*. ⟨Print a solution 49*⟩ ≡

```
  {
    for (e = emax − 1; o, sol[1][e] ≡ 0; e−−) ;
    oo, solarc[1] = e;
    for (p = 1; p < m; p++)
      if (o, snode[p].child) {
        for (q = snode[p].child; q; o, q = snode[q].sib) o, mate[q] = 0;
        oo, z = solarc[p], v = vert[z];
        o, e = tnode[v].arc, n = tnode[v].deg;
        for (g = e; g < e + n; g++) ooo, q = imate[g] = solx[p][g], mate[q] = g;
        ⟨Find a matching in which imate[z] = 0 51⟩;
        for (o, g = e, q = snode[p].child; q; o, q = snode[q].sib) {
          if (o, mate[q]) oo, solarc[q] = dual[mate[q]];
          else {    /* choose mate for a universally matchable boy */
            while (g ≡ z ∨ (o, imate[g])) g++;
            oo, solarc[q] = dual[g++];
          }
        }
      }
    oo, printf("%d", uert[solarc[1]]);
    for (p = 1; p < m; p++) oo, printf("␣%d", vert[solarc[p]]);
    printf("\n");
  }
```

This code is used in section 47.

50. ⟨Global variables 6*⟩ +≡

```
  int solarc[maxn];    /* key arcs in the solution */
```

51. Here finally is a kind of cute way to end, using the theory of *non*-augmenting paths. (That theory can be understood from the construction of the final, incomplete dag in the HK algorithm, whose critical structure we stored in *soly*[*p*].)

⟨ Find a matching in which $imate[z] = 0$ 51 ⟩ ≡

```
for (k = 0, g = z; o, q = imate[g]; k = q) {
    o, imate[g] = k;
    o, g = soly[p][g];
    o, mate[q] = g;
}
o, imate[g] = k;
```

This code is used in section 49*.

52*. ⟨ Global variables 6* ⟩ +≡

```
int record;    /* the largest bipartite matching problem encountered so far */
```

53.* Index.

The following sections were changed by the change file: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 30, 49, 52, 53.

MATULA-BIG-PLANTED