(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Introduction. This program combines the ideas of GRAYSPAN and SPSPAN, resulting in a glorious routine that generates all spanning trees of a given graph, changing only one edge at a time, with "guaranteed efficiency"—in the sense that the total running time is $O(m+n+t)$ when there are m edges, n vertices, and t spanning trees.

The reader should be familiar with both GRAYSPAN and SPSPAN, because their principles of operation are not repeated here.

The first command line argument is the name of a file that specifies an undirected graph in Stanford GraphBase SAVE_GRAPH format. Additional command line arguments are ignored except that they cause more verbose output. The least verbose output contains only overall statistics about the total number of spanning trees found and the total number of mems used.

```
#define verbose (argc > 2)
#define extraverbose (argc > 3)
#define o mems++
#define oo mems += 2
#define ooo mems += 3
#define oooo mems += 4
#define ooooo mems += 5
#define oooooo mems += 6
#include "gb_graph.h"
#include "gb_save.h"
  ⟨ Preprocessor definitions ⟩
  double mems; /* memory references made */
  double count; /* trees found */
  Graph *gg; /* a global copy of g */
  int argc; /* a global copy of argc */
  ⟨ Type definitions 5 ⟩
  ⟨ Global variables 7 ⟩
  ⟨ Subroutines 9 ⟩
  main(int argc, char *argv[])
  {
    ⟨ Local variables 3 ⟩;
    ⟨ Input the graph 2 ⟩;
    if (n > 1) ⟨ Generate all spanning trees 25 ⟩;
    printf("Altogether␣%.15g␣spanning␣trees,␣using␣%.15g␣mems.\n", count, mems);
    exit(0);
  }
```

2. ⟨ Input the graph 2 ⟩ ≡

```
if (argc < 2) {
  fprintf(stderr, "Usage: %s foo.gb [[gory] details]\n", argv[0]);
  exit(-11);
}
xargc = argc;
gg = g = restore_graph(argv[1]);
if (¬g) {
  fprintf(stderr, "Sorry, can't create the graph from file %s! (error code %d)\n", argv[1],
      panic_code);
  exit(-2);
}
⟨ Allocate additional storage 6 ⟩;
⟨ Check the graph for validity and prepare it for action 18 ⟩;
```

This code is used in section 1.

3. ⟨ Local variables 3 ⟩ ≡

```
register Graph *g;      /* the graph we're dealing with */
register int n;      /* the number of vertices */
register int m;      /* the number of edges */
register int l;      /* the current level of shrinkage */
register int k;      /* current integer of interest */
register Vertex *u, *v, *w;      /* current vertices of interest */
register Arc *e, *ee, *f, *ff;      /* current edges of interest */
register Bond *b;      /* current interest-bearing bond */
```

This code is used in section 1.

4. The method and its data structures. The basic idea of this program, which goes back to Malcolm Smith's M.S. thesis, "Generating spanning trees" (University of Victoria, 1997), is to modify the GRAYSPAN algorithm, replacing edges by series-parallel subgraphs that we will call "bonds."

A *bond* between vertices u and v is either an edge of the given graph, or a sequence of bonds joined in series, or a set of bonds joined in parallel. The GRAYSPSPAN algorithm essentially uses the GRAYSPAN method to generate spanning trees with respect to bonds, together with the SPSPAN method to generate all of the corresponding spanning trees with respect to the original edges.

Suppose we start by considering each edge to be a bond of the simplest kind. Then we can remove multiple bonds, if any, by combining them in parallel. And we can remove vertices of degree 2 by combining their adjacent bonds in series. A vertex of degree 1 and its adjacent bond can be removed from the graph, if we require that bond to be present in every spanning tree. Under the assumption that the given graph is connected, repeated reductions of this kind will eventually lead to a simple graph that is *irreducible*, either trivial (with only one point) or with all vertices of degree 3 or more.

For example, consider the smallest nontrivial irreducible graph, having four vertices and bonds between any two of them. Suppose the vertices are v_1, v_2, v_3, v_4 and the bonds are b_{12}, b_{13}, b_{14}, b_{23}, b_{24}, b_{34}. If the GRAYSPAN algorithm produces the spanning tree $b_{12}b_{13}b_{24}$, say, the SPSPAN algorithm is used to cycle through all the 1-configs of b_{12}, b_{13}, and b_{24} together with all the 0-configs of the other bonds b_{14}, b_{23}, b_{34}. Then if GRAYSPAN produces its next spanning tree by changing b_{24} to b_{34}, this change corresponds to removing the designated leaf of bond b_{24} and including the designated leaf of b_{34}, after which SPSPAN runs through the 1-configs of b_{12}, b_{13}, b_{34} with the 0-configs of b_{14}, b_{23}, b_{24}.

The actual operation of this program is not quite the same as just described, because the operation of shrinking b_{12} causes the resulting graph to reduce to a triviality; the GRAYSPAN algorithm never gets to the case $n = 2$ that used to be its goal. But conceptually we make progress on the graph as a whole by shrinking edges and removing nonbridges just as in GRAYSPAN, and each reduction in the number of bonds takes more of the computation into the highly efficient SPSPAN domain.

5. A suitable data structure to support all this machinery can be fashioned from a struct of type **Bond**, which contains the fields *typ*, *val*, *des*, *done*, *focus*, and *rsib* that we used in SPSPAN. We also need *left* and *right* pointers, because we must keep branch nodes in a doubly linked list instead of allocating them sequentially and statically as we did before. Then there are *lchild*, *lsib*, and *scope* pointers for constructing new bonds from old ones; also *lhist* and *rhist* for deconstructing when bonds are being undone. Finally there's yet another doubly linked list, containing the top-level bonds of the current spanning tree; its link fields are called *up* and *down*.

⟨ Type definitions 5 ⟩ ≡

```
typedef struct bond_struct {
  int typ;      /* 1 for series bonds, 0 for parallel bonds, 2 for deletion records */
  int val;      /* 1 if the bond is in the current spanning tree */
  struct bond_struct *lchild;      /* leftmost child; Λ for a leaf */
  struct bond_struct *lsib;      /* left sibling; wraps around cyclically */
  struct bond_struct *rsib;      /* right sibling; wraps around cyclically */
  struct bond_struct *des;      /* designated child */
  struct bond_struct *done;      /* final designated child in a sweep */
  struct bond_struct *scope;      /* last branch descendant in preorder */
  struct bond_struct *focus;      /* magic pointer control for Gray products */
  struct bond_struct *left;      /* left neighbor in the action list */
  struct bond_struct *right;      /* right neighbor in the action list */
  struct bond_struct *up;      /* upper neighbor in the tree list */
  struct bond_struct *down;      /* lower neighbor in the tree list */
  struct bond_struct *lsave,*rsave;      /* needed for restoration in one case */
  Arc *lhist,*rhist;      /* the edges that spawned this bond */
} Bond;
```

This code is used in section 1.

6. If the graph has m edges, we put the basic one-edge bonds into $bondbase + k$ for $1 \le k \le m$; series and parallel combinations, and "deletion records" for undoing a bond deletion, go into the subsequent locations, in a last-in-first-out manner.

```
#define isleaf(b) ((b) ≤ topleaf)  /* is b simply an edge? */
#define action bondbase  /* head of the action list */
#define tree (&treehead)  /* head of the tree list */
```

⟨ Allocate additional storage 6 ⟩ ≡

```
m = g→m/2;
bondbase = (Bond *) calloc(3 * m, sizeof(Bond));
action→left = action→right = action, action→typ = 2, action→focus = action;
  /* the action list starts empty */
tree→up = tree→down = tree;  /* and so does the tree list */
```

This code is used in section 2.

7. ⟨ Global variables 7 ⟩ ≡

```
Bond *bondbase;  /* base address for almost all Bond structs */
int bondcount;  /* the number of bonds currently defined */
Bond *topleaf;  /* dividing line between leaves and branches */
Bond treehead;  /* header for the tree list */
Bond *inbond, *outbond;  /* bonds to be included and excluded next */
Bond James;  /* my little joke */
```

This code is used in section 1.

8. The bonds between vertices are represented as **Arc** structs in almost the usual way; namely, a bond between u and v appears as an arc e with $e\text{→}tip = v$ in the list $u\text{→}arcs$, and as an arc f with $f\text{→}tip = u$ in the list $v\text{→}arcs$, where $f = mate(e)$ and $e = mate(f)$. However, as in GRAYSPAN, we doubly link the arc list, making $e\text{→}prev\text{→}next = e\text{→}next\text{→}prev = e$ and starting the list with a dummy entry called its header.

We also include a new field $e\text{→}bond$ pointing to the **Bond** struct that carries further information. This one has to be coerced to type (**Bond** *) when used in the C code.

```
#define deg u.I  /* utility field u of each vertex holds its current degree */
#define prev a.A  /* utility field a of each arc holds its backpointer */
#define bond b.A  /* utility field b of each arc leads to the corresponding Bond */
#define mate(e) (edge_trick & (siz_t)(e) ? (e) - 1 : (e) + 1)
#define bn(b) ((b) - bondbase)  /* the number of bond b, for printout */
#define ebn(e) bn((Bond *)((e)→bond))
#define delete(e) ee = e, oooo, ee→prev→next = ee→next, ee→next→prev = ee→prev
#define undelete(e) ee = e, oooo, ee→next→prev = ee, ee→prev→next = ee
```

9. Diagnostic routines. Here are a few handy ways to look at the current data.

```
⟨ Subroutines 9 ⟩ ≡
  void printgraph(void)      /* prints the current graph */
  {
    register Vertex *v;
    register Arc *e;
    for (v = gg→vertices; v < gg→vertices + gg→n; v++)
      if (v→mark ≥ 0) {
        printf("Bonds␣from␣%s:", v→name);
        for (e = v→arcs→next; e→tip; e = e→next) printf("␣%s(%d)", e→tip→name, ebn(e));
        printf("\n");
      }
  }
```

See also sections 10, 11, 12, 31, 38, and 39.

This code is used in section 1.

10.

```
⟨ Subroutines 9 ⟩ +≡
  void printbond(Bond *b)
  {
    printf("%d:%c", bn(b), isleaf(b) ? 'l' : b→typ ≡ 2 ? 'r' : b→typ ≡ 1 ? 's' : 'p');
    if (b→typ ≡ 2) {
      printf("␣lhist=%d", ebn(b→lhist));
      if (b→rhist) printf("␣rhist=%d", ebn(b→lhist));
    } else {
      printf("%c", b→val + '0');
      if (b→lsib) printf("␣lsib=%d,␣rsib=%d", bn(b→lsib), bn(b→rsib));
      if (¬isleaf(b)) {
        if (b→focus ≠ b) printf("␣focus=%d", bn(b→focus));
        printf("␣lchild=%d,␣scope=%d,␣des=%d,␣done=%d", bn(b→lchild), bn(b→scope), bn(b→des),
            bn(b→done));
      }
    }
    printf("\n");
  }
```

11. ⟨ Subroutines 9 ⟩ +≡

```
void printaction(void)      /* prints the current action list */
{
  register Bond *b;
  for (b = action→right; b ≠ action; b = b→right) printbond(b);
}
void printtree(void)      /* prints the current tree list */
{
  register Bond *b;
  for (b = tree→down; b ≠ tree; b = b→down) printbond(b);
}
void printleaves(void)      /* prints all the leaves */
{
  register Bond *b;
  for (b = bondbase + 1; b ≤ topleaf; b++) printbond(b);
}
```

12. Since there's so much redundancy in the data structures, I need reassurance that I haven't slipped up and forgotten to keep everything shipshape. "A data structure is only as strong as its weakest link."

The *sanitycheck* routine is designed to print out most discrepancies between my assumptions and the true state of affairs. I used it to locate lapses when this program was being debugged, and it remains as testimony to the most vital structural assumptions that are being made.

⟨ Subroutines 9 ⟩ +≡

```
⟨ Declare the recursive routine bondsanity 16 ⟩;
int sanitycheck(int flags)
{
  register Vertex *u, *v;
  register Arc *e;
  register Bond *b;
  register int k, n;
  int bugs = 0;
  if (flags & 1) ⟨ Do a sanity check on the graph 13 ⟩;
  if (flags & 2) ⟨ Do a sanity check on the action list 15 ⟩;
  if (flags & 4) ⟨ Do a sanity check on the tree list 17 ⟩;
  return bugs;
}
```

13. ⟨ Do a sanity check on the graph 13 ⟩ ≡

```
{
  for (n = 0, v = gg→vertices; v < gg→vertices + gg→n; v++)
    if (v→mark ≥ 0) ⟨ Do a sanity check on v's bond list 14 ⟩;
}
```

This code is used in section 12.

14. Some of the "bugs" detected in this routine are, of course, harmless in certain contexts. My goal is to call attention to things that might be unexpected, but to keep going in any case.

```
⟨ Do a sanity check on v's bond list 14 ⟩ ≡
  {
    n++;
    if (v→mark) bugs++, printf("Vertex␣%s␣is␣marked\n", v→name);
    if ((v→deg < 3 ∧ v→deg ≠ 0) ∨ v→deg ≥ gg→n)
      bugs++, printf("Vertex␣%s␣has␣degree␣%d\n", v→name, v→deg);
    for (k = 0, e = v→arcs; k ≤ v→deg; k++, e = e→next) {
      if (e→prev→next ≠ e ∨ e→next→prev ≠ e)
        bugs++, printf("Link␣failure␣at␣vertex␣%s,␣bond␣%d\n", v→name, k);
      if (k > 0) {
        b = (Bond *) e→bond;
        if (b ≤ bondbase ∨ b > bondbase + bondcount)
          bugs++, printf("Vertex␣%s␣has␣bad␣bond␣%d\n", v→name, k);
        else if (b→lhist ≠ e ∧ b→lhist ≠ mate(e))
          bugs++, printf("Bond␣%d␣has␣bad␣lhist␣pointer\n", bn(b));
        if (mate(e)→bond ≠ e→bond) bugs++, printf("Vertex␣%s␣has␣unmated␣bond␣%d\n", v→name, k);
        if (mate(e)→tip ≠ v)
          bugs++, printf("Vertex␣%s's␣bond␣%d␣has␣wrong␣mate␣tip\n", v→name, k);
        u = e→tip;
        if (¬u) bugs++, printf("Vertex␣%s␣has␣bad␣tip␣%d\n", v→name, k);
        else if (u→mark < 0)
          bugs++, printf("Vertex␣%s␣points␣to␣deleted␣vertex␣%s\n", v→name, u→name);
      }
    }
    if (e ≠ v→arcs) bugs++, printf("Vertex␣%s␣has␣more␣than␣%d␣bonds\n", v→name, v→deg);
  }
```

This code is used in section 13.

15. The action list is essentially a forest of bonds, in preorder.

```
⟨ Do a sanity check on the action list 15 ⟩ ≡
  {
    if (action→left→right ≠ action ∨ action→right→left ≠ action)
      bugs++, printf("Link␣failure␣at␣head␣of␣action␣list\n");
    for (b = action→right; b ≠ action; b = b→right) {
      if (b→val ≡ 1 ∧ (¬b→up ∨ b→up→down ≠ b))
        bugs++, printf("Bond␣%d␣isn't␣properly␣in␣the␣tree␣list\n", bn(b));
      if (b→lsib ∨ b→rsib) bugs++, printf("Top␣level␣bond␣%d␣has␣siblings\n", bn(b));
      if (isleaf(b)) bugs++, printf("Leaf␣%d␣is␣in␣the␣action␣list\n", bn(b));
      else {
        bugs += bondsanity(b);
        b = b→scope;
      }
    }
  }
```

This code is used in section 12.

16. ⟨ Declare the recursive routine *bondsanity* 16 ⟩ ≡

```
int bondsanity(Bond *b)
{
  int bugs = 0;
  register Bond *a, *extent;
  register int j, k;
  extent = b;
  if (b→left→right ≠ b ∨ b→right→left ≠ b) bugs++, printf("Link failure at bond %d\n", bn(b));
  for (a = b→lchild, j = k = 0; a ≠ b→lchild ∨ k ≡ 0; a = a→rsib, k++) {
    if (a ≤ bondbase ∨ a > bondbase + bondcount) {
      bugs++, printf("Bond %d has a child out of range\n", bn(b));
      break;
    }
    if (a→lsib→rsib ≠ a ∨ a→rsib→lsib ≠ a)
      bugs++, printf("Sibling link failure at bond %d\n", bn(a));
    if (a ≡ b→des) j = 1;
    else if (a→val ≠ b→typ) bugs++, printf("Bond %d should have value %d\n", bn(a), b→typ);
    if (¬isleaf(a)) {
      if (a→left ≠ extent)
        bugs++, printf("Preorder failure: bond %d doesn't follow %d\n", bn(a), bn(extent));
      bugs += bondsanity(a);
      extent = a→scope;
    }
  }
  if (¬j) bugs++, printf("Bond %d doesn't designate any of its children\n", bn(b));
  else if (b→done ≠ b→des→lsib)
    bugs++, printf("Bond %d should be done at %d\n", bn(b), bn(b→des→lsib));
  if (b→scope ≠ extent) bugs++, printf("Bond %d should have scope %d\n", bn(b), bn(extent));
  return bugs;
}
```

This code is used in section 12.

17. If *flags* & 1, we've computed the number n of current vertices.

⟨ Do a sanity check on the tree list 17 ⟩ ≡

```
{
  if (tree→up→down ≠ tree ∨ tree→down→up ≠ tree)
    bugs++, printf("Link failure at head of tree list\n");
  for (b = tree→down, k = 0; b ≠ tree; k++, b = b→down) {
    if (b→up→down ≠ b ∨ b→down→up ≠ b)
      bugs++, printf("Link failure in tree list at bond %d\n", bn(b));
    if (b→val ≠ 1) bugs++, printf("Bond %d in the tree list has value 0\n", bn(b));
  }
  if ((flags & 1) ∧ (k ≠ n − 1 − (inbond ≡ Λ)))
    bugs++, printf("The tree list holds %d bonds, not %d\n", k, n − 1 − (inbond ≡ Λ));
}
```

This code is used in section 12.

18. Graph preparation. At the beginning, we want to make sure that the given graph is truly undirected, and that we can find mates of its edges using the infamous *edge_trick*. We also need to change its representation from singly linked edges to doubly linked bonds, and to compute vertex degrees, as well as to find an initial spanning tree.

All of these things can be done easily with a single search of the graph. The following program sets $v\rightarrow mark = 2$ for each vertex that has been seen, and keeps a sequential stack of vertices that have been seen but not yet explored. (The search has aspects of both bread-first and depth-first approaches: When we explore a vertex we see all of its successors before exploring another, but we select new vertices for exploration in the last-seen-first-explored manner.)

Why is the *mark* field set to 2? Because any nonzero value will do, and it turns out that we'll want to set it to 2 shortly after this part of the program is done.

```
#define stack(k)  (g→vertices + (k))→z.V    /* utility field z is used for a stack */
#define mark  v.I    /* utility field v of each vertex holds a mark */
```

⟨ Check the graph for validity and prepare it for action 18 ⟩ ≡

```
for (v = g→vertices + 1; v ≤ g→vertices + g→n; v++) v→mark = 0;
if (verbose) printf("Graph␣%s␣has␣the␣following␣edges:\n", g→id);
v = g→vertices, stack(0) = v, k = 1, v→mark = 2, n = 1, m = 0;
while (k) {    /* k is the number of items on the stack */
  o, v = stack(−−k);
  f = gb_virgin_arc( );
  f→next = v→arcs;    /* the new header node */
  for (v→deg = 0, e = v→arcs, v→arcs = f; e; v→deg++, f = e, e = e→next) {
  looky: u = e→tip;
    if (u ≡ v) {    /* self-loops are silently ignored */
      f→next = e = e→next;
      if (¬e) break;
      goto looky;
    }
    e→prev = f;
    if (mate(e)→tip ≠ v) {
      fprintf(stderr, "Graph␣%s␣has␣an␣arc␣from␣%s␣to␣%s,\n", g→id, u→name, v→name);
      fprintf(stderr, "␣but␣the␣edge␣trick␣doesn't␣find␣the␣opposite␣arc!\n");
      exit(−3);
    }
    if (o, ¬e→bond) {
      ooo, m++, b = bondbase + m, e→bond = mate(e)→bond = (Arc *) b, b→lhist = e;
      if (verbose) printf("␣%d:␣%s␣--␣%s\n", m, v→name, u→name);
    }
    if (o, ¬u→mark) {
      ooo, u→mark = 2, stack(k++) = u, b = (Bond *) e→bond, b→val = 1;
      ooo, n++, b→up = tree→up, tree→up→down = b, tree→up = b, b→down = tree;
    }
  }
  v→arcs→prev = f, f→next = v→arcs;    /* complete the double linking */
}
if (n < g→n) {
  fprintf(stderr, "Oops,␣the␣graph␣isn't␣connected!\n");
  exit(−4);
}
o, topleaf = bondbase + m, bondcount = m;
```

This code is used in section 2.

19. Reduction. Let's start to write real code now. This program reaches a crucial step when we get to the label called *reduce*, corresponding roughly to the point where GRAYSPAN gets to its label called *enter*.

When *reduce* is reached, we're in the following state:

1) Bonds $a_1 \dots a_l$ of the current graph have been shrunk, represented as the array $aa(1)$ through $aa(l)$. After we've found all spanning trees in the shrunken graph, we'll want to unshrink them.
2) The *tree* list specifies a set of bonds that form a spanning tree on the current graph, if $inbond \neq \Lambda$, or a near-spanning tree if $inbond = \Lambda$. The next spanning tree we generate is supposed to include all of those bonds.
3) If the current graph contains any parallel edges, they are adjacent to vertices v for which $v\rightarrow mark = 2$.
4) If the current graph contains any vertices v of degree less than 3, we have $v\rightarrow mark > 0$.
5) All marked vertices appear on the stack, which currently holds k items.
6) All current bonds and subbonds appear in locations $bondbase + 1$ through $bondbase + bondcount$. The ones created before reaching level l are less than or equal to $bondbase + bonds(l)$.

Our job is to zero out the marks, continuing to reduce the graph either by forming more complex bonds or by shrinking bonds of the tree list, until the graph finally becomes trivial.

```
#define bonds(l)  (g→vertices + l)→y.I      /* utility field y of the vertex array */
#define aa(l)  (g→vertices + l)→x.A      /* utility field x holds a_l */
⟨ Reduce the graph until it's trivial 19 ⟩ ≡
reduce: while (k) {
    o, v = stack(--k);
    if (o, v→mark > 1) ⟨ Parallelize duplicate bonds from v 20 ⟩;
    if (o, v→deg < 3) ⟨ Eliminate v, then goto trivialgraph if only one vertex is left 21 ⟩
    else o, v→mark = 0;
  }
      /* now all relevant marks are zero and the graph still isn't trivial */
  o, l++, bonds(l) = bondcount;
  if (extraverbose) printf("Entering level %d\n", l);
  oooo, b = tree→down, tree→down = b→down, tree→down→up = tree;
  oo, e = b→lhist, aa(l) = e;      /* we have b→lhist→bond = (Arc *) b */
  ⟨ Shrink bond e 24 ⟩;
  goto reduce;
```

This code is used in section 25.

20. `#define dup  w.A      /* utility field w points to a previous edge */`

```
⟨ Parallelize duplicate bonds from v 20 ⟩ ≡
  {
    for (oo, e = v→arcs→next; o, e→tip; o, e = e→next) o, e→tip→dup = Λ;
    for (e = v→arcs→next; o, e→tip; o, e = e→next) {
      u = e→tip;
      if (o, u→dup) {
        makeparallel(u→dup, e);      /* create a new parallel bond */
        if (o, ¬u→mark) oo, u→mark = 1, stack(k++) = u;
      } else o, u→dup = e;
    }
  }
```

This code is used in section 19.

21. A deleted vertex is marked -1, for debugging purposes only.

```
⟨ Eliminate v, then goto trivialgraph if only one vertex is left 21 ⟩ ≡
  {
    v→mark = −1;
    if (v→deg ≡ 2) ⟨ Serialize the bonds from v 23 ⟩
    else if (v→deg ≡ 1) ⟨ Require the bond from v 22 ⟩
    else {
      v→mark = 0;      /* the last vertex doesn't go away */
      goto trivialgraph;      /* v→deg = 0, hence no other vertices remain */
    }
  }
```

This code is used in section 19.

22. If the single bond touching v isn't in the tree list, we know that the tree list must specify only a near-spanning tree. So we set *inbond*, which will complete a spanning tree later. And we eliminate v; thus the tree list henceforth is a spanning tree on the remaining vertices.

```
⟨ Require the bond from v 22 ⟩ ≡
  {
    ooo, e = v→arcs→next, b = (Bond *) e→bond;
    o, u = e→tip;
    deletebonds(mate(e), Λ);      /* remove mate(e) and decrease u→deg */
    if (o, b→val) oooo, b→up→down = b→down, b→down→up = b→up;
    else {
      if (inbond) {
        fprintf(stderr, "I've␣goofed␣(inbond␣doubly␣set)!\n");
        exit(−5);
      }
      inbond = b;
    }
    if (o, ¬u→mark) oo, u→mark = 1, stack(k++) = u;
  }
```

This code is used in section 21.

23. A subtle point arises here: We might be serializing two bonds not in the spanning tree, if v happens to be the only vertex not reachable from the current near-spanning tree. In that case we want to set *inbond* to the subbond of the new series bond that will *not* be designated by *makeseries*.

```
⟨ Serialize the bonds from v 23 ⟩ ≡
  {
    oooooo, e = v→arcs→next, f = e→next, u = e→tip, w = f→tip, b = (Bond *) e→bond;
    if (o, ¬b→val) {
      o, b = (Bond *) f→bond;
      if (o, ¬b→val) {
        if (inbond) {
          fprintf(stderr, "I've␣doubly␣goofed␣(inbond␣set)!\n");
          exit(−6);
        }
        inbond = b;
      }
    }
    makeseries(e, f);      /* create a new series bond */
    if (o, u→mark) o, u→mark = 2;
    else oo, u→mark = 2, stack(k++) = u;
    if (o, w→mark) o, w→mark = 2;
    else oo, w→mark = 2, stack(k++) = w;
  }
```

This code is used in section 21.

24. At this point the graph is irreducible, so v appears only once in u's list.

```
⟨ Shrink bond e 24 ⟩ ≡
  oo, u = e→tip, v = mate(e)→tip;
  for (oo, f = u→arcs→next; o, f→tip; o, f = f→next)
    if (f ≡ mate(e)) delete(f);
    else o, mate(f)→tip = v;
  delete(e);
  ooo, v→deg += u→deg − 2;
  o, ee = v→arcs;      /* now f = u→arcs */
  oooo, f→prev→next = ee→next, ee→next→prev = f→prev;
  ooo, f→next→prev = ee, ee→next = f→next;
  if (extraverbose) printf("␣shrinking␣%d;␣now␣%s␣has␣degree␣%d\n", ebn(e), v→name, v→deg);
  oo, u→mark = −1, v→mark = 2, k = 1, stack(0) = v;
```

This code is used in section 19.

25. The main algorithm. Now that we understand reduction, we're ready to complete the GRAYSPAN portion of this program, except for low-level details.

```
⟨ Generate all spanning trees 25 ⟩ ≡
  {
    for (k = 0; k < n; k++) o, stack(k) = g→vertices + k;      /* all vertices are initially suspect */
    o, b = tree→up, b→val = 0;      /* we delete the last edge of the preliminary tree */
    oo, tree→up = b→up, tree→up→down = tree;
    if (verbose) {
      printf("Start␣with␣the␣near-spanning␣edges");
      for (b = tree→down; b ≠ tree; b = b→down) printf("␣%d", bn(b));
      printf("\n");
    }
    l = 0;
    ⟨ Reduce the graph until it's trivial 19 ⟩;
  trivialgraph: ⟨ Obtain a new spanning tree by changing outbond and inbond 30 ⟩;
    ⟨ Do the SPSPAN algorithm on the action list 47 ⟩;
    while (l) {
      ⟨ Undo all changes to the graph since entering level l 29 ⟩;
      o, e = aa(l);
      ⟨ Unshrink bond e 26 ⟩;
      l−−;
      ⟨ If e is not a bridge, delete it, set outbond = e→bond, and goto reduce 27 ⟩;
      oooooo, b = (Bond *) e→bond, b→up = tree→up, tree→up = b→up→down = b, b→down = tree;
    }
  }
```

This code is used in section 1.

26. After unshrinking, the graph will still be irreducible.

```
⟨ Unshrink bond e 26 ⟩ ≡
  ooooo, u = e→tip, v = mate(e)→tip, v→deg −= u→deg − 2;
  oo, ee = v→arcs, f = u→arcs;
  oooo, ee→next = f→prev→next, ee→next→prev = ee;
  ooo, f→prev→next = f, f→next→prev = f;
  for (o, f = f→next; o, f→tip; o, f = f→next) o, mate(f)→tip = u;
  undelete(e), undelete(mate(e));
  if (extraverbose) printf("␣unshrinking␣%d;␣now␣%s␣has␣degree␣%d\n", ebn(e), v→name, v→deg);
  u→mark = 0;      /* it was −1 */
```

This code is used in section 25.

27. We use a field *bfs* that shares space with *mark*, because *mark* is zero in all relevant vertices at this time.

```
#define bfs v.V /* link for the breadth-first search: nonnull if vertex seen */
⟨ If e is not a bridge, delete it, set outbond = e→bond, and goto reduce 27 ⟩ ≡
  oo, u = e→tip, v = mate(e)→tip;
  for (o, u→bfs = v, w = u; u ≠ v; o, u = u→bfs) {
    for (oo, f = u→arcs→next; o, f→tip; o, f = f→next)
      if (o, f→tip→bfs ≡ Λ) {
        if (f→tip ≡ v) {
          if (f ≠ mate(e)) ⟨ Nullify all bfs links, delete e, and goto reduce 28 ⟩;
        } else oo, f→tip→bfs = v, w→bfs = f→tip, w = f→tip;
      }
  }
  if (extraverbose) printf("Leaving level %d: %d is a bridge\n", l + 1, ebn(e));
  for (o, u = e→tip; u ≠ v; o, u→bfs = Λ, u = w) o, w = u→bfs;
```

This code is used in section 25.

28. We have discovered that e is not a bridge.

```
⟨ Nullify all bfs links, delete e, and goto reduce 28 ⟩ ≡
  {
    for (o, u = e→tip; u ≠ v; o, u→bfs = Λ, u = w) o, w = u→bfs;
    outbond = (Bond *)(e→bond);
    deletebonds(e, mate(e));
    oooo, k = 2, stack(0) = e→tip, stack(1) = mate(e)→tip;
    oo, e→tip→mark = mate(e)→tip→mark = 1;
    goto reduce;
  }
```

This code is used in section 27.

29. ⟨ Undo all changes to the graph since entering level l 29 ⟩ ≡

```
o; while (bondcount > bonds(l)) unbuildbond( );
```

This code is used in section 25.

30. When the program reaches *trivialgraph*, the variables *outbond* and *inbond* point to bonds whose values are to become 0 and 1, respectively, in the next spanning tree.

(Exception: On the first iteration, *outbond* is null and we've already printed $n-2$ edges of the first spanning tree, as sort of a pump-priming process.)

Note that *inbond* might not be a top-level bond, because of the subtle point mentioned earlier. But *outbond* always at the top level, because it was removed from the graph when *outbond* was set.

```
⟨ Obtain a new spanning tree by changing outbond and inbond 30 ⟩ ≡
  count++;
  if (verbose) printf("%.15g:", count);
  if (outbond) {
    for (b = outbond; ; o,b = b→des) {
      o,b→val = 0;
      if (isleaf(b)) break;
    }
    if (verbose) printf("-%d", bn(b));
  }
  if (¬inbond) {
    fprintf(stderr, "Internal␣error␣(no␣inbond)!\n");
    exit(-7);
  }
  for (b = inbond; ; o,b = b→des) {
    o,b→val = 1;
    if (isleaf(b)) break;
  }
  if (verbose) {
    printf("+%d", bn(b));
    if (extraverbose) {
      printf("␣(");
      for (b = bondbase + 1; b ≤ topleaf; b++)
        if (b→val) printf("␣%d", bn(b));
      printf("␣)\n");
    } else printf("\n");
  }
  inbond = outbond = Λ;
```

This code is used in section 25.

31. Construction. Reducing the main graph means constructing more bonds.

The down side of having a complex data structure is that we have to do tedious maintenance work when conditions change. The following part of the program was least fun to write, and it is the most likely place where silly errors might have crept in. But I gritted my teeth and I think I've gotten the job done. (As Anne Lamott would say, this part of the program was written "bird by bird.")

The two subroutines *makeseries* and *makeparallel* are each called only once, so I needn't have made them subroutines. They do share a lot of common code, however, so I've simplified my task by writing a combined routine that handles both cases. Hopefully this will reduce the chances of error. But mems are computed as if the subroutine had been expanded inline and customized for the cases $t = 0$ and $t = 1$.

```
#define makeseries(e, f)  buildbond(1, e, f)
#define makeparallel(e, f)  buildbond(0, e, f)
⟨ Subroutines 9 ⟩ +≡
  void buildbond(int t, Arc *aa, Arc *bb)
  {
    register Bond *a, *b, *c, *d;
    register int at, av, bt, bv;
    register Vertex *u, *v;
    register Arc *ee;      /* used by the delete macro */
    bondcount++, c = bondbase + bondcount;
    oooo, c→typ = t, c→lhist = aa, c→rhist = bb, c→focus = c;
    oo, a = (Bond *) aa→bond, b = (Bond *) bb→bond;
    oo, av = a→val, at = a→typ, bv = b→val, bt = b→typ;
    if (t) ⟨ Update aa and bb for a new series bond 32 ⟩
    else ⟨ Update aa and bb for a new parallel bond 33 ⟩;
    oo, mate(bb)→bond = aa→bond;      /* remember the bond that's going away */
    oo, aa→bond = mate(aa)→bond = (Arc *) c;      /* c→lhist→bond points to c */
    if (isleaf(a))
      if (isleaf(b)) ⟨ Build leaf with leaf 34 ⟩
      else ⟨ Build leaf with branch 35 ⟩
    else if (isleaf(b)) ⟨ Build branch with leaf 36 ⟩
    else ⟨ Build branch with branch 37 ⟩;
    if (av) oooo, a→up→down = a→down, a→down→up = a→up;
    if (bv) oooo, b→up→down = b→down, b→down→up = b→up;
    if (av ≡ bv ∨ bv ≡ t)
      if (isleaf(a) ∨ at ≠ t) o, c→des = a;
      else oo, c→des = a→des;
    else if (isleaf(b) ∨ bt ≠ t) o, c→des = b;
    else oo, c→des = b→des;
    oooo, c→val = c→des→val, c→done = c→des→lsib;
    if (c→val) ooooo, c→up = tree→up, tree→up = c→up→down = c, c→down = tree;
  }
```

32. The new series bond has to agree with its mate. So we move the arc node *aa* from one list to another.

⟨ Update *aa* and *bb* for a new series bond 32 ⟩ ≡

```
{
  if (extraverbose) printf("␣%d=series(%d,%d)␣between␣%s␣and␣%s\n", bn(c), bn(a), bn(b),
      aa→tip→name, bb→tip→name);
  oooo, ee = mate(bb), aa→prev = ee→prev, aa→next = ee→next;
  oo, aa→prev→next = aa→next→prev = aa;
  oo, mate(aa)→tip = bb→tip;
}
```

This code is used in section 31.

33. ⟨ Update *aa* and *bb* for a new parallel bond 33 ⟩ ≡

```
{
  oo, u = aa→tip, v = mate(aa)→tip;
  if (extraverbose)
    printf("␣%d=parallel(%d,%d)␣between␣%s␣and␣%s\n", bn(c), bn(a), bn(b), u→name, v→name);
  oooo, delete(bb), delete(mate(bb)), u→deg−−, v→deg−−;
}
```

This code is used in section 31.

34. ⟨ Build leaf with leaf 34 ⟩ ≡

```
{
  oooo, a→lsib = a→rsib = b, b→lsib = b→rsib = a;
  o, c→lchild = a;
  ooo, c→right = action→right, c→right→left = c;
  oo, c→left = action, action→right = c;
  o, c→scope = c;
}
```

This code is used in section 31.

35. ⟨ Build leaf with branch 35 ⟩ ≡

```
{
  if (bt ≠ t) {
    oooo, a→lsib = a→rsib = b, b→lsib = b→rsib = a;
    ooo, c→right = b, c→scope = b→scope;
  } else {
    oooo, a→rsib = b→lchild, a→lsib = b→lchild→lsib;
    oo, a→rsib→lsib = a→lsib→rsib = a;
    oooo, c→right = b→right, c→scope = (b→scope ≡ b ? c : b→scope);
  }
  o, c→lchild = a;
  oo, c→left = b→left;
  oo, c→left→right = c→right→left = c;
}
```

This code is used in section 31.

36. ⟨ Build branch with leaf 36 ⟩ ≡

```
{
  if (at ≠ t) {
    oooo, a→lsib = a→rsib = b, b→lsib = b→rsib = a;
    oooo, c→right = c→lchild = a, c→scope = a→scope;
  } else {
    oooo, b→rsib = a→lchild, b→lsib = a→lchild→lsib;
    oo, b→rsib→lsib = b→lsib→rsib = b;
    oooo, c→right = a→right, c→lchild = a→lchild;
    oo, c→scope = (a→scope ≡ a ? c : a→scope);
  }
  oo, c→left = a→left;
  oo, c→left→right = c→right→left = c;
}
```

This code is used in section 31.

37. ⟨ Build branch with branch 37 ⟩ ≡

```
{
  ooo, d = a→scope, c→rsave = d→right;
  oooo, a→left→right = d→right, d→right→left = a→left;
  ooo, c→left = b→left, c→left→right = c;
  if (at ≠ t) {
    oo, c→lsave = a→left;
    ooo, c→lchild = a, c→right = a, a→left = c;
    if (bt ≠ t) {
      oooo, a→lsib = a→rsib = b, b→lsib = b→rsib = a;
      oo, d→right = b, b→left = d;
      ooo, c→scope = b→scope;
    } else {
      oooo, a→rsib = b→lchild, a→lsib = b→lchild→lsib;
      oo, a→lsib→rsib = a→rsib→lsib = a;
      ooo, d→right = b→right, d→right→left = d;
      oo, c→scope = (b→scope ≡ b ? d : b→scope);
    }
  } else if (bt ≠ t) {
    ooo, c→lchild = b→rsib = a→lchild;
    oo, b→lsib = a→lchild→lsib;
    oo, b→lsib→rsib = b→rsib→lsib = b;
    if (d ≡ a) o, c→right = b;
    else oooo, c→right = a→right, d→right = b, b→left = d;
    o, c→right→left = c;
    oo, c→scope = b→scope;
  } else {
    if (d ≡ a) oo, c→right = b→right;
    else ooooo, c→right = a→right, d→right = b→right, b→right→left = d;
    o, c→right→left = c;
    oo, c→lchild = a→lchild;
    oooo, d = a→lchild→lsib, a→lchild→lsib = b→lchild→lsib;
    ooo, b→lchild→lsib→rsib = a→lchild, b→lchild→lsib = d, d→rsib = b→lchild;
    oo, c→scope = (b→scope ≡ b ? (a→scope ≡ a ? c : a→scope) : b→scope);
  }
}
```

This code is used in section 31.

38. A much simpler subroutine is used to record the fact that one or two bonds are temporarily being deleted from the graph.

```
⟨ Subroutines 9 ⟩ +≡
  void deletebonds(Arc *aa, Arc *bb)
  {
    register Bond *a, *c;
    register Vertex *u, *v;
    register Arc *ee;
    bondcount++, c = bondbase + bondcount;
    ooo, c→typ = 2, c→lhist = aa, c→rhist = bb;
    oo, u = mate(aa)→tip, a = (Bond *) aa→bond;
    o, v = (bb ? mate(bb)→tip : aa→tip);
    if (extraverbose) printf("␣%d:␣deleting␣bond␣%d␣between␣%s␣and␣%s%s\n", bn(c), bn(a), u→name,
          bb ? "" : "endpoint␣", v→name);
    oo, delete(aa), u→deg--;
    if (bb) oo, delete(bb), v→deg--;
  }
```

39. Deconstruction. Every change to the graph after we first reach level 1 must be undone again before we finish the program. But fortunately the changes are always made in a strictly last-done-first-undone manner. Therefore we can use the "dancing links" principle to exploit of the fact that pointers within deleted structures still retain the values to which they should be restored.

Thus I could write *unbuildbond* by just looking at the code for *buildbond* and unchanging everything that it changed. (These remarks apply only to changes to *prev*, *next*, *bond*, *lchild*, *lsib*, *rsib*, *scope*, *left*, and *right*, which govern the state of the bonds. The *val*, *des*, *done*, *up*, and *down* fields vary dynamically with the spanning tree and should not be restored blindly. The *focus* fields always point to self when construction or deconstruction is being done.)

(The programming task still was tedious and error-prone though; sigh.)

```
⟨ Subroutines 9 ⟩ +≡
  void unbuildbond( )
  {
    register Bond *a, *b, *c, *d;
    register int t, at, bt;
    register Vertex *u, *v;
    register Arc *aa, *bb, *ee;      /* used by the undelete macro */
    c = bondbase + bondcount, bondcount--;
    ooo, t = c→typ, aa = c→lhist, bb = c→rhist;
    if (t > 1) ⟨ Restore one or two deleted bonds and return 46 ⟩
    oo, a = (Bond *) mate(bb)→bond, b = (Bond *) bb→bond;
    o, mate(bb)→bond = (Arc *) b;
    oo, aa→bond = mate(aa)→bond = (Arc *) a;
    oo, at = a→typ, bt = b→typ;
    if (t) ⟨ Downdate aa and bb from an old series bond 40 ⟩
    else ⟨ Downdate aa and bb from an old parallel bond 41 ⟩;
    if (isleaf(a))
      if (isleaf(b)) ⟨ Unbuild leaf with leaf 42 ⟩
      else ⟨ Unbuild leaf with branch 43 ⟩
    else if (isleaf(b)) ⟨ Unbuild branch with leaf 44 ⟩
    else ⟨ Unbuild branch with branch 45 ⟩;
    if (c→val) oooo, c→up→down = c→down, c→down→up = c→up;
    if (a→val) ooooo, a→up = tree→up, tree→up = a→up→down = a, a→down = tree;
    if (b→val) ooooo, b→up = tree→up, tree→up = b→up→down = b, b→down = tree;
  }
```

40. ⟨ Downdate *aa* and *bb* from an old series bond 40 ⟩ ≡

```
  {
    undelete(mate(bb));      /* now ee = mate(bb) */
    oo, v = ee→tip, mate(aa)→tip = v, v→mark = 0;
    ooo, aa→prev = v→arcs, aa→next = bb;
    if (extraverbose) printf("␣removing␣%d=series(%d,%d)␣between␣%s␣and␣%s\n", bn(c), bn(a), bn(b),
        aa→tip→name, bb→tip→name);
  }
```

This code is used in section 39.

41. ⟨ Downdate aa and bb from an old parallel bond 41 ⟩ ≡

```
{
  oo, u = aa→tip, v = mate(aa)→tip;
  oooo, undelete(mate(bb)), undelete(bb), u→deg ++, v→deg ++;
  if (extraverbose) printf("␣removing␣%d=parallel(%d,%d)␣between␣%s␣and␣%s\n", bn(c), bn(a),
      bn(b), u→name, v→name);
}
```

This code is used in section 39.

42. Sibling links of bonds at the top level are never examined. This program nullifies them only to be tidy and possibly catch errors, but no mems are counted.

⟨ Unbuild leaf with leaf 42 ⟩ ≡

```
{
  a→lsib = a→rsib = b→lsib = b→rsib = Λ;
  ooo, action→right = c→right, action→right→left = action;
}
```

This code is used in section 39.

43. The nonobvious part here is the calculation of *des* when a disabled bond reenters the picture.

⟨ Unbuild leaf with branch 43 ⟩ ≡

```
{
  if (bt ≠ t) {
    a→lsib = a→rsib = b→lsib = b→rsib = Λ;
    oo, b→left = c→left;
  } else {
    if (o, c→des ≡ a) o, b→val = bt;
    else oo, b→val = c→val;
    if (b→val ≠ bt) b→des = c→des;      /* mem is charged below */
    oooo, b→lchild→lsib = a→lsib, a→lsib→rsib = b→lchild;
    ooo, b→done = b→des→lsib;
    a→lsib = a→rsib = Λ;
    oo, b→right→left = b;
  }
  oo, b→left→right = b;
}
```

This code is used in section 39.

44. ⟨ Unbuild branch with leaf 44 ⟩ ≡

```
{
  if (at ≠ t) {
    a→lsib = a→rsib = b→lsib = b→rsib = Λ;
    oo, a→left = c→left;
  } else {
    if (o, c→des ≡ b) o, a→val = at;
    else oo, a→val = c→val;
    if (a→val ≠ at) a→des = c→des;
    oooo, a→lchild→lsib = b→lsib, b→lsib→rsib = a→lchild;
    ooo, a→done = a→des→lsib;
    b→rsib = b→lsib = Λ;
    oo, a→right→left = a;
  }
  oo, a→left→right = a;
}
```

This code is used in section 39.

45. If we previously combined two series bonds into a larger series bond, or two parallel bonds into a larger parallel bond, we may have to search through the children in order to figure out where *c→des* lies.

```
⟨ Unbuild branch with branch 45 ⟩ ≡
  {
    ooo, d = a→scope, d→right = c→rsave;
    if (at ≠ t) {
      oo, a→left = c→lsave;
      if (bt ≠ t) {
        a→lsib = a→rsib = b→lsib = b→rsib = Λ;
        ooo, b→left = c→left, b→left→right = b;
      } else {
        if (o, c→des ≡ a) o, b→val = bt;
        else oo, b→val = c→val;
        if (b→val ≠ bt) b→des = c→des;
        oooo, b→lchild→lsib = a→lsib, a→lsib→rsib = b→lchild;
        ooo, b→done = b→des→lsib;
        oooo, b→left→right = b→right→left = b;
        a→lsib = a→rsib = Λ;
      }
    } else {
      if (d ≠ a) oo, a→right→left = a;
      if (bt ≠ t) {
        if (o, c→des ≡ b) o, a→val = at;
        else oo, a→val = c→val;
        if (a→val ≠ at) a→des = c→des;
        oooo, a→lchild→lsib = b→lsib, b→lsib→rsib = a→lchild;
        ooo, a→done = a→des→lsib;
        oooo, b→left = c→left, b→left→right = b;
        b→lsib = b→rsib = Λ;
      } else {
        if (c→val ≡ t) oo, a→val = b→val = t;
        else for (ooo, d = c→des; ; o, d = d→lsib)
            if (d ≡ a→lchild) {
              ooo, a→val = 1 − t, b→val = t, a→des = c→des; break;
            } else if (d ≡ b→lchild) {
              ooo, b→val = 1 − t, a→val = t, b→des = c→des; break;
            }
        oooooo, d = a→lchild→lsib, a→lchild→lsib = b→lchild→lsib, b→lchild→lsib = d;
        oo, d→rsib = b→lchild, a→lchild→lsib→rsib = a→lchild;
        ooooo, a→done = a→des→lsib, b→done = b→des→lsib;
        d = a→scope;
        oooo, b→left→right = b→right→left = b;
      }
    }
    oooo, a→left→right = a, d→right→left = d;
  }
```

This code is used in section 39.

46. ⟨ Restore one or two deleted bonds and **return** 46 ⟩ ≡

```
{
  oo, u = mate(aa)→tip, a = (Bond *) aa→bond;
  o, v = (bb ? mate(bb)→tip : aa→tip);
  if (bb) oo, undelete(bb), v→deg++;
  else v→mark = 0;      /* for debugging, remove its negative mark */
  oo, undelete(aa), u→deg++;
  if (extraverbose) printf("␣restoring␣bond␣%d␣between␣%s␣and␣%s%s\n", bn(a), u→name,
      bb ? "" : "endpoint␣", v→name);
  if (a→val) ooooo, a→up = tree→up, tree→up = a→up→down = a, a→down = tree;
  return;
}
```

This code is used in section 39.

47. Pulsing the action list. OK, all the hard stuff is done; only one more piece of the program needs to be put in place. And from SPSPAN, we know how to do the remaining job nicely.

```
#define easy(b)  o,b→typ ≡ b→val
⟨ Do the SPSPAN algorithm on the action list 47 ⟩ ≡
  while (1) {
    register Bond *q,*l,*r;
    for (o,r = action→left; easy(r); o,r = r→left) ;      /* find the rightmost uneasy node */
    oo,b = r→focus,r→focus = r;      /* steps (1) and (3) */
    if (b ≡ action) break;
    ⟨ Change b→des and visit a new spanning tree 49 ⟩;
    if (o,b→des ≡ b→done) ⟨ Passivate b 48 ⟩;
  }
```

This code is used in section 25.

48. All uneasy nodes to the right of b are now active, and l is the former b→*des*.

```
⟨ Passivate b 48 ⟩ ≡
  {
    o,b→done = l;
    for (o,l = b→left; easy(l); o,l = l→left) ;      /* find the first uneasy node to the left */
    ooo,b→focus = l→focus,l→focus = l;
  }
```

This code is used in section 47.

49. If the user has asked for *verbose* output, we print only the edge that has entered the spanning tree and the edge that has left.

```
⟨ Change b→des and visit a new spanning tree 49 ⟩ ≡
  count++;
  oo,l = b→des,r = l→rsib;
  o,k = b→val;      /* k = l→val ≠ r→val */
  for (q = l; ; o,q = q→des) {
    o,q→val = k ⊕ 1;
    if (isleaf(q)) break;
  }
  if (verbose) printf("%.15g:␣%c%d",count,k ? '-' : '+',bn(q));
  for (q = r; ; o,q = q→des) {
    o,q→val = k;
    if (isleaf(q)) break;
  }
  if (verbose) {
    printf("%c%d",k ? '+' : '-',bn(q));
    if (extraverbose) {
      printf("␣(");
      for (q = bondbase + 1; q ≤ topleaf; q++)
        if (q→val) printf("␣%d",bn(q));
      printf("␣)\n");
    } else printf("\n");
  }
  o,b→des = r;      /* "visiting" */
```

This code is used in section 47.

50. Note: I haven't proved the efficiency claim made in the opening paragraph. I don't have time to write the proof down now, sorry. But it is based on the fact that a connected graph on n vertices with all vertices of degree 3 or more always has more than $2^{n/2}$ spanning trees. Therefore the time that is spent checking for bridges, etc., which is polynomial in the number of vertices, is asymptotically less than the time needed to spew out the trees.

51. Index.

GRAYSPSPAN