November 24, 2020 at 13:23

1. Intro. This program was written (somewhat hastily) in order to experiment with an algorithm that generates all spanning trees of a given graph, changing only one edge at a time. Most of the basic ideas are adapted from Malcolm Smith's M.S. thesis, "Generating spanning trees" (University of Victoria, 1997), which also contains more complex variations that guarantee better asymptotic performance. I intend to experiment with those additional bells and whistles later.

The first command line argument is the name of a file that specifies an undirected graph in Stanford GraphBase SAVE_GRAPH format; the graph may have repeated edges, but it must not contain loops. 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

#include "gb_graph.h"
#include "gb_save.h"
  ⟨Preprocessor definitions⟩
  double mems;   /* memory references made */
  double count;   /* trees found */
  ⟨Subroutines 5⟩
  main(int argc, char *argv[])
  {
    ⟨Local variables 3⟩;
    ⟨Input the graph 2⟩;
    ⟨Initialize the algorithm 16⟩;
    ⟨Generate all spanning trees 7⟩;
    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(1);
  }
  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(−1);
  }
  n = g→n;
  ⟨Check the graph for validity and prepare it for action 4⟩;
```

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 k;   /* current integer of interest */
register Vertex *u, *v, *w;   /* current vertices of interest */
register Arc *e, *ee, *f, *ff;   /* current edges of interest */
```

See also section 8.

This code is used in section 1.

4. Graph preparation. While we're checking to see that the graph meets certain minimal standards, we might as well also compute the degree of each vertex, since our algorithm will be using that information. We also ensure that the SGB "edge trick" works on our computer.

In this program we deviate from normal conventions of the Stanford GraphBase by using a *doubly* linked list of arcs from each vertex v. Namely, $v\to arcs$ points to a header node h, and the arcs from v are $h\to next$, $h\to next\to next$, etc., until returning to h again. All arc nodes e in this list have $e\to next\to prev = e\to prev\to next = e$. The header node is distinguished by the property $h\to tip = \Lambda$.

The "length" of each edge is changed to an identifying number for use in printouts.

```
#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 mate(e) (edge_trick & (siz_t)(e) ? (e) − 1 : (e) + 1)
```

⟨Check the graph for validity and prepare it for action 4⟩ ≡

```
  if (verbose) printf("Graph␣%s␣has␣the␣following␣edges:\n", g→id);
  for (v = g→vertices, k = 0; v < g→vertices + n; v++) {
    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) {
      e→prev = f;
      u = e→tip;
      if (u ≡ v) {
        fprintf(stderr, "Oops,␣there's␣a␣loop␣from␣%s␣to␣itself!\n", v→name);
        exit(−3);
      }
      if (mate(e)→tip ≠ v) {
        fprintf(stderr, "Oops:␣There's␣an␣arc␣from␣%s␣to␣%s,\n", u→name, v→name);
        fprintf(stderr, "␣but␣the␣edge␣trick␣doesn't␣find␣the␣opposite␣arc!\n");
        exit(−4);
      }
      if (u > v) {
        e→len = mate(e)→len = ++k;
        if (verbose) printf("␣%d:␣%s␣--␣%s\n", k, v→name, u→name);
      }
    }
    v→arcs→prev = f, f→next = v→arcs;    /* complete the double linking */
    if (v→deg ≡ 0) {
      fprintf(stderr, "Graph␣%s␣has␣an␣isolated␣vertex␣%s!\n", g→id, v→name);
      exit(−5);
    }
  }
```

This code is used in section 2.

5. Here's something I might like to use when debugging.

⟨Subroutines 5⟩ ≡

```
  void print_arcs(Vertex *v)
  {
    register Arc *a;
    printf("Arcs␣leading␣from␣%s:\n", v→name);
    for (a = v→arcs→next; a→tip; a = a→next) printf("␣%d␣(to␣%s)\n", a→len, a→tip→name);
  }
```

See also section 17.

This code is used in section 1.

6. The method. Let G be a graph with $n > 1$ vertices. The basic idea of Smith's algorithm is to generate all spanning trees of G in such a way that the first one includes a given *near-tree*, namely a set of $n-2$ edges that don't contain a cycle. This task is easy if $n = 2$: We simply list all the edges.

If $n > 2$ and if the near-tree is $\{e_1, \ldots, e_{n-2}\}$, we proceed as follows: First form the graph $G \cdot e_1$ by shrinking edge e_1 (making its endpoints identical). All spanning trees of G that include e_1 are obtained by appending e_1 to a spanning tree of $G \cdot e_1$; so we proceed recursively to generate all spanning trees of $G \cdot e_1$, beginning with the near-tree $\{e_2, \ldots, e_{n-2}\}$. If no such trees exist, we stop; in this case $G \cdot e_1$ is not connected, so G is not connected and it has no spanning trees. Otherwise, suppose the last spanning tree found for $G \cdot e_1$ is $f_1 \ldots f_{n-2}$. Then we complete our task by deleting edge e_1 and generating all spanning trees in the resulting graph $G \setminus e_1$, starting with the near-tree $\{f_1, \ldots, f_{n-2}\}$.

7. This program implements the recursion directly by maintaining an array of edges $a_1 \ldots a_n$. When we enter level l, positions $a_1 \ldots a_{l-1}$ contain edges to include in the tree, and those edges have been shrunk in the current graph. Position a_l is effectively blank, and the remaining positions $a_{l+1} \ldots a_{n-1}$ contain the edges of a near-tree that should be part of the next spanning tree generated.

We don't delete an edge that is a "bridge," whose removal would disconnect the current graph. When a non-bridge edge e is deleted at level l, we set *change_e* $= e$. If the previously found spanning tree was $a_1 \ldots a_{n-1}$, the next tree found will be $a_1 \ldots a_{l-1} a_{l+1} \ldots a_{n-1} e'$ for some new edge e'; thus it will differ from its predecessor by removing edge *change_e* and replacing it with e'.

It's convenient to keep array element a_l in a utility field within the vertex array, represented by $aa(l)$. Another such utility field, $del(l)$, points to a stack of the edges deleted before coming to a bridge; edges on this list are linked together via a *link* field.

Experienced readers will not be shocked by the fact that this part of the program has a **goto** leading from one loop into another.

```
#define aa(l) (g→vertices + l)→z.A    /* the edge a_l */
#define del(l) (g→vertices + l)→y.A    /* the most recent edge deleted on level l */
#define link b.A    /* points from one edge to another */

⟨Generate all spanning trees 7⟩ ≡
  change_e = Λ;
  v = g→vertices;    /* this instruction needed only if n = 2 */
  for (l = 1; l < n − 1; l++) {
    o, del(l) = Λ;
  enter: ooo, e = aa(l + 1), u = e→tip, v = mate(e)→tip;
    if (oo, u→deg > v→deg) v = u, e = mate(e), u = e→tip;
    ⟨Shrink e by changing u to v 10⟩;
    o, aa(l) = e;
  }
  for (o, e = v→arcs→next; o, e→tip; o, e = e→next) {
    o, aa(l) = e;
    ⟨Produce a new spanning tree by changing change_e to e 9⟩;
    change_e = e;
  }
  for (l−−; l; l−−) {
    e = aa(l), u = e→tip, v = mate(e)→tip;
    ⟨Unshrink e by restoring u 11⟩;
    ⟨If e is not a bridge, delete it, set change_e = e, and goto enter 12⟩;
    ⟨Undelete all edges deleted since entering level l 15⟩;
  }
```

This code is used in section 1.

8. ⟨ Local variables 3 ⟩ +≡

```
register int l;   /* the current level */
Arc *change_e;   /* edge that may change next */
```

9. ⟨ Produce a new spanning tree by changing *change_e* to *e* 9 ⟩ ≡

```
count++;
if (verbose) {
  if (¬change_e ∨ extraverbose) {
    printf("%.15g:", count);
    for (k = 1; k < n; k++) printf("␣%d", aa(k)→len);
    if (extraverbose ∧ change_e) printf("␣(-%d+%d)\n", change_e→len, e→len);
    else printf("\n");
  } else printf("%.15g:␣-%d+%d\n", count, change_e→len, e→len);
}
```

This code is used in section 7.

10. To shrink an edge between u and v, we insert u's adjacency list into v's, changing all references to u into references to v; those references occur in *tip* fields of mates of the arcs in u's list.

We also delete all former edges between u and v, since those would otherwise become loops. Those former edges are linked together via their *link* fields, so that we can restore them later.

Note that $e\to tip = u$, so e appears in the v list while $mate(e)$ appears in the u list.

```
#define delete(e) ee = e, oooo, ee→prev→next = ee→next, ee→next→prev = ee→prev
```

⟨ Shrink e by changing u to v 10 ⟩ ≡

```
oo, k = u→deg + v→deg;
for (o, f = u→arcs→next, ff = Λ; o, f→tip; o, f = f→next)
  if (f→tip ≡ v) delete(f), delete(mate(f)), k −= 2, o, f→link = ff, ff = f;
  else o, mate(f)→tip = v;
oo, e→link = ff, v→deg = k;
if (extraverbose)
  printf("level␣%d:␣Shrinking␣%d;␣now␣%s␣has␣degree␣%d\n", l, e→len, v→name, v→deg);
o, ff = v→arcs;   /* now f = u→arcs; we will merge the two lists */
oooo, f→prev→next = ff→next, ff→next→prev = f→prev;
ooo, f→next→prev = ff, ff→next = f→next;
```

This code is used in section 7.

11. Unshrinking uses the principle of "dancing links," whereby we exploit the fact that previously deleted nodes still have good information in their *prev* and *next* fields, provided that we undelete in reverse order.

```
#define undelete(e) ee = e, oooo, ee→next→prev = ee, ee→prev→next = ee
```

⟨ Unshrink e by restoring u 11 ⟩ ≡

```
oo, f = u→arcs, ff = v→arcs;
ooo, ff→next = f→prev→next;
o, ff→next→prev = ff;
ooo, f→prev→next = f, f→next→prev = f;
for (f = f→prev; o, f→tip; o, f = f→prev) o, mate(f)→tip = u;
for (oo, f = e→link, k = v→deg; f; o, f = f→link) k += 2, undelete(mate(f)), undelete(f);
oo, v→deg = k − u→deg;
if (extraverbose)
  printf("level␣%d:␣Unshrinking␣%d;␣now␣%s␣has␣degree␣%d\n", l, e→len, v→name, v→deg);
```

This code is used in section 7.

12. For bridge detection, we try a heuristic that often gives a quick answer when the graph is sparse (namely, to test if u has degree 1). Or if $e\to link\to link \neq \Lambda$, there was another edge between u and v. Otherwise we resort to brute-force breadth-first, testing whether one can get from u to v without e.

When I put this algorithm in a book, I'll probably leave out the two quick-try heuristics, in order to keep the algorithm shorter; breadth-first search will resolve both cases without too much additional calculation. But for now I'm trying to see how useful they are.

```
#define bfs v.V   /* link for the breadth-first search: nonnull if vertex seen */
⟨If e is not a bridge, delete it, set change_e = e, and goto enter 12⟩ ≡
  if (o, u→deg ≡ 1) {
    if (extraverbose) printf("level␣%d:␣%d␣is␣a␣bridge␣with␣endpoint␣%s\n", l, e→len, u→name);
    goto bridge;
  }
  if (o, e→link→link) {
    if (extraverbose) printf("level␣%d:␣%d␣is␣parallel␣to␣%d\n", l, e→len,
          e→link→len ≠ e→len ? e→link→len : e→link→link→len);
    goto nonbridge;
  }
  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 and goto nonbridge 13⟩;
        } else oo, f→tip→bfs = v, w→bfs = f→tip, w = f→tip;
      }
  }
  if (extraverbose) printf("level␣%d:␣%d␣is␣a␣bridge\n", l, e→len);
  for (o, u = e→tip; u ≠ v; o, u→bfs = Λ, u = w) o, w = u→bfs;
  goto bridge;
nonbridge: change_e = e;
  ⟨Delete e and goto enter 14⟩;
bridge:
```

This code is used in section 7.

13. ⟨Nullify all *bfs* links and **goto** *nonbridge* 13⟩ ≡

```
  {
    for (o, u = e→tip; u ≠ v; o, u→bfs = Λ, u = w) o, w = u→bfs;
    goto nonbridge;
  }
```

This code is used in section 12.

14. ⟨Delete e and **goto** *enter* 14⟩ ≡

```
  if (extraverbose) printf("level␣%d:␣deleting␣%d\n", l, e→len);
  ooo, e→link = del(l), del(l) = e;
  delete(e), delete(mate(e)), oo, e→tip→deg−−, v→deg−−;
  goto enter;
```

This code is used in section 12.

15. ⟨ Undelete all edges deleted since entering level l 15 ⟩ ≡

```
for (o, e = del(l); e; o, e = e→link) {
  oooo, mate(e)→tip→deg++, e→tip→deg++, undelete(mate(e)), undelete(e);
  if (extraverbose) printf("undeleting␣%d\n", e→len);
}
```

This code is used in section 7.

16. Getting started. We're done, except for one embarrassing detail: It is necessary to prime the pump by setting up the original near-tree $a_2 \ldots a_{n-1}$. For this purpose I'll use depth-first search, since it seems a bit faster than the alternatives. And I might as well check that the graph is connected.

```
#define sentinel (g→vertices)

⟨ Initialize the algorithm 16 ⟩ ≡
  for (v = g→vertices + 1; v < g→vertices + n; v++) v→bfs = Λ;
  for (k = n − 1, o, w = v = g→vertices, w→bfs = sentinel; ; o, v = w, w = w→bfs) {
    for (oo, e = v→arcs→next; o, u = e→tip; o, e = e→next)
      if (o, u→bfs ≡ Λ) {
        o, aa(k) = e, k−−;
        if (k ≡ 0) goto connected;
        o, u→bfs = w, w = u;
      }
    if (w ≡ sentinel) break;
  }
  printf("Oops,␣the␣graph␣isn't␣connected!\n"); exit(0);
connected:
  for (u = g→vertices; u < g→vertices + n; u++) o, u→bfs = Λ;
  if (extraverbose) {
    printf("Depth-first␣search␣yields␣the␣following␣spanning␣tree:\n");
    print_a(g);
  }
  if (verbose) printf("(%.15g␣mems␣for␣initialization)\n", mems);
```

This code is used in section 1.

17. One final debugging aid.

```
⟨ Subroutines 5 ⟩ +≡
  void print_a(register Graph *g)
  {
    register int k;
    for (k = 1; k < g→n; k++)
      printf("␣a%d=%d␣(%s␣--␣%s)\n", k, aa(k)→len, aa(k)→tip→name, mate(aa(k))→tip→name);
  }
```

18. Index.

GRAYSPAN