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

1. Intro. This is an experimental program to find all of the graceful labelings of a given graph. Some vertex labels may be prespecified if desired, by giving assignments of the form `VERTEX=label` on the command line.

If there are no prespecifications, the solutions are required to be "canonical," in the sense that the vertices of the edge labeled $m-1$ are labeled 1 and m, not 0 and $m-1$. (This saves of factor of 2, because every graceful labeling without prespecifications can be "complemented" by changing each label l to $m-l$.)

I've tried to make the inner loops run fast, using some ideas of Tom Rokicki, together with strange ideas of my own called '*labunlab*' and '*vertunlab*'.

This program is based on BACK-GRACEFUL-ROOTED, which finds only a subset of the graceful labelings but works much faster.

Implementation note: This program uses the function '*__builtin_popcountll*' that's provided by the `gcc` compiler. You should include '`-march=native`' as one of the `CFLAGS` in the `Makefile` that you use when compiling this.

```
#define maxn 64 /* at most this many vertices */
#define maxm 63 /* at most this many edges (could go to 127 with double bitmaps) */
#define sadd64 __builtin_popcountll /* 64-bit sideways addition */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
  ⟨ Global variables 3 ⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, l, m, n, p, q, r, t, bad, vv, ll, carry, forced;
    register unsigned long long ebits, rebits, vbits, del;
    Graph *g;
    Vertex *v, *w;
    Arc *a;
    ⟨ Process the command line, and set prespec to the prespecified labelings 2 ⟩;
    ⟨ Solve the problem 7 ⟩;
    ⟨ Say farewell 4 ⟩;
  }
```

2. ⟨ Process the command line, and set *prespec* to the prespecified labelings 2 ⟩ ≡

```
if (argc < 2) {
  fprintf(stderr, "Usage: %s foo.gb [VERTEX=label...]\n", argv[0]);
  exit(-1);
}
g = restore_graph(argv[1]);
if (¬g) {
  fprintf(stderr, "I couldn't reconstruct graph %s!\n", argv[1]);
  exit(-2);
}
m = g→m/2, n = g→n;
if (m > maxm) {
  fprintf(stderr, "Sorry, at present I require m<=%d!\n", maxm);
  exit(-3);
}
if (n > maxn) {
  fprintf(stderr, "Sorry, at present I require n<=%d!\n", maxn);
  exit(-4);
}
for (k = 2; argv[k]; k++) {
  for (i = 1; argv[k][i]; i++)
    if (argv[k][i] ≡ '=') break;
  if (¬argv[k][i] ∨ sscanf(&argv[k][i + 1], "%d", &label) ≠ 1 ∨ label < 0 ∨ label > m) {
    fprintf(stderr, "spec `%s' doesn't have the form `VERTEX=label'!\n", argv[k]);
    exit(-3);
  }
  argv[k][i] = 0;
  for (j = 0; j < n; j++)
    if (strcmp((g→vertices + j)→name, argv[k]) ≡ 0) break;
  if (j ≡ n) {
    fprintf(stderr, "There's no vertex named `%s'!\n", argv[k]);
    exit(-5);
  }
  if (verttoprespec[j]) {
    fprintf(stderr, "Vertex %s was already specified!\n", (g→vertices + j)→name);
    exit(-6);
  }
  verttoprespec[j] = 1;
  prespec[prespecptr++] = (j ≪ 8) + label;
}
fprintf(stderr, "OK, I've got a graph with %d vertices, %d edges, %d prespec%s.\n", n, m,
    prespecptr, prespecptr ≡ 1 ? "" : "s");
```

This code is used in section 1.

3. ⟨ Global variables 3 ⟩ ≡

```
int vbose = 0;      /* set this nonzero to watch me work */
int label;      /* a label value read from argv[k] */
int prespec[maxn];      /* prespecified labels */
int verttoprespec[maxn];      /* has this vertex been prespecified? */
int prespecptr;      /* how many are prespecified? */
```

See also sections 5 and 15.

This code is used in section 1.

4. ⟨ Say farewell 4 ⟩ ≡

```
fprintf(stderr, "Altogether␣%lld␣%sgraceful␣labeling%s%s", count, prespecptr ? "" : "canonical␣",
    count ≡ 1 ? "" : "s", prespecptr ? "␣with" : "");
for (k = 0; k < prespecptr; k++)
  fprintf(stderr, "␣%s=%d", (g→vertices + (prespec[k] ≫ 8))→name, prespec[k] & #ff);
fprintf(stderr, ";␣%lld␣nodes.\n", nodes);
```

This code is used in section 1.

5. Data structures. The vertices are internally numbered from 0 to $n-1$. Vertex v has $deg[v]$ neighbors, and they appear in the first $deg[v]$ slots of $edges[v]$.

Labels potentially range from 0 to m. If label l hasn't yet been used, $labunlab[l]$ is negative, and $labtovert[l]$ is undefined. Otherwise $labtovert[l]$ is the vertex labeled l, and $labunlab[l]$ is the number of unlabeled neighbors of that vertex.

The value of $verttolab[v]$ is the label of v, if any, otherwise -1. If v is unlabeled, the value of $vertunlab[v]$ tells how many of v's neighbors are also unlabeled. Otherwise $vertunlab[v]$ is the number of neighbors that were unlabeled when v became labeled.

At level l of the backtracking, the first l vertices of $vlist$ have been labeled. The others haven't. (In fact, $vlist$ is a permutation, and $ilist[vlist[k]] = vlist[ilist[i]] = k$ for $0 \le k < n$; $vlist[k]$ is the vertex that was labeled at level k, for $0 \le k < l$.)

Three bitmaps are maintained: $ebits$ records the edge labels that have appeared, with $1_{\rm LL} \ll q$ representing label q; $vbits$ records the vertex labels that have *not* appeared, in the same fashion; and $rebits$ records $ebits$ "backwards," with $1_{\rm LL} \ll (m-q)$ representing label q.

```
⟨ Global variables 3 ⟩ +≡
  int deg[maxn];      /* how many neighbors of v? */
  int edges[maxn][maxm];      /* their identities */
  int verttolab[maxn];      /* what is v's label? */
  int vertunlab[maxn];      /* how many unlabeled neighbors does it have? */
  int labtovert[maxm + 1];      /* what vertex v[l] is labeled l? */
  int labunlab[maxm + 1];      /* how many unlabeled neighbors does v[l] have? */
  int vlist[maxn];      /* a permutation of all the variables */
  int ilist[maxn];      /* the inverse of that permutation */
```

6. We begin by converting from Stanford GraphBase format to the data structures used here.

```
⟨ Initialize the data structures 6 ⟩ ≡
  for (k = 0; k < n; k++) {
    v = g→vertices + k;
    verttolab[k] = -1, vlist[k] = ilist[k] = k;
    for (a = v→arcs; a; a = a→next) {
      w = a→tip;
      edges[k][deg[k]++] = w - g→vertices;
    }
  }
  for (q = 0; q ≤ m; q++) labunlab[q] = -1;
  for (k = 0; k < n; k++) vertunlab[k] = deg[k];
  ebits = rebits = 0, vbits = (1_LL ≪ (m + 1)) - 1;
```

This code is used in section 7.

7. Backtracking. The main computation is based on Walker's backtrack method, Algorithm 7.2.2W. It's an implicit recursion, spelled out so that the costs of updating and downdating are made explicit.

```
⟨ Solve the problem 7 ⟩ ≡
w1: ⟨ Initialize the data structures 6 ⟩;
  l = carry = forced = 0;
w2: nodes++;
  if (l > prespecptr)
    ⟨ Check to see if any unlabeled vertex has at most one option; if so goto w3 or w4 14 ⟩;
  if (carry) ⟨ Determine the r potential moves that might make edge q start with vv 10 ⟩
  else if (l < prespecptr) r = 1, move[l][0] = prespec[l];
  else {
    for (q = (l ≡ prespecptr ? m : target[l − 1] − 1), del = 1LL ≪ q; ebits & del; q−−, del ≫= 1) ;
    if (q ≡ 0) ⟨ Visit a solution and goto w4 9 ⟩;
    target[l] = q;
    ⟨ Determine the r potential moves that might create edge q 11 ⟩;
  }
  moves[l] = r;
w3: if (r > 0) {
    t = move[l][−−r];
    carry = t ≫ 16, vv = (t ≫ 8) & #ff, ll = t & #ff;
    if (vbose) ⟨ Show this potential move 8 ⟩;
    forced = 0;
    ⟨ Give label ll to vertex vv, jumping to abort if it fails 12 ⟩;
    x[l++] = r;
    goto w2;
  }
w4: if (−−l ≥ 0) {
    r = x[l], t = move[l][r], vv = (t ≫ 8) & #ff, ll = t & #ff;
    ⟨ Take label ll from vertex vv, possibly starting from abort 13 ⟩;
    goto w3;
  }
```

This code is used in section 1.

8. ⟨ Show this potential move 8 ⟩ ≡

```
if (forced) fprintf(stderr, "L%d: %s=%d (forced)\n", l, (g→vertices + vv)→name, ll);
else if (l < prespecptr) fprintf(stderr, "L%d: %s=%d (prespecified)\n", l, (g→vertices + vv)→name, ll);
else if (carry) fprintf(stderr, "L%d: %s=%d (%d of %d starting edge %d)\n", l,
        (g→vertices + vv)→name, ll, moves[l] − r, moves[l], target[l]);
else if (l > 0 ∧ move[l − 1][x[l − 1]] ≥ (1 ≪ 16))
  fprintf(stderr, "L%d: %s=%d (%d of %d completing edge %d)\n", l, (g→vertices + vv)→name, ll,
        moves[l] − r, moves[l], target[l]);
else fprintf(stderr, "L%d: %s=%d (%d of %d for edge %d)\n", l, (g→vertices + vv)→name, ll,
        moves[l] − r, moves[l], target[l]);
```

This code is used in section 7.

9. ⟨ Visit a solution and **goto** *w4* 9 ⟩ ≡

```
{
  count++;
  for (k = 0; k ≤ m; k++)
    if (labunlab[k] ≥ 0) {
      if (labunlab[k] > 0) fprintf(stderr, "This␣can't␣happen!\n");
      vv = labtovert[k];
      printf("%s=%d␣", (g→vertices + vv)→name, k);
    }
  printf("#%lld\n", count);
  fflush(stdout);
  goto w4;
}
```

This code is used in section 7.

10. At this point *vv* and *ll* have been set on the previous level, when vertex *vv* was labeled *ll* and we're hoping to give the label *ll* + *target*[*l* − 1] to one of *v*'s neighbors.

⟨ Determine the *r* potential moves that might make edge *q* start with *vv* 10 ⟩ ≡

```
{
  q = target[l − 1], target[l] = q;
  for (r = 0, i = deg[vv] − 1; i ≥ 0; i−−) {
    t = verttolab[edges[vv][i]];
    if (t < 0) move[l][r++] = (edges[vv][i] ≪ 8) + (ll + q);
  }
}
```

This code is used in section 7.

11. There are essentially two ways to create an edge labeled q, for each pair of vertex labels (j,k) with $k=j+q$: Either exactly one of $labunlab[j]$ and $labunlab[k]$ is positive; or both of them are negative. The latter case, which doesn't occur in "rooted" solutions, can potentially involve a huge number of subcases, because *any* pair of neighboring unlabeled vertices might qualify.

I think this is the inner loop.

```
⟨ Determine the r potential moves that might create edge q 11 ⟩ ≡
  for (r=0, j=(l≡2∧¬prespecptr), k=j+q; k≤m; j++, k++) {
    if (labunlab[j]>0∧labunlab[k]<0) {
      for (vv=labtovert[j], i=deg[vv]-1; i≥0; i--) {
        t=verttolab[edges[vv][i]];
        if (t<0) move[l][r++]=(edges[vv][i]≪8)+k;
      }
    } else if (labunlab[j]<0) {
      if (labunlab[k]>0) {
        for (vv=labtovert[k], i=deg[vv]-1; i≥0; i--) {
          t=verttolab[edges[vv][i]];
          if (t<0) move[l][r++]=(edges[vv][i]≪8)+j;
        }
      } else if (labunlab[k]<0) {
        for (i=n-1; i≥l; i--) {
          vv=vlist[i];
          if (vertunlab[vv]) move[l][r++]=(1≪16)+(vv≪8)+j;
        }
      }
    }
  }
```

This code is used in section 7.

12. And this loop too is pretty much "inner."

I apologize for being unable to resist jumping from this section into the next, when backtracking is seen to be needed.

```
⟨ Give label ll to vertex vv, jumping to abort if it fails 12 ⟩ ≡
  for (p=deg[vv], i=p-1, bad=0; i≥0; i--) {
    j=edges[vv][i], t=verttolab[j];
    if (t≥0) {
      p--, labunlab[t]--, q=abs(t-ll);
      del=1LL≪q, bad|=ebits & del;
      ebits+=del, rebits+=1LL≪(m-q);
    } else vertunlab[j]--;
  }
  labunlab[ll]+=p+1;
  if (bad) {
    if (vbose) fprintf(stderr, "L%d, conflict setting %s=%d\n", l, (g→vertices+vv)→name, ll);
    goto abort;
  }
  verttolab[vv]=ll, labtovert[ll]=vv;
  t=ilist[vv], p=vlist[l];
  vlist[l]=vv, vlist[t]=p, ilist[vv]=l, ilist[p]=t;
  vbits-=1LL≪ll;
```

This code is used in section 7.

13. Here I use the "sparse-set" trick to avoid downdating *vlist* and *ilist*. (See 7.2.2–(23).)

```
⟨ Take label ll from vertex vv, possibly starting from abort 13 ⟩ ≡
  vbits += 1LL ≪ ll;
  verttolab[vv] = −1;
abort: for (i = deg[vv] − 1; i ≥ 0; i−−) {
    j = edges[vv][i], t = verttolab[j];
    if (t ≥ 0) {
      labunlab[t]++, q = abs(t − ll);
      ebits −= 1LL ≪ q, rebits −= 1LL ≪ (m − q);
    } else vertunlab[j]++;
  }
  labunlab[ll] = −1;
```

This code is used in section 7.

14. Empirical tests showed, at least in the problems I studied, that the domain of possible values for an unlabeled vertex begins to dwindle until only one value is left, or even no values at all, on the very levels of the search that consume the most time.

The heuristic checks that are performed in this section are purely optional. Indeed, the loop is rather lengthy and unlikely to succeed at levels near the root, when many vertices must still be labeled. So my first inclination was to perform these tests at deeper levels only. However, I also noticed that relatively little total time was needed to make (admittedly fruitless) tests at the shallow levels, at least in my limited experiments; so I stopped asking the user to decide where to make them kick in. A user who wants more control could do that by making a change file that selectively avoids the code below.

I should point out that the test here is not complete. Suppose, for example, that an unlabeled vertex v has two neighbors labeled 10 and 20, but there is no vertex labeled 15 and no edge labeled 5. The test we make does not remove 15 from v's domain, although that value would fail (because it would create two 5s).

A subtlety arises when we discover an unlabeled vertex vv that has only one viable label ll remaining. When *carry* is set, we cannot force vv to be labeled ll, because we're obliged to create an edge whose label is $target[l-1]$, using a neighbor of the vertex labeled at level $l-1$. Forcing vv could lead to duplicate solutions. On the other hand, if *carry* is not set, we *can* force vv; but we must set $target[l] = target[l-1]$, which is the largest edge whose existence is known. (I've made sure that $l > prespecptr$, so that $target[l-1]$ is meaningful.)

⟨ Check to see if any unlabeled vertex has at most one option; if so **goto** *w3* or *w4* 14 ⟩ ≡

```
{
  register int i, j, k, vv, ll;
  register unsigned long long vvbits;
  for (forced = 0, k = l; k < n; k++) {
    vv = vlist[k], vvbits = vbits;
    for (i = deg[vv] - 1; i ≥ 0; i--) {
      j = edges[vv][i], ll = verttolab[j];
      if (ll ≥ 0)      /* j is a labeled neighbor of vv, which is unlabeled */
        vvbits &= ∼((ebits ≪ ll) + (rebits ≫ (m - ll)));
    }
    i = sadd64(vvbits);
    if (i > 1) continue;
    if (i ≡ 0) {
      if (vbose) fprintf(stderr, "L%d, %s stuck\n", l, (g→vertices + vv)→name);
      goto w4;
    }
    if (carry) continue;
    ll = sadd64(vvbits - 1);
    move[l][0] = (vv ≪ 8) + ll, forced = 1;
  }
  if (forced) {
    r = 1, moves[l] = 1;      /* forced move */
    target[l] = target[l - 1];      /* see above */
    goto w3;
  }
}
```

This code is used in section 7.

15. ⟨ Global variables 3 ⟩ +≡

```
long long count;      /* this many solutions found so far */
long long nodes;      /* this many nodes in the search tree so far */
int target[maxn];     /* the edge we try to set, on each level */
int move[maxn][maxn * maxm];      /* the things we want to try, on each level */
int x[maxn];      /* the moves currently being tried, on each level */
int moves[maxn];      /* used in debugging and verbose tracing only */
```

16. Index.

⟨ Check to see if any unlabeled vertex has at most one option; if so **goto** *w3* or *w4* 14 ⟩ Used in section 7.
⟨ Determine the r potential moves that might create edge q 11 ⟩ Used in section 7.
⟨ Determine the r potential moves that might make edge q start with vv 10 ⟩ Used in section 7.
⟨ Give label ll to vertex vv, jumping to *abort* if it fails 12 ⟩ Used in section 7.
⟨ Global variables 3, 5, 15 ⟩ Used in section 1.
⟨ Initialize the data structures 6 ⟩ Used in section 7.
⟨ Process the command line, and set *prespec* to the prespecified labelings 2 ⟩ Used in section 1.
⟨ Say farewell 4 ⟩ Used in section 1.
⟨ Show this potential move 8 ⟩ Used in section 7.
⟨ Solve the problem 7 ⟩ Used in section 1.
⟨ Take label ll from vertex vv, possibly starting from *abort* 13 ⟩ Used in section 7.
⟨ Visit a solution and **goto** *w4* 9 ⟩ Used in section 7.

BACK-GRACEFUL