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

**1. Intro.** This is an experimental program to find "rooted" graceful labelings of a given graph. (Some of the vertex labels may be prespecified. Every edge has a vertex in common with a longer edge, or has at least one prespecified vertex, except possibly for edge $m$ itself.)

I hacked this code from BACK-GRACEFUL-ROOTED, which considered the special case where vertex 0 (only) was prespecified, and which looked exhaustively for *all* solutions. By contrast, this program is intended for large graphs, where we feel lucky to find even a single solution and we can't hope to find them all. Therefore we'll use randomization with frequent restarts.

(Thanks to Tom Rokicki for many of the ideas used here.)

```
#define maxn 64     /* at most this many vertices */
#define maxm 128    /* at most this many edges */
#define interval 100    /* print '.' to show progress, every so often */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gb_graph.h"
#include "gb_save.h"
#include "gb_flip.h"
  ⟨Global variables 5⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, l, m, n, p, q, r, t, bad, vv, ll;
    Graph *g;
    Vertex *v, *w;
    Arc *a;
    ⟨Process the command line, and set prespec to the prespecified labelings 2⟩;
    ⟨Set up the fixed data structures 3⟩;
    while (1) {
      rounds++;
      if ((rounds % interval) ≡ 0) fprintf(stderr, ".");
      ⟨Set the cutoff for a new trial 4⟩;
      ⟨Backtrack for at most T steps 8⟩;
    }
  }
```

**2.** The command line names a graph in SGB format, followed by a minimum cutoff time $T_{\min}$ (measured in search tree nodes examined before restarting). Then comes a random seed, so that results of this run can be replicated if necessary. Then come zero or more prespecifications, in the form '`VERTNAME=label`'.

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

```
if (argc < 4 ∨ sscanf(argv[2], "%lld", &Tmin) ≠ 1 ∨ sscanf(argv[3], "%d", &seed) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣foo.gb␣Tmin␣seed␣[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 = 4; 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;
  if (¬xprespec ∧ (label ≡ 0 ∨ label ≡ m)) xprespec = 1, prespec[0] = (j ≪ 8) + label;
  else prespec[prespecptr++] = (j ≪ 8) + label;
}
gb_init_rand(seed);
fprintf(stderr, "OK,␣I've␣got␣a␣graph␣with␣%d␣vertices,␣%d␣edges.\n", n, m);
```

This code is used in section 1.

**3.** ⟨ Set up the fixed data structures 3 ⟩ ≡

$$\begin{array}{l}
\textbf{for } (k = 0;\ k < n;\ k{+}{+})\ \{\\
\quad v = g\text{→}vertices + k;\\
\quad \textbf{for } (a = v\text{→}arcs;\ a;\ a = a\text{→}next)\ \{\\
\quad\quad w = a\text{→}tip;\\
\quad\quad edges[k][deg[k]{+}{+}] = w - g\text{→}vertices;\\
\quad \}\\
\}\\
\textbf{for } (k = 0;\ k < prespecptr;\ k{+}{+})\ \ moves[k] = 1, move[k][0] = prespec[k];
\end{array}$$

This code is used in section 1.

**4.** Las Vegas algorithms like this one are best controlled by multiples of the "reluctant doubling" sequence defined by Luby, Sinclair, and Zuckerman (see equation 7.2.2.2–(131) in *TAOCP*), unless we already know a pretty good cutoff value.

⟨ Set the cutoff for a new trial 4 ⟩ ≡

$$\begin{array}{l}
T = Tmin * reluctant_v;\\
\textbf{if } ((reluctant_u\ \&\ {-reluctant_u}) \neq reluctant_v)\ \ reluctant_v \ll= 1;\\
\textbf{else}\ \ reluctant_u{+}{+}, reluctant_v = 1;
\end{array}$$

This code is used in section 1.

**5.** ⟨ Global variables 5 ⟩ ≡

```
int vbose = 0;    /* set this nonzero to watch me work */
int rounds;    /* how many random trials have we started? */
long long nodes;    /* how many nodes have we started on this round? */
long long reluctant_u = 1, reluctant_v = 1;    /* restart parameters */
long long T;    /* cutoff time for the current random trial */
long long Tmin;    /* minimum cutoff time (from the command line) */
int seed;    /* seed for gb_init_rand */
int label;    /* a label value read from argv[k] */
int prespec[maxn];    /* prespecified labels */
int verttoprespec[maxn];    /* has this vertex been prespecified? */
int prespecptr = 1;    /* how many are prespecified? */
int xprespec;    /* did any of them specify label 0 or label m? */
```

See also sections 6 and 17.

This code is used in section 1.

**6. 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 $edges[v]$. Its label is $verttolab[v]$; but $verttolab[v] = -1$ if $v$ hasn't yet been labeled.

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.

For each $q$ between 1 and $m$, $edgecount[q]$ is the number of edges labeled $q$. (This number might momentarily exceed 1, although it will be exactly equal to 1 when the labeling is graceful.)

⟨ Global variables 5 ⟩ +≡

```
int deg[maxn];   /* how many neighbors of v? */
int edges[maxn][maxm];   /* their identities */
int verttolab[maxn];   /* what is v's label? */
int labtovert[maxm + 1];   /* what vertex v[l] is labeled l? */
int labunlab[maxm + 1];   /* how many unlabeled neighbors does v[l] have? */
int edgecount[maxm + 1];   /* how many edges are labeled q? */
```

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

⟨ Initialize the data structures 7 ⟩ ≡

```
for (k = 0; k < n; k++) {
  verttolab[k] = -1;
}
for (q = 0; q <= m; q++) labunlab[q] = -1, edgecount[q] = 0;
```

This code is used in section 8.

**8. 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.

```
⟨ Backtrack for at most T steps 8 ⟩ ≡
w1: ⟨ Initialize the data structures 7 ⟩;
  l = 0, nodes = 0;
  if (¬xprespec) {
    while (1) {
      vv = (n * (unsigned long long) gb_next_rand()) ≫ 31;
      if (¬verttoprespec[vv]) break;
    }
    move[0][0] = vv ≪ 8;
  }
w2: if (++nodes > T) goto done;
  if (l < prespecptr) {
    r = 1;
    goto w3;
  }
  q = target[l − 1];
  for (q = (q ? q − 1 : m); edgecount[q]; q−−) ;
  if (q ≡ 0) ⟨ Visit a solution and goto done 10 ⟩;
  target[l] = q;
  ⟨ Determine the r potential moves that might create edge q 12 ⟩;
  ⟨ Shuffle those moves 11 ⟩;
  moves[l] = r;
w3: if (r > 0) {
    t = move[l][−−r];
    vv = t ≫ 8, ll = t & #ff;
    if (vbose) ⟨ Show this potential move 9 ⟩;
    ⟨ Update the edge counts that would result from verttolab[vv] = ll, setting bad nonzero if any of them
        would exceed 1, also setting p to the number of unlabeled neighbors of vv 13 ⟩;
    if (bad) goto w4a;
    ⟨ Give label ll to vertex vv 15 ⟩;
    x[l++] = r;
    goto w2;
  }
w4: if (−−l ≥ 0) {
    r = x[l], t = move[l][r], vv = t ≫ 8, ll = t & #ff;
    ⟨ Take label ll from vertex vv 16 ⟩;
  w4a: ⟨ Downdate the edge counts that would result from verttolab[vv] = ll 14 ⟩;
    goto w3;
  }
  done:
```

This code is used in section 1.

**9.** 
```
⟨ Show this potential move 9 ⟩ ≡
  if (target[l]) 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]);
  else fprintf(stderr, "L%d:␣%s=%d␣(prespecified)\n", l, (g→vertices + vv)→name, ll);
```

This code is used in section 8.

**10.** ⟨ Visit a solution and **goto** *done* 10 ⟩ ≡

```
{
  count++;
  fprintf(stderr, "\n");
  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("#%d (round %d, step %lld\n", count, rounds, nodes);
  fflush(stdout);
  goto done;
}
```

This code is used in section 8.

**11.** By giving an arbitrary permutation to the list of possible moves, we're providing the maximum randomization over the entire search tree for all rooted labelings that meet the prespecifications.

I don't think this will add a significant amount to the running time. But if it does, we could back off by doing only a partial shuffle (for example, only on certain levels, or a maximum of 10 swaps, or . . . ).

⟨ Shuffle those moves 11 ⟩ ≡

```
for (q = r − 1; q > 0; q−−) {
  p = ((q + 1) * ((unsigned long long) gb_next_rand())) ≫ 31;
  t = move[l][p];
  move[l][p] = move[l][q];
  move[l][q] = t;
}
```

This code is used in section 8.

**12.** I think this is the inner loop.

⟨ Determine the $r$ potential moves that might create edge $q$ 12 ⟩ ≡

```
for (r = j = 0, k = 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 ∧ 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;
    }
  }
}
```

This code is used in section 8.

**13.** And this loop too is pretty much "inner."

⟨ Update the edge counts that would result from $verttolab[vv] = ll$, setting $bad$ nonzero if any of them would exceed 1, also setting $p$ to the number of unlabeled neighbors of $vv$ 13 ⟩ ≡

```
for (p = deg[vv], i = p - 1, bad = 0; i >= 0; i--) {
  t = verttolab[edges[vv][i]];
  if (t >= 0) {
    p--;
    q = abs(t - ll);
    t = edgecount[q], bad |= t;
    edgecount[q] = t + 1;
  }
}
```

This code is used in section 8.

**14.** Maybe the last line here will go faster if rewritten '$edgecount[abs(t-ll)] -= (t \ge 0)$', because of branch prediction? If so, are modern compilers smart enough to see this?

⟨ Downdate the edge counts that would result from $verttolab[vv] = ll$ 14 ⟩ ≡

```
for (i = deg[vv] - 1; i >= 0; i--) {
  t = verttolab[edges[vv][i]];
  if (t >= 0) edgecount[abs(t - ll)]--;
}
```

This code is used in section 8.

**15.** The value of $p$ has been set up for us nicely at this point.

We need to perform another loop, but this one isn't needed quite so often.

⟨ Give label $ll$ to vertex $vv$ 15 ⟩ ≡

```
verttolab[vv] = ll, labtovert[ll] = vv, labunlab[ll] = p;
for (i = deg[vv] - 1; i >= 0; i--) {
  t = verttolab[edges[vv][i]];
  if (t >= 0) labunlab[t]--;
}
```

This code is used in section 8.

**16.** ⟨ Take label $ll$ from vertex $vv$ 16 ⟩ ≡

```
for (i = deg[vv] - 1; i >= 0; i--) {
  t = verttolab[edges[vv][i]];
  if (t >= 0) labunlab[t]++;
}
verttolab[vv] = labunlab[ll] = -1;
```

This code is used in section 8.

**17.** ⟨ Global variables 5 ⟩ +≡

```
int count;  /* this many solutions found so far */
int target[maxn];  /* the edge we try to set, on each level */
int move[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 verbose tracing only */
```

**18. Index.**

# BACK-GRACEFUL-ROOTED-RANDOMRESTARTS