1. Intro. This program is an experiment. I'm trying to find a function $h(G)$, defined for any (simple) graph on n vertices, that has the following properties:

0) $h(G)$ can be computed rapidly, with a fairly short program.
1) $0 \le h(G) < 2^{32}$.
2) $h(G) = h(G')$ whenever G is isomorphic to G'.
3) $h(G) \ne h(G')$ whenever G is *not* isomorphic to G'.

I'm thinking of $n < 10$, say; there are 274668 simple graphs with 9 vertices.

Of course there are oodles of functions $h(G)$ that satisfy (0), (1), and (2). But in order to satisfy property (3), I can't think of any plausible method that needs only $O(n)$ steps. My best idea so far takes time $O(n^2)$, and consists of n linear-time hashes on the n graphs obtained by giving a special mark to each of the vertices in turn.

(When the graph has m edges, I could try instead for m linear-time hashes on the graphs obtained by marking each of the *edges* in turn. My goal, however, is to perform approximately cn^2 operations where c is as small as possible. Hence I'm experimenting first with the simplest methods that seem plausible, before resorting to anything fancier.)

The method adopted here is based on a multivariate polynomial that is an invariant of any graph. By evaluating this polynomial at a "random" point, mod 2^{32}, I'm hoping to have a suitable hash.

Input to this program on *stdin* is a list of simple graphs, one per line, in the format defined by G6-TO-EZ. (Different graphs in the input need not actually have the same number of vertices.) The output is the sequence of (hexadecimal) hash codes, again one per line, in the same order.

Note, 07 December 2020: Success — almost! I tried seeds 1000–1999, and found two cases with only one collision: 1286 and 1654. So I used the seed 1286 to make the files in `/home/lib/graphs`; see the file `/home/lib/graphs/collision` for details. There's no real ambiguity, because one graph has 18 edges, the other has 28 edges. (Both have 9 vertices.)

2. The command line contains a random seed. If the polynomials defined here are truly distinct for nonisomorphic graphs — and I really don't understand the situation well enough to be sure of that! — I'm hoping to find a seed that distinguishes all of them.

```
#define maxn 11     /∗ beware: there are more than a billion 11-vertex graphs ∗/
#define maxrn 50     /∗ only this many “random” numbers are generated ∗/
#define bufsize 120     /∗ 112 is enough to handle n = 11 ∗/

#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h"
  char buf[bufsize];
  int seed;     /∗ command-line parameter ∗/
  unsigned int ran[maxrn];     /∗ the “random” numbers used — all even ∗/
  unsigned int hash[maxn];     /∗ individual hashes for different marked vertices ∗/
  unsigned int tag[maxn], ttag[maxn];     /∗ intermediate hash values on vertices ∗/
  int deg[maxn];     /∗ degree of each vertex ∗/
  int nbr[maxn][maxn];     /∗ edges at each vertex ∗/
  int vert[maxn];     /∗ vertices found breadth-first ∗/
  int start[maxn + 2] = {0, 1};     /∗ boundaries between levels in vert ∗/

  main(int argc, char ∗argv[ ])
  {
    register int i, j, k, l, n, p, q, v, w, vvv;
    register unsigned int h, t;

    ⟨Process the command line and set ran 3⟩;
    while (1) {
      if (¬fgets(buf, bufsize, stdin)) break;
      for (n = k = 0; ; k += 2) {
        if (buf[k] ≡ '0') continue;     /∗ isolated vertex is ignored ∗/
        i = buf[k] − 'a', j = buf[k + 1] − 'a';
        if (i < 0 ∨ j < 0 ∨ i > maxn ∨ j > maxn) break;
        while (i ≥ n) deg[n++] = 0;
        while (j ≥ n) deg[n++] = 0;
        nbr[i][deg[i]++] = j;
        nbr[j][deg[j]++] = i;
      }
      if (buf[k] ≠ '\n') {
        fprintf(stderr, "bad␣format:␣%s", buf);
        exit(−2);
      }
      ⟨Output the hash value for buf 4⟩;
    }
  }
```

3. ⟨Process the command line and set *ran* 3⟩ ≡

```
  if (argc ≠ 2 ∨ sscanf(argv[1], "%d", &seed) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣seed␣<␣graphs\n", argv[0]);
    exit(−1);
  }
  gb_init_rand(seed);
  for (k = 0; k < maxrn; k++) ran[k] = gb_next_rand( ) ≪ 1;
```

This code is used in section 2.

4. ⟨ Output the hash value for *buf* 4 ⟩ ≡

```
for (i = 0; i < n; i++) ⟨ Compute hash[i] for the graph with vertex i marked 5 ⟩;
for (h = i = 0; i < n; i++) h += hash[i] ≫ 1;      /* each hash[i] is odd */
printf("%08x\n", h);
```

This code is used in section 2.

5. We start by creating the breadth-first search tree for vertices reachable from i. Vertices at distance j are tagged with *ran*[j].

⟨ Compute *hash*[i] for the graph with vertex i marked 5 ⟩ ≡

```
{
  for (k = 0; k < n; k++) tag[k] = 0;
  vert[0] = i, tag[i] = 1, p = 1;
  for (j = 0; start[j] < p; j++) {
    for (k = start[j]; k < start[j + 1]; k++) {
      v = vert[k];
      for (l = 0; l < deg[v]; l++) {
        w = nbr[v][l];
        if (¬tag[w]) tag[w] = ran[j] + 1, vert[p++] = w;
      }
    }
    start[j + 2] = p;
  }
  ⟨ Compute h by mixing up the tags 7 ⟩;
  hash[i] = h;
}
```

This code is used in section 4.

6. The key nontrivial step is to modify the tag of each vertex v at a given level q in the BFS tree, multiplying it by the product of *ran*[j] − *tag*[w] over v's neighbors w. (The multiplication is mod 2^{32}. Those factors are odd, because *ran*[j] is even and *tag*[w] is odd.)

⟨ Do a multiplicative step in level q 6 ⟩ ≡

```
{
  for (k = start[q]; k < start[q + 1]; k++) {
    v = vert[k], t = tag[v];
    for (l = 0; l < deg[v]; l++) {
      w = nbr[v][l];
      t *= ran[j] − tag[w];
    }
    ttag[v] = t, h *= t;
  }
  for (k−−; k ≥ start[q]; k−−) v = vert[k], tag[v] = ttag[v];
}
```

This code is used in section 7.

7. At this point $p = start[j] = start[j+1]$ is the size of the component that contains vertex i; we might have $p < n$ if the graph isn't connected. The maximum distance from i is $j - 1$.

At present I simply transform the tags from the bottom of the tree back up to the top. (If that isn't sufficient, I plan to try going back down to the bottom.)

⟨ Compute h by mixing up the tags 7 ⟩ ≡

```
for (h = 1, q = j − 1; q ≥ 0; q−−, j++) ⟨ Do a multiplicative step in level q 6 ⟩;
```

This code is used in section 5.

8. Index.

GRAPH-HASH