1. Intro. This experimental program tries to construct a 4-regular graph with girth ≥ 6, having a given number n of vertices. It does this by random choices, using a method more-or-less suggested by Exoo, McKay, Myrvold, and Nadon [*J. Discrete Algorithms* **9** (2011), 168–169]: We start with an empty graph and add one edge at a time, provided that the new edge doesn't complete a too-short cycle. We prefer to add an edge between two unsaturated vertices that already have the highest possible sum of degrees.

I wrote this in a hurry, because I don't think such graphs are rare (unless n is small).

```
#define maxn 100   /* max potential vertices */
#define maxe ((maxn * (maxn - 1)) >> 1)   /* max potential edges */
#include <stdio.h>
#include <stdlib.h>
#include "gb_flip.h"
  int nn;   /* the given number */
  int seed;   /* the given seed for pseudorandom numbers */
  int e;   /* the current number of edges */
  ⟨ Typedefs 4 ⟩;
  ⟨ Global variables 3 ⟩;
  ⟨ Subroutines 6 ⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, s, d1, d2, u, v, w, p, t, uu;
    ⟨ Process the command line 2 ⟩;
    ⟨ Initialize the graph to empty 8 ⟩;
    for (e = 0; e < 2 * nn; ) {
      if (sanity_checking) sanity();
      for (s = 6; s ≥ 0; s--) {
        for (p = 0, d1 = (s > 3 ? 3 : s), d2 = s - d1; d1 ≥ d2; d1--, d2++)
          ⟨ Record each eligible edge between u of degree d1 and v of degree d2 16 ⟩;
        if (p) goto progress;   /* we found p eligible edges */
      }
      ⟨ Delete a random edge 20 ⟩;
      e--; continue;   /* bad luck: no edges are eligible */
    progress: ⟨ Insert a random eligible edge 19 ⟩;
      e++;
    }
    ⟨ Output the solution 21 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
  if (argc ≠ 3 ∨ sscanf(argv[1], "%d", &nn) ≠ 1 ∨ sscanf(argv[2], "%d", &seed) ≠ 1) {
    fprintf(stderr, "Usage: %s n seed\n", argv[0]);
    exit(-1);
  }
  if (nn > maxn) {
    fprintf(stderr, "Recompile me: I don't allow n (%d) to exceed %d!\n", nn, maxn);
    exit(-2);
  }
  gb_init_rand(seed);
```

This code is used in section 1.

3. Data structures. I'm in a mood to use sequential data structures today, so I'll leave doubly linked lists for another day. This program has sequential lists for (i) the vertices of given degree, (ii) the neighbors of a given vertex, and (iii) the set of all current edges. There also are inverted lists so that we can do deletions.

For each vertex v, with $0 \le v < nn$, there's an array $adj[v]$ that contains the $deg[v]$ current neighbors of v.

For each d, with $0 \le d \le 4$, there's an array $dg[d]$ that lists the $dglen[d]$ vertices whose current degree is d. The position of vertex v in $dg[deg[v]]$ is $dginx[v]$.

⟨ Global variables 3 ⟩ ≡

```
int adj[maxn][4];   /* vertices adjacent to a given v */
int deg[maxn];   /* current length of adj entries */
int dg[5][maxn];   /* vertices having a given degree */
int dglen[5];   /* current length of dg entries */
int dginx[maxn];   /* where does v appear in the dg table? */
```

See also sections 5, 7, and 17.

This code is used in section 1.

4. Each edge is captured also in a record that contains its endpoints u and v, together with the places *uinx*, *vinx* where they appear within $adj[v]$ and $adj[u]$.

⟨ Typedefs 4 ⟩ ≡

```
typedef struct {
  int u, v;   /* endpoints */
  int uinx, vinx;   /* inverted indices */
} edge;
```

This code is used in section 1.

5. The current edges appear in an array called *ee*. The edge that corresponds to $adj[v][j]$ appears in position $adjinx[v][j]$ of this array.

Since every vertex has at most four neighbors, there are at most $2n$ edges.

⟨ Global variables 3 ⟩ +≡

```
edge ee[2 * maxn];   /* the current edges */
int adjinx[maxn][4];   /* inverse indexes for edges */
```

6. Here's a routine that verifies all the redundant aspects of these structures. I found it helpful when debugging (and also when first writing the code).

⟨ Subroutines 6 ⟩ ≡

```
void sanity_fail(char *m, int x, int y)
{
  fprintf(stderr, "%s␣(%d,%d)!\n", m, x, y);
}

void sanity(void)
{     /* check validity of the edge structures */
  register d, j, s, u, v;

  for (v = s = 0; v < nn; v++) s += deg[v];
  if (s ≠ 2 * e) sanity_fail("bad␣sum␣of␣degs", s, 2 * e);
  for (d = s = 0; d ≤ 4; d++) s += dglen[d];
  if (s ≠ nn) sanity_fail("bad␣sum␣of␣dglens", s, nn);
  for (d = 0; d ≤ 4; d++)
    for (j = 0; j < dglen[d]; j++) {
      v = dg[d][j];
      if (deg[v] ≠ d) sanity_fail("bad␣deg", v, d);
      if (dginx[v] ≠ j) sanity_fail("bad␣dginx", v, j);
    }
  for (j = 0; j < e; j++) {
    u = ee[j].u, v = ee[j].v;
    if (u ≠ adj[v][ee[j].uinx]) sanity_fail("bad␣uinx", u, v);
    if (v ≠ adj[u][ee[j].vinx]) sanity_fail("bad␣vinx", u, v);
    if (adjinx[u][ee[j].vinx] ≠ j) sanity_fail("bad␣adjinx", u, j);
    if (adjinx[v][ee[j].uinx] ≠ j) sanity_fail("bad␣adjinx", v, j);
  }
}
```

See also sections 9 and 12.

This code is used in section 1.

7. A global "*time*" counter advances by 1 every time we insert or delete an edge.

The *wait* array is used to place a temporary embargo on edges that have been deleted; such edges cannot be reinserted until $time \ge wait[u][v]$.

While deciding what edge to insert next, we will build a list of all the currently eligible ones, called *elig*.

```
#define sanity_checking  time ≥ 0    /* should sanity be checked now? */
```

⟨ Global variables 3 ⟩ +≡

```
int time;     /* the number of updates we've made */
int wait[maxn][maxn];     /* lower bound on the time when an edge is eligible */
int uelig[maxe], velig[maxe];     /* endpoints of available choices */
int elig;     /* the current number of choices */
```

8. ⟨ Initialize the graph to empty 8 ⟩ ≡

```
e = 0;     /* actually e is already zero; but what the heck */
for (v = 0; v < nn; v++) deg[v] = 0, dg[0][v] = v, dginx[v] = v;
dglen[0] = nn;
```

This code is used in section 1.

9. Here's how to insert a new edge uv:

$\langle$ Subroutines 6 $\rangle$ $+\equiv$

```
void insert(register int u, register int v)
{
  register int d, j, k, w;
  ee[e].u = u, ee[e].v = v;
  ⟨ Append v to adj[u] 10 ⟩;
  ⟨ Append u to adj[v] 11 ⟩;
}
```

10. When increasing $deg[u]$ we must move u from one slot in dg to another.

$\langle$ Append v to $adj[u]$ 10 $\rangle$ $\equiv$

```
d = deg[u], adj[u][d] = v, adjinx[u][d] = e, ee[e].vinx = d, deg[u] = d + 1;
j = dginx[u], k = dglen[d];
w = dg[d][k − 1], dg[d][j] = w, dginx[w] = j, dglen[d] = k − 1;   /* remove u from dg[d] */
k = dglen[d + 1], dg[d + 1][k] = u, dginx[u] = k, dglen[d + 1] = k + 1;   /* put it in dg[d + 1] */
```

This code is used in section 9.

11. And similarly, ...

$\langle$ Append u to $adj[v]$ 11 $\rangle$ $\equiv$

```
d = deg[v], adj[v][d] = u, adjinx[v][d] = e, ee[e].uinx = d, deg[v] = d + 1;
j = dginx[v], k = dglen[d];
w = dg[d][k − 1], dg[d][j] = w, dginx[w] = j, dglen[d] = k − 1;   /* remove v from dg[d] */
k = dglen[d + 1], dg[d + 1][k] = v, dginx[v] = k, dglen[d + 1] = k + 1;   /* put it in dg[d + 1] */
```

This code is used in section 9.

12. The **delete** routine goes the other way, which is a bit harder.

$\langle$ Subroutines 6 $\rangle$ $+\equiv$

```
void delete (register int j)
{
  register int d, i, k, u, v, ui, vi, w;
  u = ee[j].u, v = ee[j].v, ui = ee[j].uinx, vi = ee[j].vinx;
  ⟨ Delete u from adj[v] 13 ⟩;
  ⟨ Delete v from adj[u] 14 ⟩;
  ⟨ Delete ee[j] from ee 15 ⟩;
}
```

13. $\langle$ Delete u from $adj[v]$ 13 $\rangle$ $\equiv$

```
d = deg[v];
if (ui ≠ d − 1) {
  w = adj[v][d − 1], i = adjinx[v][d − 1], adj[v][ui] = w, adjinx[v][ui] = i;
  if (w ≡ ee[i].u) ee[i].uinx = ui;
  else ee[i].vinx = ui;
}
deg[v] = d − 1, i = dginx[v], k = dglen[d];
w = dg[d][k − 1], dg[d][i] = w, dginx[w] = i, dglen[d] = k − 1;   /* remove v from dg[d] */
k = dglen[d − 1], dg[d − 1][k] = v, dginx[v] = k, dglen[d − 1] = k + 1;   /* put it in dg[d − 1] */
```

This code is used in section 12.

14. ⟨ Delete v from $adj[u]$ 14 ⟩ ≡
$d = deg[u]$;
if $(vi \neq d - 1)$ {
  $w = adj[u][d - 1], i = adjinx[u][d - 1], adj[u][vi] = w, adjinx[u][vi] = i$;
  if $(w \equiv ee[i].u)$ $ee[i].uinx = vi$;
  else $ee[i].vinx = vi$;
}
$deg[u] = d - 1, i = dginx[u], k = dglen[d]$;
$w = dg[d][k - 1], dg[d][i] = w, dginx[w] = i, dglen[d] = k - 1$; /* remove u from $dg[d]$ */
$k = dglen[d - 1], dg[d - 1][k] = u, dginx[u] = k, dglen[d - 1] = k + 1$; /* put it in $dg[d - 1]$ */

This code is used in section 12.

15. ⟨ Delete $ee[j]$ from ee 15 ⟩ ≡
if $(j \neq e - 1)$ {
  $u = ee[e - 1].u, v = ee[e - 1].v, ui = ee[e - 1].uinx, vi = ee[e - 1].vinx$;
  $ee[j] = ee[e - 1]$;
  $adjinx[v][ui] = j$;
  $adjinx[u][vi] = j$;
}

This code is used in section 12.

16. Doing it. Now we've got the basic mechanisms nicely in place, so we just need to use them.

⟨ Record each eligible edge between u of degree $d1$ and v of degree $d2$ 16 ⟩ ≡

```
if (dglen[d1] ∧ dglen[d2]) {
  for (i = dglen[d1] − 1; i ≥ 0; i−−) {
    u = dg[d1][i];
    ⟨ Stamp all vertices at distance < 5 from u 18 ⟩;
    for (j = (d1 ≡ d2 ? i − 1 : dglen[d2] − 1); j ≥ 0; j−−) {
      v = dg[d2][j];
      if ((stamp[v] ≠ curstamp) ∧ (time ≥ wait[u][v])) uelig[p] = u, velig[p] = v, p++;
    }
  }
}
```

This code is used in section 1.

17. Reachability from u is conveniently monitored by using a unique stamp. We also use a *queue* for a breadth-first search.

⟨ Global variables 3 ⟩ +≡

```
int stamp[maxn];   /* the most recent stamp received by each vertex */
int curstamp;
int queue[maxn];   /* elements known to be close to u, in order of distance */
int vbose;   /* this variable can be set positive while debugging */
```

18. ⟨ Stamp all vertices at distance < 5 from u 18 ⟩ ≡

```
{
  register int front, rear, nextfront;

  curstamp++;   /* advance to a unique new stamp value */
  if (curstamp ≡ 0) {
    fprintf(stderr, "Hey, you better give up!\n");
    exit(−666);   /* more than four billion failures */
  }
  queue[0] = u, front = 0, rear = 1, stamp[u] = curstamp;   /* u goes in the queue */
  for (k = 0; k < 4; k++) {
    for (nextfront = rear; front < nextfront; front++) {
      uu = queue[front];   /* a vertex at distance k from u */
      for (t = deg[uu] − 1; t ≥ 0; t−−) {
        w = adj[uu][t];
        if (stamp[w] ≠ curstamp) stamp[w] = curstamp, queue[rear++] = w;
            /* w is at distance k + 1 */
      }
    }
  }
}
```

This code is used in section 16.

19. ⟨ Insert a random eligible edge 19 ⟩ ≡

```
j = gb_unif_rand(p);
if (vbose) fprintf(stderr, "%d: Inserting %d--%d (%d,%d; %d elig)\n", time, uelig[j], velig[j],
      deg[uelig[j]], deg[velig[j]], p);
insert(uelig[j], velig[j]);
time++;
```

This code is used in section 1.

20. **#define** *embargo* 10

⟨ Delete a random edge 20 ⟩ ≡

```
j = gb_unif_rand(e);
if (vbose) fprintf(stderr, "%d: Deleting %d--%d (%d present)\n", time, ee[j].u, ee[j].v, e);
delete(j);
wait[ee[j].u][ee[j].v] = wait[ee[j].v][ee[j].u] = time + embargo;
```

This code is used in section 1.

21. ⟨ Output the solution 21 ⟩ ≡

```
for (j = 0; j < e; j++) printf("%d %d\n", ee[j].u, ee[j].v);
```

This code is used in section 1.

22. Index.

RAND-D4G6