November 24, 2020 at 13:23

1. Hamiltonian cycles. This program finds all Hamiltonian cycles of an undirected graph. [It's a slight revision of the program published in my paper "Mini-indexes for literate programs," *Software—Concepts and Tools* **15** (1994), 2–11.] The input graph should be in Stanford GraphBase format, and should be named on the command line as, for example, `foo.gb`. An optional second command-line parameter is a modulus m, which causes every mth solution to be printed.

We use a utility field to record the vertex degrees.

```
#define deg u.I
#include "gb_graph.h"   /* the GraphBase data structures */
#include "gb_save.h"   /* restore_graph */
  main(int argc, char *argv[])
  {
    Graph *g;
    Vertex *x, *y, *z, *tmax;
    register Vertex *t, *u, *v;
    register Arc *a, *aa;
    register int d;
    Arc *b, *bb;
    int count = 0;
    int dmin, modulus;
    ⟨Process the command line, inputting the graph 2⟩;
    ⟨Prepare g for backtracking, and find a vertex x of minimum degree 3⟩;
    for (v = g→vertices; v < g→vertices + g→n; v++) printf("␣%d", v→deg);
    printf("\n");   /* TEMPORARY CHECK */
    if (x→deg < 2) {
      printf("The␣minimum␣degree␣is␣%d␣(vertex␣%s)!\n", x→deg, x→name);
      return −1;
    }
    for (b = x→arcs; b→next; b = b→next)
      for (bb = b→next; bb; bb = bb→next) {
        v = b→tip;
        z = bb→tip;
        ⟨Find all simple paths of length g→n − 2 from v to z, avoiding x 4⟩;
      }
    printf("Altogether␣%d␣solutions.\n", count);
    for (v = g→vertices; v < g→vertices + g→n; v++) printf("␣%d", v→deg);
    printf("\n");   /* TEMPORARY CHECK, SHOULD AGREE WITH FORMER VALUES */
  }
```

2. ⟨Process the command line, inputting the graph 2⟩ ≡

```
  if (argc > 1) g = restore_graph(argv[1]); else g = Λ;
  if (argc < 3 ∨ sscanf(argv[2], "%d", &modulus) ≠ 1) modulus = 1000000000;
  if (¬g ∨ modulus ≤ 0) {
    fprintf(stderr, "Usage:␣%s␣foo.gb␣[modulus]\n", argv[0]);
    exit(−1);
  }
```

This code is used in section 1.

3. Vertices that have already appeared in the path are "taken," and their *taken* field is nonzero. Initially we make all those fields zero.

#define *taken* $v.I$

⟨ Prepare g for backtracking, and find a vertex x of minimum degree 3 ⟩ ≡

```
dmin = g->n;
for (v = g->vertices; v < g->vertices + g->n; v++) {
  v->taken = 0;
  d = 0;
  for (a = v->arcs; a; a = a->next) d++;
  v->deg = d;
  if (d < dmin) dmin = d, x = v;
}
```

This code is used in section 1.

4. The data structures. I use one simple rule to cut off unproductive branches of the search tree: If one of the vertices we could move to next is adjacent to only one other unused vertex, we must move to it now.

The moves will be recorded in the vertex array of g. More precisely, the kth vertex of the path will be $t\to vert$ when t is the kth vertex of the graph. If the move was not forced, $t\to ark$ will point to the Arc record representing the edge from $t\to vert$ to $(t+1)\to vert$; otherwise $t\to ark$ will be Λ.

This program is a typical backtrack program. I am more comfortable doing it with labels and goto statements than with while loops, but some day I may learn my lesson.

#define *vert* $w.V$
#define *ark* $x.A$

⟨ Find all simple paths of length $g\to n-2$ from v to z, avoiding x 4 ⟩ ≡
$t = g\to vertices$; $tmax = t + g\to n - 1$;
$x\to taken = 1$; $t\to vert = x$;
$t\to ark = \Lambda$;
advance: ⟨ Increase t and update the data structures to show that vertex v is now taken; **goto** *backtrack* if no further moves are possible 5 ⟩;
try: ⟨ Look at edge a and its successors, advancing if it is a valid move 7 ⟩;
restore: ⟨ Downdate the data structures to the state they were in when level t was entered 6 ⟩;
backtrack: ⟨ Decrease t, if possible, and try the next possibility; or **goto** *done* 8 ⟩;
done:

This code is used in section 1.

5. ⟨ Increase t and update the data structures to show that vertex v is now taken; **goto** *backtrack* if no further moves are possible 5 ⟩ ≡

```
t++;
t→vert = v;
v→taken = 1;
if (v ≡ z) {
  if (t ≡ tmax) ⟨ Record a solution 9 ⟩;
  goto backtrack;
}
for (aa = v→arcs, y = Λ; aa; aa = aa→next) {
  u = aa→tip;
  d = u→deg − 1;
  if (d ≡ 1 ∧ u→taken ≡ 0) {
    if (y) goto restore;    /* restoration will stop at aa */
    y = u;
  }
  u→deg = d;
}
if (y) {
  t→ark = Λ;
  v = y;
  goto advance;
}
a = v→arcs;
```

This code is used in section 4.

6. ⟨ Downdate the data structures to the state they were in when level t was entered 6 ⟩ ≡

```
for (a = t→vert→arcs; a ≠ aa; a = a→next) a→tip→deg++;
```

This code is used in section 4.

7. ⟨ Look at edge a and its successors, advancing if it is a valid move 7 ⟩ ≡

```
while (a) {
   v = a→tip;
   if (v→taken ≡ 0) {
      t→ark = a;
      goto advance;
   }
   a = a→next;
}
restore_all: aa = Λ;      /* all moves tried; we fall through to restore */
```

This code is used in section 4.

8. ⟨ Decrease t, if possible, and try the next possibility; or **goto** *done* 8 ⟩ ≡

```
t→vert→taken = 0;
t−−;
if (t→ark) {
   a = t→ark→next;
   goto try;
}
if (t ≡ g→vertices) goto done;
goto restore_all;      /* the move was forced */
```

This code is used in section 4.

9. ⟨ Record a solution 9 ⟩ ≡

```
{
   count++;
   if (count % modulus ≡ 0) {
      printf("%d:␣", count);
      for (u = g→vertices; u ≤ tmax; u++) printf("%s␣", u→vert→name);
      printf("\n");
   }
}
```

This code is used in section 5.

10. Index.

HAM