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

1. Symmetric Hamiltonian cycles. This program finds all Hamiltonian cycles of an undirected graph in which the mapping $v \mapsto N-1-v$ is an automorphism, such that the same automorphism also applies to the cycle.

We use a utility field to record the vertex degrees.

```
#define deg u.I
#define mm 8 /* should be even */
#define nn 9
#include "gb_graph.h" /* the GraphBase data structures */
#include "gb_basic.h" /* standard graphs */
main()
{
  Graph *g = board(mm, nn, 0, 0, 5, 0, 0); /* knight moves on rectangular chessboard */
  Vertex *x, *z, *tmax;
  register Vertex *t, *u, *v;
  register Arc *a, *aa, *yy;
  register int d;
  Arc *b, *bb;
  int count = 0, dcount = 0;
  int dmin;
  ⟨Reduce g to half size 2⟩;
  ⟨Prepare g for backtracking, and find a vertex x of minimum degree 4⟩;
  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) {
      a = b;
      z = bb→tip;
      ⟨Find all simple paths of length g→n − 2 from a→tip to z, avoiding x 5⟩;
    }
  printf("Altogether␣%d␣solutions␣and␣%d␣wannabees.\n", count, dcount);
  for (v = g→vertices; v < g→vertices + g→n; v++) printf("␣%d", v→deg);
  printf("\n"); /* TEMPORARY CHECK, SHOULD AGREE WITH FORMER VALUES */
}
```

2. We identify each vertex with its symmetric mate, and set the length of an arc to 1 if the arc crosses to the mate instead of staying in the same class.

Multiple arcs and self-loops can be introduced in this step.

```
#define mate(v)
        (Vertex *)(((unsigned long) g→vertices) + ((unsigned long)(g→vertices + g→n − 1)) − ((unsigned
            long) v))
⟨ Reduce g to half size 2 ⟩ ≡
  for (v = g→vertices; mate(v) > v; v++)
    for (a = v→arcs; a; a = a→next) {
      u = mate(a→tip);
      if (u > a→tip) a→len = 0;
      else {
        a→len = 1;
        a→tip = u;
      }
    }
  g→n /= 2;
```

This code is used in section 1.

3. Self-loops caused a subtle bug (my test for $v\rightarrow deg \equiv 1$ below failed), and they are of no interest in Hamiltonian circuits. So I'm now getting rid of them.

```
⟨ Remove self-loops 3 ⟩ ≡
  for (v = g→vertices; v < g→vertices + g→n; v++)
    for (a = v→arcs, aa = Λ; a; a = a→next)
      if (a→tip ≡ v) {
        if (aa) aa→next = a→next;
        else v→arcs = a→next;
      }
      else aa = a;
```

This code is used in section 4.

4. 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 4 ⟩ ≡
  ⟨ Remove self-loops 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.

5. 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 arc of the path will be $t\rightarrow ark$ when t is the kth vertex of the graph.

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 ark  x.A
⟨ Find all simple paths of length g→n − 2 from a→tip to z, avoiding x 5 ⟩ ≡
  v = a→tip;
  t = g→vertices; tmax = t + g→n − 1;
  x→taken = 1;
  a→len += 4;      /* the first move is "forced" */
advance: ⟨ Increase t and update the data structures to show that vertex v is now taken; goto backtrack if
      no further moves are possible 6 ⟩;
  try : ⟨ Look at edge a and its successors, advancing if it is a valid move 8 ⟩;
restore: ⟨ Downdate the data structures to the state they were in when level t was entered 7 ⟩;
backtrack: ⟨ Decrease t, if possible, and try the next possibility; or goto done 9 ⟩;
  done:
```

This code is used in section 1.

6. ⟨ Increase t and update the data structures to show that vertex v is now taken; **goto** *backtrack* if no further moves are possible 6 ⟩ ≡

```
  t→ark = a;
  t++;
  v = a→tip;
  v→taken = 1;
  if (v ≡ z) {
    if (t ≡ tmax ∧ v→deg ≡ 1) ⟨ Record a solution 10 ⟩;
    goto backtrack;
  }
  yy = Λ;      /* yy is a forced arc, if any exist */
  for (aa = v→arcs; aa; aa = aa→next) {
    u = aa→tip;
    d = u→deg − 1;
    if (d ≡ 1 ∧ u→taken ≡ 0) {
      if (yy) goto restore;      /* restoration will stop at aa */
      yy = aa;
    }
    u→deg = d;
  }
  if (yy) {
    a = yy;
    a→len += 4;
    goto advance;
  }
  a = v→arcs;
```

This code is used in section 5.

7. ⟨ Downdate the data structures to the state they were in when level t was entered 7 ⟩ ≡

```
for (a = (t − 1)→ark→tip→arcs; a ≠ aa; a = a→next) a→tip→deg++;
```

This code is used in section 5.

8. ⟨ Look at edge a and its successors, advancing if it is a valid move 8 ⟩ ≡

```
while (a) {
  if (a→tip→taken ≡ 0) {
    a→len += 2;      /* oops, this is unnecessary residue of SHAMR */
    goto advance;
  }
  a = a→next;
}
restore_all: aa = Λ;      /* all moves tried; we fall through to restore */
```

This code is used in section 5.

9. Here we come to a subtle point. When a forced move is a duplicated arc, we want to continue with the duplicate arc; we don't want to backtrack!

But that isn't the most subtle part. It turns out that we want to consider the duplicate arc *previous* to the one that worked. (That one should really have been considered forced; if on the other hand the first of two duplicate arcs is selected, the second one will decrease the degree to zero and cannot lead to a complete tour, so we don't want to reconsider it.) Get it? The present logic works only when there are at most two duplicate arcs.

⟨ Decrease t, if possible, and try the next possibility; or **goto** *done* 9 ⟩ ≡

```
t−−;
a = t→ark;
a→tip→taken = 0;
d = a→len;
a→len &= 1; if (d < 4) { a = a→next; goto
try ;
}
if (t ≡ g→vertices) goto done;
for (aa = (t − 1)→ark→tip→arcs; aa ≠ a; aa = aa→next)
  if (aa→tip ≡ a→tip) {
    aa→len += 4;
    a = aa;
    goto advance;
  }
goto restore_all;      /* the move was forced */
```

This code is used in section 5.

10. ⟨ Record a solution 10 ⟩ ≡

```
{
  int s = 0;
  for (u = g→vertices; u < tmax; u++) s ⊕= u→ark→len & 1;
  if (s) {
    count++;
    if (count % 100000 ≡ 0) {      /* use 100000 for the 8 × 8 */
      printf("%d:␣%s", count, x→name);
      for (u = g→vertices; u < tmax; u++) printf("%s%s", u→ark→len & 1 ? "*" : "␣", u→ark→tip→name);
      printf("\n");
    }
  }
  else {
    dcount++;
    if (dcount % 100000 ≡ 0) {      /* use 1 for small cases */
      printf(">%d:␣%s", dcount, x→name);
      for (u = g→vertices; u < tmax; u++) printf("%s%s", u→ark→len & 1 ? "*" : "␣", u→ark→tip→name);
      printf("\n");
    }
  }
}
```

This code is used in section 6.

11. Index.

SHAM