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

1. Intro. This program finds all the Hamiltonian cycles of a given graph. It uses an interesting algorithm that chooses the edges of subpaths without knowing where those edges will appear in the final cycle until they are all linked together. (That important idea was introduced in Geoffrey Selby's thesis (1970), and extended by Silvano Martello in 1983 to incorporate the MRV branching heuristic. I rediscovered Selby's approach independently in 2001, in a slightly more symmetrical form; in my variant, all subpaths have equal status, so that we needn't branch only on the ways of extending one end of a principal subpath.) My first implementation, HAMDANCE, used dancing links; here I'm using sparse-set data structures instead.

As in other programs such as SSXCC, this one reports the running time in "mems." One mem is counted whenever we read or write a 64-bit word of memory, but not when we access data that's already in a register. (The number of mems reported does not include the work that we do when inputting the graph or printing the results.)

```
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define O "%" /* used for percent signs in format strings */
#define mod % /* used for percent signs denoting remainder in C */
#define maxn 1000 /* at most this many vertices */
#define infty maxn /* larger than any vertex number */
#include "gb_graph.h" /* use the Stanford GraphBase conventions */
#include "gb_save.h" /* and its routine for inputting graphs */
#include "gb_flip.h" /* and its random number generator */
  ⟨ Preprocessor definitions ⟩
  typedef unsigned long long ullng; /* a convenient abbreviation */
  ⟨ Type definitions 7 ⟩;
  ⟨ Global variables 2 ⟩
  Graph *g; /* the given graph */
  ⟨ Subroutines 5 ⟩
  int main(int argc, char *argv[])
  {
    register int i, j, k, d, t, u, v, w;
    ⟨ Process the command line, inputting the graph 3 ⟩;
    ⟨ Prepare the graph for backtracking 21 ⟩;
    imems = mems, mems = 0;
    ⟨ Backtrack through all solutions 29 ⟩;
  done: ⟨ Print the results 4 ⟩;
    exit(0);
  }
```

2. The command line names the graph, which is supplied in a file such as `"foo.gb"` in Stanford GraphBase format. Other options may follow this file name, in order to cause printing of some or all of the solutions, or to provide diagnostic information.

Here's a list of the available options:

- '`v`⟨ integer ⟩' enables or disables various kinds of verbose output on *stderr*, specified as a sum of binary codes such as *show_choices*;
- '`m`⟨ integer ⟩' causes every mth solution to be output (the default is `m0`, which merely counts them);
- '`s`⟨ integer ⟩' causes the algorithm to randomize the input graph data (thus providing some variety, although the solutions are by no means uniformly random);
- '`d`⟨ integer ⟩' sets *delta*, which causes periodic state reports on *stderr* after the algorithm has performed approximately *delta* mems since the previous report (default 10000000000);
- '`t`⟨ positive integer ⟩' causes the program to stop after this many solutions have been found;
- '`T`⟨ integer ⟩' sets *timeout* (which causes abrupt termination if *mems* > *timeout* at the beginning of a level);

```
#define show_basics 1 /* vbose code for basic stats; this is the default */
#define show_choices 2 /* vbose code for backtrack logging */
#define show_details 4 /* vbose code for further commentary */
#define show_raw_sols 64 /* vbose code to show solutions in order of arcs added */
#define show_profile 128 /* vbose code to show the search tree profile */
#define show_full_state 256 /* vbose code for complete state reports */
```

⟨ Global variables 2 ⟩ ≡

```
int random_seed = 0; /* seed for the random words of gb_rand */
int randomizing; /* has option 's' been specified? */
int vbose = show_basics; /* level of verbosity */
int spacing; /* solution k is output if k is a multiple of spacing */
int maxl; /* maximum level actually reached */
ullng count; /* solutions found so far */
ullng imems, rmems, mems; /* mem counts */
ullng delta = 10000000000; /* report every delta or so mems */
ullng thresh = 10000000000; /* report when mems exceeds this, if delta ≠ 0 */
ullng maxcount = #ffffffffffffffff; /* stop after finding this many solutions */
ullng timeout = #1fffffffffffffff; /* give up after this many mems */
ullng nodes; /* total size of search tree */
ullng profile[maxn]; /* number of nodes at each level of the search tree */
int nn; /* number of vertices in the given graph */
int mind; /* smallest degree in the given graph */
```

See also sections 6, 8, 11, 14, 18, 22, 26, 30, and 48.

This code is used in section 1.

3. If an option appears more than once on the command line, the first appearance takes precedence.

```
⟨ Process the command line, inputting the graph 3 ⟩ ≡
  for (j = argc − 1, k = 0; j > 1; j−−)
    switch (argv[j][0]) {
    case 'v': k |= (sscanf(argv[j] + 1, ""O"d", &vbose) − 1); break;
    case 'm': k |= (sscanf(argv[j] + 1, ""O"d", &spacing) − 1); break;
    case 's': k |= (sscanf(argv[j] + 1, ""O"d", &random_seed) − 1), randomizing = 1; break;
    case 'd': k |= (sscanf(argv[j] + 1, ""O"lld", &delta) − 1), thresh = delta; break;
    case 't': k |= (sscanf(argv[j] + 1, ""O"lld", &maxcount) − 1); break;
    case 'T': k |= (sscanf(argv[j] + 1, ""O"lld", &timeout) − 1); break;
    default: k = 1;      /* unrecognized command-line option */
    }
  if (argc < 2) k = 1;
  if (k ≡ 0) {
    g = restore_graph(argv[1]);
    if (¬g) {
      fprintf(stderr, "I couldn't reconstruct graph "O"s!\n", argv[1]);
      k = 1;
    } else {
      nn = g→n;
      if (nn > maxn) {
        fprintf(stderr, "Sorry, graph "O"s has too many vertices ("O"d>"O"d)!\n", argv[1], nn,
            maxn);
        exit(−2);
      }
    }
  }
  if (k) {
    fprintf(stderr, "Usage: "O"s foo.gb [v<n>] [m<n>] [s<n>] [d<n>] [t<n>] [T<n>]\n", argv[0]);
    exit(−1);
  }
  if (randomizing) gb_init_rand(random_seed);
```

This code is used in section 1.

4.

```
⟨ Print the results 4 ⟩ ≡
  if (vbose & show_profile) ⟨ Print the profile 54 ⟩;
  if (vbose & show_basics) {
    fprintf(stderr, "Altogether "O"llu solution"O"s, "O"llu nodes,",
        count, count ≡ 1 ? "" : "s", nodes);
    fprintf(stderr, " "O"llu+"O"llu mems.\n", imems, mems);
  }
```

This code is used in section 1.

5. To help detect faulty reasoning, we provide a routine that we hope is never invoked.

```
⟨ Subroutines 5 ⟩ ≡
  void confusion(char *m)
  {
    fprintf(stderr, "This can't happen: "O"s!\n", m);
    exit(666);
  }
```

See also sections 9, 12, 16, 20, 24, 28, 33, 45, and 53.

This code is used in section 1.

6. Data structures. This program can be regarded as an algorithm that finds Hamiltonian cycles by starting with a graph g and removing edges until only a cycle is left.

We use a sparse-set representation for g, because such structures are an especially attractive way to maintain the current status of a graph that is continually getting smaller. The idea is to have two arrays, nbr and adj, with one row for each vertex v. If v has d neighbors in g, they're listed (in any order) in the first d columns of $nbr[v]$. And if $nbr[v][k] = u$, where $0 \le k < d$, we have $adj[v][u] = k$; in other words, there's an important invariant relation,

$$nbr[v][adj[v][u]] = u.$$

Neighbors can be deleted by moving them to the right and decreasing d; neighbors can be undeleted by simply increasing d. Furthermore, if u is not a neighbor of v, $adj[v][u]$ has the impossible value $infty$; thus the adj matrix functions also as an adjacency matrix.

```
⟨ Global variables 2 ⟩ +≡
  int nbr[maxn][maxn], adj[maxn][maxn];      /* sparse-set representation of g */
  int degree[maxn];      /* vertex degree in the input graph (for diagnostics only) */
```

7. The edges of g are considered to be pairs of arcs that run in opposite directions. (In other words, the edge $u \mathrel{—} v$ is actually treated as two arcs, $u \longrightarrow v$ and $v \longrightarrow u$.) When an edge is deleted, we often need to delete only one of those arcs, because our algorithm doesn't always depend on both of them.

The algorithm proceeds not only by removing unwanted edges but also by choosing edges that will *not* be removed. Those edges appear in an array e of **edge** structs, each of which has two fields u and v. If the kth chosen edge is $u \mathrel{—} v$, we have $e[k].u = u$ and $e[k].v = v$.

```
⟨ Type definitions 7 ⟩ ≡
  typedef struct edge_struct {
    int u, v;      /* the vertices joined by this edge */
  } edge;
```

See also sections 10, 17, and 25.

This code is used in section 1.

8.
```
⟨ Global variables 2 ⟩ +≡
  edge e[maxn];      /* the edges chosen so far */
  int eptr;      /* we've currently chosen this many edges */
```

9.
```
⟨ Subroutines 5 ⟩ +≡
  void print_edges(void)
  {
    register int k;
    for (k = 0; k < eptr; k++) printf("""O"s--"O"s\n", name(e[k].u), name(e[k].v));
  }
```

10. The chosen edges form one or more subpaths of the final cycle. If

$$v_0 \text{ — } v_1 \text{ — } \cdots \text{ — } v_k$$

is a chosen subpath, where v_0 and v_k participate in only one of the edges chosen so far, we say that v_0 and v_k are "outer" vertices, while $\{v_1, \ldots, v_{k-1}\}$ are "inner." A vertex that's neither outer nor inner is called "bare." Every vertex begins bare, and is eventually clothed. At the end all vertices will have become inner, except for the two vertices of the last-chosen edge; and the chosen edges will be a Hamiltonian cycle.

As the algorithm proceeds, two crucial integer values are associated with every vertex v, namely $mate(v)$ and $deg(v)$. In the chosen subpath above, we have $mate(v_0) = v_k$ and $mate(v_k) = v_0$; that rule defines $mate(v)$ for all outer vertices v. We also define $mate(v) = -1$ if and only if v is bare. The value of $mate(v)$ is undefined when v is an inner vertex, except for the fact that it's nonnegative.

If v is an outer vertex or a bare vertex, the value of $deg(v)$ is the number of unchosen edges touching v that haven't yet been ruled out for the final path. (Again, $deg(v)$ is undefined if v is inner; an inner vertex is essentially invisible to the algorithm.)

The current values of $mate(v)$ and $deg(v)$ are maintained in a **vert** struct, so that we can conveniently access both of them at once.

```
#define mate(v) vrt[v].m
#define deg(v) vrt[v].d
⟨ Type definitions 7 ⟩ +≡
  typedef struct vert_struct {
    int m, d;      /* the mate and deg of this vertex */
  } vert;
```

11.
```
⟨ Global variables 2 ⟩ +≡
  vert vrt[maxn];
```

12.
```
⟨ Subroutines 5 ⟩ +≡
  void print_vert(int v)
  {
    register int k;
    printf(""O"s:", name(v));
    for (k = 0; ; k++) {
      if (k ≡ deg(v)) printf("|"); else printf("␣");
      if (k ≡ degree[v]) break;
      printf(""O"s", name(nbr[v][k]));
    }
    if (mate(v) < 0) printf("␣bare\n");
    else if (ivis[v] ≥ visible) printf("␣mate␣"O"s,␣inner\n", name(mate(v)));
    else printf("␣mate␣"O"s\n", name(mate(v)));
  }
  void print_verts(void)
  {
    register int v;
    for (v = 0; v < nn; v++) {
      printf(""O"d,", v);
      print_vert(v);
    }
  }
```

13. As mentioned above, an inner vertex is essentially invisible to the algorithm. An array *vis* lists the visible vertices — those that are either bare or outer. It's a sparse-set representation, containing a permutation of the vertices, with the invisible ones at the end. The inverse permutation appears in *ivis*, a companion array, so that we have

$$vis[k] = v \qquad \Leftrightarrow \qquad ivis[v] = k.$$

Vertex v is visible if and only if $ivis[v] < visible$; thus v is inner if and only if $ivis[v] \ge visible$.

```
#define makeinner(v)
        { register int vv, k;
          o, vv = vis[--visible];
          o, k = ivis[v];
          oo, vis[visible] = v, ivis[v] = visible;
          oo, vis[k] = vv, ivis[vv] = k;
        }
```

14. ⟨ Global variables 2 ⟩ +≡

```
int vis[maxn], ivis[maxn];      /* sparse-set representation of visibility */
int visible;      /* this many vertices are currently visible */
```

15. ⟨ Initialize 15 ⟩ ≡

```
for (k = 0; k < nn; k++) oo, vis[k] = ivis[k] = k;
visible = nn;
```

See also section 19.

This code is used in section 21.

16. Here's how we remove an existing arc from u to v. We assume that u is visible, and that v is currently a neighbor of u, namely that $adj[u][v] < deg(u)$.

In a production version of this program, the *remove_arc* subroutine can be declared **inline**. Thus we don't charge any extra mems for invoking it.

The test for $k = d$ in this case saves six mems, at the cost of possibly fouling up branch prediction. So it may not be wise.

⟨ Subroutines 5 ⟩ +≡

```
void remove_arc(int u, int v)
{
  register int d, k, w;
  o, d = deg(u) - 1;
  oo, k = adj[u][v];      /* we assume that k ≤ d */
  if (k ≠ d) {
    oo, w = nbr[u][d];
    o, nbr[u][d] = v;
    o, nbr[u][k] = w;
    o, adj[u][v] = d;
    o, adj[u][w] = k;
  }
  o, deg(u) = d;
}
```

17. We maintain a doubly linked list of all the outer vertices in the current partial solution. Each entry of this list is a **pair** struct, containing two pointers *llink* and *rlink*. The list head is a **pair** struct called *head*.

Vertex v is an outer vertex if and only if the pair $act[v]$ is currently in the list reachable from *head*. Putting it into this list is called "activating" v; taking it out is "deactivating" it.

```
⟨ Type definitions 7 ⟩ +≡
  typedef struct pair_struct {
    int llink, rlink;      /* links to left and right in a doubly linked list */
  } pair;
```

18.
```
#define head  maxn      /* address of the list header in the act array */
#define activate(v)
        { register int l = (o, act[head].llink);
          oo, act[l].rlink = act[head].llink = v;
          o, act[v].llink = l, act[v].rlink = head; }
#define deactivate(v)
        { register int l = (o, act[v].llink), r = act[v].rlink;
          oo, act[l].rlink = r, act[r].llink = l;
          makeinner(v); }
⟨ Global variables 2 ⟩ +≡
  pair act[maxn + 1];
```

19.
```
⟨ Initialize 15 ⟩ +≡
  o, act[head].llink = act[head].rlink = head;      /* active list starts empty */
```

20.
```
⟨ Subroutines 5 ⟩ +≡
  void print_actives(void)
  {
    register int v;
    for (v = act[head].rlink; v ≠ head; v = act[v].rlink) printf("␣"O"s", name(v));
    printf("\n");
  }
```

21. One of the command-line options listed above allows randomization of the input graph. Vertex k of our graph corresponds to vertex $perm[k]$ of that one, where $perm[0], \ldots, perm[nn-1]$ is a random permutation.

```
#define name(v) (g→vertices + iperm[v])→name
⟨ Prepare the graph for backtracking 21 ⟩ ≡
  ⟨ Initialize 15 ⟩;
  if (randomizing) {
    for (j = 0; j < nn; j++) {
      mems += 4, k = gb_unif_rand(j + 1);
      ooo, perm[j] = perm[k], perm[k] = j;
    }
    for (j = 0; j < nn; j++) iperm[perm[j]] = j;
  } else for (j = 0; j < nn; j++) perm[j] = iperm[j] = j;
  ⟨ Set up the nbr and adj arrays 23 ⟩;
  for (mind = infty, u = 0; u < nn; u++) {
    if (o, deg(u) < mind) mind = deg(u), curv = u;
    if (deg(u) ≡ 2) o, trigger[trigptr++] = u;
    for (v = 0; v < nn; v++)
      if (u ≠ v) {
        if (adj[u][v] ≠ infty ∧ adj[v][u] ≡ infty) {
          fprintf(stderr, "graph␣"O"s␣is␣directed␣at␣"O"s␣and␣"O"s!\n", argv[1], name(u),
              name(v));
          exit(−5);
        }
      }
  }
  if (mind < 2) {
    printf("There␣are␣no␣Hamiltonian␣cycles,␣because␣"O"s␣has␣degree␣"O"d!\n", name(curv),
        mind);
    exit(0);
  }
  fprintf(stderr, "OK,␣I've␣got␣a␣graph␣with␣"O"d␣vertices,␣"O"ld␣edges,\n", nn, (g→m)/2);
  fprintf(stderr, "␣and␣minimum␣degree␣"O"d.\n", mind);
```

This code is used in section 1.

22.

```
⟨ Global variables 2 ⟩ +≡
  int perm[maxn];      /* vertex mapping between this program and the input graph */
  int iperm[maxn];     /* the inverse mapping */
```

23. ⟨ Set up the *nbr* and *adj* arrays 23 ⟩ ≡

```
for (i = 0; i < nn; i++)
  for (o, j = 0; j < nn; j++) o, adj[i][j] = infty;
for (v = 0; v < nn; v++) {
  register int up, vp;
  register Arc *a;
  rmems++, vp = perm[v];
  oo;     /* mems to fetch nbr[vp] and adj[vp], needed in the following loop */
  for (d = 0, o, a = (g→vertices + v)→arcs; a; o, a = a→next, d++) {
    o, u = a→tip - g→vertices;
    if (u ≡ v) {
      fprintf(stderr, "graph␣"O"s␣has␣a␣self␣loop␣"O"s--"O"s!\n", argv[1],
          (g→vertices + v)→name, (g→vertices + u)→name);
      exit(-44);
    }
    rmems++, up = perm[u];
    if (adj[vp][up] ≠ infty) {
      fprintf(stderr, "graph␣"O"s␣has␣a␣repeated␣edge␣"O"s--"O"s!\n", argv[1],
          (g→vertices + v)→name, (g→vertices + u)→name);
      exit(-4);
    }
    oo, nbr[vp][d] = up, adj[vp][up] = d;
  }
  o, mate(vp) = -1, deg(vp) = degree[vp] = d;
  if (randomizing) {     /* permute the list of neighbors */
    for (j = 1; j < d; j++) {
      mems += 4, k = gb_unif_rand(j + 1);
      oo, u = nbr[vp][j], w = nbr[vp][k];
      oo, nbr[vp][j] = w, nbr[vp][k] = u;
      oo, adj[vp][w] = j, adj[vp][u] = k;
    }
  }
}
if (randomizing) mems += rmems;     /* rmems are ignored if perm is the identity */
```

This code is used in section 21.

24. Here's a subroutine that painstakingly doublechecks the integrity of the data structures in their current state. It does not, however, attempt to be bulletproof. For example, it assumes that links in the *act* array aren't out of bounds. It doesn't even bother to check that *vis* and *ivis* are inverse permutations.

```
#define sanity_checking 0 /* set this to 1 if you suspect a bug */
⟨ Subroutines 5 ⟩ +≡
  void sanity(void)
  {
    register int u, v, pv, k;
    for (pv = head, v = act[pv].rlink; v ≠ head; pv = v, v = act[pv].rlink) {
      if (act[v].llink ≠ pv) fprintf(stderr, "llink of "O"s is bad!\n", name(v));
      if (ivis[v] ≥ visible) fprintf(stderr, "active "O"s is invisible!\n", name(v));
      u = mate(v);
      if (u < 0) fprintf(stderr, "active "O"s has no mate!\n", name(v));
      else if (u ≥ nn) fprintf(stderr, "active "O"s has bad mate!\n", name(v));
      else if (mate(u) ≠ v) fprintf(stderr, "mate(mate("O"s))!="O"s!\n", name(v), name(v));
      else if (adj[v][u] < deg(v))
        fprintf(stderr, "there's an arc from "O"s to its mate!\n", name(v));
    }
    if (act[head].llink ≠ pv) fprintf(stderr, "llink of head is bad!\n");
    for (v = 0; v < nn; v++) {
      for (k = 0; k < degree[v]; k++)
        if (adj[v][nbr[v][k]] ≠ k) fprintf(stderr, "Bad nbr["O"s]["O"d]!\n", name(v), k);
      for (u = 0; u < nn; u++)
        if (adj[v][u] ≠ infty ∧ nbr[v][adj[v][u]] ≠ u)
          fprintf(stderr, "Bad adj["O"s]["O"s]!\n", name(v), name(u));
      if (ivis[v] < visible ∧ eptr < nn) {      /* v is outer or bare */
        for (k = 0; k < deg(v); k++) {
          u = nbr[v][k];
          if (ivis[u] ≥ visible)
            fprintf(stderr, "inner "O"s is touched by "O"s!\n", name(u), name(v));
          else if (adj[u][v] ≥ deg(u))
            fprintf(stderr, "arc "O"s to "O"s is missing!\n", name(u), name(v));
        }
      }
    }
  }
```

25. Nodes, stacks, and the trigger list. Our backtrack process corresponds to traversing the nodes of a search tree, and we control that traversal by maintaining status information in an array of **node** structs. On the current level, that info is in $nd[level]$; and we'll eventually be resuming what we were doing in $nd[level-1], \ldots, nd[0]$.

Thus nd is a stack that helps to control this algorithm.

```
⟨ Type definitions 7 ⟩ +≡
  typedef struct node_struct {
    int v;      /* the active vertex curv on which we're branching */
    int m;      /* the number of edges chosen so far */
    int i;      /* the index curi of curv's current neighbor curu */
    int d;      /* the total number deg(curv) of possibilities for curi */
    int s;      /* the number of visible vertices */
    int t;      /* base position in the trigger list (see below) */
    int a;      /* base position in the active stack (see below) */
  } node;
```

26. Two more stacks act in parallel with nd, but grow at different rates, namely *savestack* (which records the mates and degrees of vertices) and *actstack* (which records which vertices were active). The *savestack* grows by exactly nn entries at each level.

```
⟨ Global variables 2 ⟩ +≡
  int level;      /* the depth of branching */
  node nd[maxn];      /* nodes between current level and the search tree root */
  int trigger[maxn * maxn];      /* vertices whose degree became 2 while bare */
  int trigptr;      /* the number of vertices in the trigger lists */
  vert savestack[maxn * maxn];      /* data for the visible vertices at each level */
  int saveptr;      /* number of entries on savestack */
  int actstack[maxn * maxn];      /* lists of active vertices at each level */
  int actptr;      /* number of entries on actstack */
```

27. If a bare vertex v has degree 2 in the current graph, every Hamiltonian cycle must contain the two edges that touch v. We put v into a list called *trigger*, because we want it to force those edges to be chosen as soon as we have a chance to do so.

28. We call *removex* instead of *remove_arc* when u might be bare, because *removex* will make u a trigger at the appropriate time.

(Note: Sometimes *remove_arc* is called when u is bare but will soon become outer. It's more efficient to do that than to use *removex* in all cases.)

```
⟨ Subroutines 5 ⟩ +≡
  void removex(int u, int v)
  {
    register int d, k, w;
    o, d = deg(u) − 1;
    if (mate(u) < 0 ∧ d ≡ 2) o, trigger[trigptr++] = u;
    oo, k = adj[u][v];      /* we assume that k ≤ d */
    if (k ≠ d) {
      oo, w = nbr[u][d];
      o, nbr[u][d] = v;
      o, nbr[u][k] = w;
      o, adj[u][v] = d;
      o, adj[u][w] = k;
    }
    o, deg(u) = d;
  }
```

29. Marching forward. Here we follow the usual pattern of a backtrack process (and I follow my usual practice of **goto**-ing). In this particular case it's a bit tricky to get the whole process started, so I'm deferring that bootstrap calculation until the program for nonroot levels is in place and understood.

```
⟨ Backtrack through all solutions 29 ⟩ ≡
  ⟨ Bootstrap the backtrack process 49 ⟩;
advance: ⟨ Clothe everything on the trigger list, or goto try_again 31 ⟩;
  if (sanity_checking) sanity( );
  nodes++, level++;
  if (level > maxl) maxl = level;
  if (vbose & show_profile) profile[level]++;
  if (vbose & show_details) fprintf(stderr, "Entering␣level␣"O"d:\n", level);
  if (eptr ≥ nn − 1) ⟨ Check for solution and goto backup 44 ⟩;
  ⟨ Do special things if enough mems have accumulated 52 ⟩;
  ⟨ Set curv to an outer vertex of minimum degree d 37 ⟩;
  if (d ≡ 0) goto backup;
  e[eptr].u = curv;      /* no mem charged for the e array */
  o, trigptr = nd[level − 1].t;
  ⟨ Promote curv from outer to inner 38 ⟩;
  if (sanity_checking) sanity( );
  curi = 0;
  ⟨ Record the current status, for backtracking later 40 ⟩;
try_move: ⟨ Choose the edge from curv to nbr[curv][curi] 39 ⟩;
  goto advance;
backup: if (−−level < 0) goto done;
  if (vbose & show_details) fprintf(stderr, "Back␣to␣level␣"O"d\n", level);
try_again: ⟨ Restore d and curi, increasing curi 41 ⟩;
  if (curi ≥ d) {
    if (level) goto backup;
    goto done;
  }
  ⟨ Undo the other changes made at the current level 42 ⟩;
  if (level) {
    if (sanity_checking) sanity( );
    goto try_move;
  }
  ⟨ Advance at root level 50 ⟩;
```

This code is used in section 1.

30. ⟨ Global variables 2 ⟩ +≡

```
  int curt, curu, curv, curw;      /* current vertices of interest */
  int curi;      /* index of the neighbor currently chosen */
```

31. Here's where we force edges to be in the cycle, because some bare vertex of degree 2 had entered the trigger list. As we work through that list, the situation might have changed, because the formerly bare vertex may have become active.

Indeed, giving clothes to one bare vertex might have a ripple effect, causing other bare vertices to enter the trigger list. The value of *trigptr* in the following loop might therefore be a moving target.

One case needs to handled with special care: If the two neighbors of v are mates of each other, we are forced to complete a cycle. This is legitimate only if the cycle includes all vertices.

When this loop has finished, every remaining bare vertex will have degree 3 or more.

```
#define vprint()
        if (vbose & show_choices)
          fprintf(stderr, "␣␣␣␣␣"O"s--"O"s\n", name(e[eptr - 1].u), name(e[eptr - 1].v));
⟨ Clothe everything on the trigger list, or goto try_again 31 ⟩ ≡
  for (o, j = (level ? nd[level - 1].t : 0); j < trigptr; j++) {
    o, v = trigger[j];
    if (o, mate(v) ≥ 0) continue;      /* v is no longer bare */
    if (deg(v) < 2) {
      if (vbose & show_details) fprintf(stderr, "oops,␣low␣degree␣at␣"O"s\n", name(v));
      goto try_again;
    }
    ooo, u = nbr[v][0], w = nbr[v][1];
    if ((o, mate(u) ≡ w) ∧ eptr ≠ nn - 2) {
      if (vbose & show_details)
        fprintf(stderr, "oops,␣short␣cycle␣"O"s--"O"s--"O"s--..."O"s␣forced\n", name(u),
            name(v), name(w), name(u));
      goto try_again;
    }     /* now v is bare and connected only to u and w, which aren't mates */
    e[eptr].u = u, e[eptr++].v = v; vprint( );      /* the e array is memfree */
    e[eptr].u = v, e[eptr++].v = w; vprint( );
    o, mate(v) = v; makeinner(v);
    if (o, mate(u) < 0) {
      if (o, mate(w) < 0) ⟨ Promote BBB to OIO 32 ⟩
      else ⟨ Promote BBO to OII 35 ⟩;
    } else if (o, mate(w) < 0) ⟨ Promote OBB to IIO 34 ⟩
    else ⟨ Promote to OBO to III 36 ⟩;
  }
```

This code is used in section 29.

32. A subtle point ought to be explained here: Suppose the input graph is simply the complete graph K_3, so that its vertices are $\{0,1,2\}$ and the edges are $0 \mathrel{—} 1 \mathrel{—} 2 \mathrel{—} 0$. Then all vertices go immediately onto the trigger list. The first promotion, $trigger[0] = 0$, will generate two forced edges $1 \mathrel{—} 0$ and $0 \mathrel{—} 2$. Then 1 and 2 will no longer be bare; so they won't actually trigger anything. Moreover, they will be mates; and the edge between them will have been removed (by *makemates*), leaving them with degree 0! But that won't be a problem, because the algorithm never branches after $nn - 1$ edges have been chosen.

⟨ Promote BBB to OIO 32 ⟩ ≡

```
{
  activate(u);
  activate(w);
  remove_arc(u, v);
  remove_arc(w, v);
  makemates(u, w);
}
```

This code is used in section 31.

33. ⟨ Subroutines 5 ⟩ +≡

```
void makemates(int u, int w)
{
  if (ooo, adj[w][u] < deg(w)) {      /* u is a neighbor of w */
    remove_arc(w, u);
    remove_arc(u, w);
  }
  oo, mate(u) = w, mate(w) = u;
}
```

34. ⟨ Promote OBB to IIO 34 ⟩ ≡

```
{
  activate(w);
  remove_arc(w, v);
  for (oo, k = deg(u) − 1; k ≥ 0; k−−) {
    o, t = nbr[u][k];      /* the mem for fetching nbr[u] was charged above */
    if (t ≠ v) removex(t, u);
  }
  o, makemates(mate(u), w);
  deactivate(u);
}
```

This code is used in section 31.

35. ⟨ Promote BBO to OII 35 ⟩ ≡

```
{
  activate(u);
  remove_arc(u, v);
  for (oo, k = deg(w) − 1; k ≥ 0; k−−) {
    o, t = nbr[w][k];      /* the mem for fetching nbr[w] was charged above */
    if (t ≠ v) removex(t, w);
  }
  o, makemates(mate(w), u);
  deactivate(w);
}
```

This code is used in section 31.

36. In the final case, u and w are outer vertices. We have cleverly arranged things so that they are mates if and only if $eptr = nn$, in which case all edges of a Hamiltonian cycle have already been chosen.

(Consider, for example, the case where the input graph is the cyclic graph C_4, with edges $0 — 1 — 2 — 3 — 0$. Then $trigger[0] = 0$ will choose edges $1 — 0$ and $0 — 3$. And $trigger[1] = 1$ will do nothing, because 1 is no longer bare. Then $trigger[2] = 2$ will choose edges $1 — 2$ and $2 — 3$, as desired, in spite of the fact that 1 and 3 are mates. The subsequent $trigger[3] = 3$ will again do nothing.)

```
⟨ Promote to OBO to III 36 ⟩ ≡
  if (eptr ≠ nn) {
    for (oo, k = deg(u) − 1; k ≥ 0; k−−) {
      o, t = nbr[u][k];      /* the mem for fetching nbr[u] was charged above */
      if (t ≠ v) removex(t, u);
    }
    for (oo, k = deg(w) − 1; k ≥ 0; k−−) {
      o, t = nbr[w][k];      /* the mem for fetching nbr[w] was charged above */
      if (t ≠ v) removex(t, w);
    }
    oo, makemates(mate(u), mate(w));
    deactivate(u);
    deactivate(w);
  }
```

This code is used in section 31.

37.

```
⟨ Set curv to an outer vertex of minimum degree d 37 ⟩ ≡
  for (oo, curv = k = act[head].rlink, d = deg(curv); k ≠ head; o, k = act[k].rlink) {
    if (vbose & show_details) fprintf(stderr, "␣"O"s("O"d)", name(k), deg(k));
    if (o, deg(k) < d) curv = k, d = deg(k);
  }
  if (vbose & show_details) fprintf(stderr, ",␣branching␣on␣"O"s("O"d)\n", name(curv), d);
```

This code is used in section 29.

38. The d neighbors of $curv$ will remain in $curv$'s list. But $curv$ will be removed from *their* lists.

```
⟨ Promote curv from outer to inner 38 ⟩ ≡
  for (o, k = 0; k < d; k++) {
    o, u = nbr[curv][k];      /* the mem for fetching nbr[curv] was charged above */
    removex(u, curv);
  }
  deactivate(curv);
```

This code is used in section 29.

39. ⟨ Choose the edge from $curv$ to $nbr[curv][curi]$ 39 ⟩ ≡

```
o, curu = nbr[curv][curi];
o, curw = mate(curv);
e[eptr++].v = curu;
if (vbose & show_choices) fprintf(stderr, ""O"3d:␣"O"s--"O"s␣("O"d␣of␣"O"d)\n", level,
        name(e[eptr − 1].u), name(e[eptr − 1].v), curi + 1, d);
o, curt = mate(curu);
if (curt < 0) {      /* curu is bare */
  makemates(curu, curw);
  activate(curu);
} else {      /* curu is outer */
  makemates(curt, curw);
  for (oo, k = deg(curu) − 1; k ≥ 0; k−−) {
    o, u = nbr[curu][k];      /* the mem for fetching nbr[curu] was charged above */
    removex(u, curu);
  }
  deactivate(curu);
}
```

This code is used in section 29.

40. Backtracking. As we explore the search tree, we often want to go back and investigate the branches not yet taken.

Only one mem is needed to access both $nd[level].v$ and $nd[level].m$ simultaneously, because those 32-bit **int**s occupy the same 64-bit word. A similar remark applies to other pairs of fields.

```
⟨ Record the current status, for backtracking later 40 ⟩ ≡
  {
    o, nd[level].d = d, nd[level].i = curi;
    o, nd[level].s = visible, nd[level].t = trigptr;
    if (d > 1) {
      o, nd[level].v = curv, nd[level].m = eptr;
      saveptr = level * nn;
      for (k = 0; k < visible; k++) {
        o, u = vis[k];
        oo, savestack[saveptr + u] = vrt[u];
      }
      for (o, u = act[head].rlink; u ≠ head; o, u = act[u].rlink) actstack[actptr++] = u;
    }
    o, nd[level].a = actptr;
  }
```

This code is used in sections 29 and 49.

41. Here, as suggested by Peter Weigel, we restore only the two most crucial state variables — because they might tell us that we needn't bother to restore any more.

```
⟨ Restore d and curi, increasing curi 41 ⟩ ≡
  oo, d = nd[level].d, curi = ++nd[level].i;
```

This code is used in section 29.

42.

```
⟨ Undo the other changes made at the current level 42 ⟩ ≡
  for (o, actptr = nd[level].a, v = head, k = (level ? o, nd[level − 1].a : 0); k < actptr; k++) {
    o, u = actstack[k];
    oo, act[v].rlink = u, act[u].llink = v;
    v = u;
  }
  oo, act[v].rlink = head, act[head].llink = v;
  o, visible = nd[level].s, trigptr = nd[level].t;
  saveptr = level * nn;
  for (k = 0; k < visible; k++) {
    o, u = vis[k];
    oo, vrt[u] = savestack[saveptr + u];
  }
  o, curv = nd[level].v, eptr = nd[level].m;
```

This code is used in section 29.

43. Reaping the rewards. Once all vertices have been connected up, no more decisions need to be made. In most such cases, we'll have found a valid Hamiltonian cycle, although its last link usually still needs to be filled in.

At this point, exactly two vertices should be active.

44. ⟨ Check for solution and **goto** *backup* 44 ⟩ ≡

```
{
  if (eptr < nn) {
    ⟨ If the two outer vertices aren't adjacent, goto backup 46 ⟩;
    e[eptr].u = act[head].llink, e[eptr++].v = act[head].rlink; vprint();
  }
  count++;
  if (spacing ∧ count mod spacing ≡ 0) {
    nd[level].i = 0, nd[level].d = 1;
    nd[level].m = eptr;
    if (vbose & show_raw_sols) {
      printf("\n"O"llu:\n", count); print_state(stdout);
    } else ⟨ Unscramble and print the current solution 47 ⟩;
    fflush(stdout);
  }
  if (count ≥ maxcount) goto done;
  goto backup;
}
```

This code is used in section 29.

45. ⟨ Subroutines 5 ⟩ +≡

```
void print_state(FILE *stream)
{
  register int i, j, l;
  for (j = l = 0; l ≤ level; j++, l++) {
    while (j < nd[l].m) {
      fprintf(stream, "␣␣␣␣␣␣"O"s--"O"s\n", name(e[j].u), name(e[j].v));
      j++;
    }
    if (l) {
      if (j < nn) fprintf(stream, "␣"O"3d:␣"O"s--"O"s␣("O"d␣of␣"O"d)\n", l, name(e[j].u),
          name(e[j].v), nd[l].d ≡ 1 ? 1 : nd[l].i + 1, nd[l].d);
    } else ⟨ Print the state line for the root level 51 ⟩;
  }
}
```

46. At this point we've formed a Hamiltonian path, which will be a Hamiltonian cycle if and only if its two *outer* vertices are neighbors.

⟨ If the two *outer* vertices aren't adjacent, **goto** *backup* 46 ⟩ ≡

```
{
  o, u = act[head].llink, v = act[head].rlink;
  if (oo, adj[u][v] ≡ infty) goto backup;
}
```

This code is used in section 44.

47. #**define** *index*(*v*) (*v*) − *g*→*vertices*)

⟨ Unscramble and print the current solution 47 ⟩ ≡

```
{
  register int i, j, k;
  for (k = 0; k < nn; k++) v1[k] = −1;
  for (k = 0; k < nn; k++) {
    i = e[k].u, j = e[k].v;
    if (v1[i] < 0) v1[i] = j;
    else v2[i] = j;
    if (v1[j] < 0) v1[j] = i;
    else v2[j] = i;
  }
  path[0] = 0, path[1] = v1[0];
  for (k = 2; ; k++) {
    if (v1[path[k − 1]] ≡ path[k − 2]) path[k] = v2[path[k − 1]];
    else path[k] = v1[path[k − 1]];
    if (path[k] ≡ 0) break;
  }
  for (k = 0; k ≤ nn; k++) printf(""O"s␣", name(path[k]));
  printf("#"O"llu\n", count);
}
```

This code is used in section 44.

48. ⟨ Global variables 2 ⟩ +≡

```
int v1[maxn], v2[maxn];      /* the neighbors of a given vertex */
int path[maxn + 1];          /* the Hamiltonian cycle, in order */
```

49. Getting started. Our program is almost complete, but we still need to figure out how to get the ball rolling by setting things up properly at backtrack level 0.

There's no problem if the graph has at least one vertex of degree 2, because the *trigger* list will provide us with at least two active vertices in such a case. But if all vertices have degree 3 or more, we've got to have some outer vertices as seeds for the rest of the computation.

In the former (easy) case, we set *curv* to -1. In the latter case, we take a vertex *curv* of minimum degree d; we set $nd[0].v = curv$, and try each neighbor of *curv* in turn. (More precisely, after we've found all Hamiltonian cycles that contain an edge from *curv* to some other vertex, u, we'll remove that edge permanently from the graph, and repeat the process until *curv* or some other vertex has only two neighbors left.)

```
⟨ Bootstrap the backtrack process 49 ⟩ ≡
  level = 0;
  d = mind − 1;
  if (d ≡ 1) curv = −1;
  else {
    curi = 0;
  force: oo, curu = nbr[curv][d − curi];
    e[0].u = curv, e[0].v = curu, eptr = 1;
    if (vbose & show_choices)
      fprintf(stderr, "␣␣0:␣"O"s--"O"s␣("O"d␣of␣"O"d)\n", name(e[0].u), name(e[0].v), curi + 1, d);
    activate(curu);
    activate(curv);
    makemates(curu, curv);
  }
  ⟨ Record the current status, for backtracking later 40 ⟩;
```

This code is used in section 29.

50. Back at root level, all vertices are again bare. Since the edge that was previously tried at root level is now no longer present, one or both of its vertices might now have degree 2; and in such a case the trigger list will provide a way to finish the final round.

```
⟨ Advance at root level 50 ⟩ ≡
  o, curu = mate(curv);      /* the previous edge curv to curu is now gone */
  o, act[head].llink = act[head].rlink = head, actptr = 0;      /* nothing is active */
  oo, mate(curu) = mate(curv) = −1, visible = nn;      /* everything is bare */
  if (deg(curu) ≡ 2) trigger[0] = curu, trigptr = 1; else trigptr = 0;
  if (deg(curv) ≡ 2) trigger[trigptr++] = curv;
  if (trigptr ≡ 0) goto force;
  oo, nd[0].v = −1, nd[0].a = eptr = 0;
  if (vbose & show_choices) fprintf(stderr, "␣␣0:␣("O"d␣of␣"O"d)\n", curi + 1, d);
  goto advance;
```

This code is used in section 29.

51. If $nd[0].v$ is negative, the root level began with its first edges supplied by the trigger list, so there was no "chosen" edge.

```
⟨ Print the state line for the root level 51 ⟩ ≡
  {
    if (nd[0].v ≥ 0 ∨ nd[0].d > 1) fprintf(stream, "␣␣␣0:␣("O"d␣of␣"O"d)\n", nd[0].i + 1, nd[0].d);
    j−−;      /* compensate for j++ in the for loop */
  }
```

This code is used in section 45.

52. Progress reports. It's interesting to watch this algorithm in operation, and we provide several ways for a user to do that.

```
⟨ Do special things if enough mems have accumulated 52 ⟩ ≡
  if (delta ∧ (mems ≥ thresh)) {
    thresh += delta;
    if (vbose & show_full_state) print_state(stderr);
    else print_progress( );
  }
  if (mems ≥ timeout) {
    fprintf(stderr, "TIMEOUT!\n"); goto done;
  }
```

This code is used in section 29.

53. During a long run, it's helpful to have some way to measure progress. The following routine prints a string that indicates roughly where we are in the search tree. The string consists of character pairs, separated by blanks, where each character pair represents a branch of the search tree. When a node has d descendants and we are working on the kth, the two characters respectively represent k and d in a simple code; namely, the values 0, 1, ..., 61 are denoted by

$$\texttt{0, 1, ..., 9, a, b, ..., z, A, B, ...,Z.}$$

All values greater than 61 are shown as '`*`'. Notice that as computation proceeds, this string will increase lexicographically.

Following that string, a fractional estimate of total progress is computed, based on the naïve assumption that the search tree has a uniform branching structure. If the tree consists of a single node, this estimate is .5; otherwise, if the first choice is 'k of d', the estimate is $(k-1)/d$ plus $1/d$ times the recursively evaluated estimate for the kth subtree. (This estimate might obviously be very misleading, in some cases, but at least it tends to grow monotonically.)

```
⟨ Subroutines 5 ⟩ +≡
  void print_progress(void)
  {
    register int l, k, d, p;
    register double f, fd;
    fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"lld␣sols,", mems, count);
    for (f = 0.0, fd = 1.0, l = 0; l < level; l++) {
      d = nd[l].d;
      k = (d ≡ 1 ? 1 : nd[l].i + 1);
      fd *= d, f += (k − 1)/fd;      /* choice l is k of d */
      fprintf(stderr, "␣"O"c"O"c", k < 10 ? '0' + k : k < 36 ? 'a' + k − 10 : k < 62 ? 'A' + k − 36 : '*',
          d < 10 ? '0' + d : d < 36 ? 'a' + d − 10 : d < 62 ? 'A' + d − 36 : '*');
    }
    fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
  }
```

54. ⟨ Print the profile 54 ⟩ ≡

```
  {
    fprintf(stderr, "Profile:\n");
    for (level = 1; level ≤ maxl; level++) fprintf(stderr, ""O"3d:␣"O"lld\n", level, profile[level]);
  }
```

This code is used in section 4.

55. Index.

SSHAM