1. Introduction. This little program finds all the Hamiltonian circuits of a given graph, using an interesting algorithm that illustrates the technique of "dancing links" [see my paper in *Millennial Perspectives in Computer Science*, edited by Jim Davies, Bill Roscoe, and Jim Woodcock (Houndmills, Basingstoke, Hampshire: Palgrave, 2000), 187–214]. The idea is to allow long paths to grow in segments that gradually merge together, instead of to build such paths strictly in order from beginning to end. At each stage in the decision process, certain edges have been chosen to be in the final circuit, with no three touching any vertex; we repeatedly choose further edges, preserving this condition while not completing any cycles that are too short.

```
#include "gb_graph.h"   /* use the Stanford GraphBase conventions */
#include "gb_save.h"   /* and its routine for inputting graphs */
  ⟨Preprocessor definitions⟩
  ⟨Global variables 3⟩
  Graph *g;   /* the given graph */
  ⟨Subroutines 5⟩
  int main(int argc, char *argv[])
  {
    register Vertex *u, *v, *w;
    register Arc *a;
    int k, d;
    ⟨Process the command line, inputting the graph 2⟩;
    ⟨Prepare the graph for backtracking 9⟩;
    ⟨Backtrack through all solutions 14⟩;
    ⟨Print the results 13⟩;
    exit(0);
  }
```

2. The given graph should be in Stanford GraphBase format, in a file like "foo.gb" named on the command line. This file name can optionally be followed by a modulus m, which causes every $|m|$th solution to be printed. If a third command line argument appears, the output will be extremely verbose.

The modulus m might be negative; this indicates that solutions should be printed showing edges in the order they were discovered, rather than in the natural circuit order.

```
#define max_n 100   /* our arrays will accommodate this many vertices at most */
#define infty 1000000000   /* infinity (approximately) */
⟨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 = infty;
  if (¬g ∨ modulus ≡ 0) {
    fprintf(stderr, "Usage:␣%s␣foo.gb␣[[-]modulus]␣[verbose]\n", argv[0]);
    exit(−1);
  }
  if (g→n > max_n) {
    fprintf(stderr, "Sorry,␣I'm␣set␣up␣to␣handle␣at␣most␣%d␣vertices!\n", max_n);
    exit(−2);
  }
  if (argc > 3) verbose = 1;
```

This code is used in section 1.

3. The *verbose* variable is declared in `gb_graph.h`.

⟨ Global variables 3 ⟩ ≡

```
int modulus;  /* how often we should show solutions */
```

See also sections 4, 8, 10, 11, and 35.

This code is used in section 1.

4. Data structures. Each vertex is either *bare* (touching none of the chosen edges) or *outer* (touching just one) or *inner* (touching two). An *outer* vertex has a *mate*, which is the vertex at the other end of the path of chosen vertices that it belongs to. All nonchosen edges that touch inner vertices have effectively been removed from the graph. Any edge that runs from a vertex to its mate has also effectively been removed.

The degree *deg* of a *bare* or *outer* vertex is the number of edges that currently touch it. All vertices begin *bare* and end up *inner*. A bare vertex of degree 2 is converted to an inner vertex, since its two edges must be in the final circuit; this mechanism causes *outer* vertices to spring up more or less spontaneously, and it helps in the decision-making. At moments when all bare vertices have degree 3 or more, we choose an end vertex of minimum degree, and make it inner in all possible ways.

The main data structure is a doubly linked list of all the *outer* vertices. Links in this list are called *llink* and *rlink*. When a vertex is removed from the list, its *llink* and *rlink* retain important information about how to undo this operation when backtracking; this idea makes the links "dance." Similarly, when an *outer* vertex becomes *inner*, its *mate* field retains the name of its former mate, so that we needn't recompute mates when undoing previous changes to the data structures.

The *mate* field of a vertex that was promoted directly from *bare* to *inner* is one of its two neighbors. The other neighbor is stored in another field called *comate*.

Utility fields u, v, w, x, y, and z of a **Vertex** are used to hold the *type*, *deg*, *llink*, *rlink*, *mate*, and *comate*.

```
#define bare 2
#define outer 1
#define inner 0
#define type u.I    /* either bare, outer, or inner */
#define deg v.I    /* current degree, for non-inner vertices */
#define llink w.V    /* link to the left in the basic list */
#define rlink x.V    /* link to the right in the basic list */
#define mate y.V    /* the mate of an outer vertex */
#define comate z.V    /* neighbor of fast-promoted inner vertex */
#define head (&list_head)

⟨Global variables 3⟩ +≡
  Vertex list_head;    /* the doubly linked list starts here */
  char *decode[3] = {"inner","outer","bare"};
```

5. Here's a routine that should be useful for debugging: It displays the fields of a given vertex symbolically.

```
⟨Subroutines 5⟩ ≡
  void print_vert(Vertex *v)
  {
    printf("%s:␣%s,␣deg=%d", v→name, decode[v→type], v→deg);
    if (v→llink) printf(",␣llink=%s", v→llink→name);
    if (v→rlink) printf(",␣rlink=%s", v→rlink→name);
    if (v→mate) printf(",␣mate=%s", v→mate→name);
    if (v→comate) printf(",␣comate=%s", v→comate→name);
    printf("\n");
  }
```

See also sections 6, 7, and 12.

This code is used in section 1.

6. And if we want to see them all:

⟨ Subroutines 5 ⟩ +≡
```
void print_verts()
{
  register Vertex *v;
  for (v = g->vertices; v < g->vertices + g->n; v++) print_vert(v);
}
```

7. Even more important for debugging is the *sanity_check* routine, which painstakingly makes sure that I haven't let the data structure get out of sync with itself. (Vertex *vv* is either Λ or an *inner* vertex whose mate is currently *outer*. In the latter case, some of the sanity checks are not made.)

⟨ Subroutines 5 ⟩ +≡
```
void sanity_check(Vertex *vv)
{
  register Vertex *u, *v, *w;
  register Arc *a;
  register int c, d;
  for (v = g->vertices, c = 0; v < g->vertices + g->n; v++) {
    w = v->mate;
    if (v->type ≡ bare ∧ w ≠ Λ)
      printf("Bare vertex %s shouldn't have mate %s!\n", v->name, w->name);
    if (v->type ≡ outer) c++;
    if (v->type ≡ outer ∧ (w->mate ≠ v ∨ w->type ≠ outer))
      if (w ≠ vv ∨ w->type ≠ inner)
        printf("Outer vertex %s has mate problem vis-a-vis %s!\n", v->name, w->name);
    for (a = v->arcs, d = 0; a; a = a->next) {
      u = a->tip;
      if (u->type ≠ inner ∧ u ≠ w) d++;
    }
    if (v->type ≠ inner ∧ v->deg ≠ d ∧ ocount ≠ g->n − 1)
      printf("Vertex %s should have degree %d, not %d!\n", v->name, d, v->deg);
    if (v->type ≡ bare ∧ d < 3 ∧ vv ≡ Λ)
      printf("Vertex %s (degree %d) should not be bare!\n", v->name, d);
  }
  for (v = head->rlink; c > 0; c−−, v = v->rlink) {
    if (v->type ≠ outer)
      printf("Vertex %s (%s) shouldn't be in the list!\n", v->name, decode[v->type]);
    if (v->llink->rlink ≠ v ∨ v->rlink->llink ≠ v)
      printf("Double-link failure at vertex %s!\n", v->name);
  }
  if (v ≠ head) printf("The list doesn't contain all the outer vertices!\n");
}
```

8. The next most interesting data structure is the *barelist*, which receives the names of *bare* vertices at the moment their degree drops to 2. Such vertices must be clothed before we advance to a new level of backtracking.

⟨ Global variables 3 ⟩ +≡
```
Vertex *barelist[max_n];
int bcount;   /* the current number of entries in barelist */
int curb[max_n];   /* value of bcount at the beginning of each level */
int curbb[max_n];   /* value of bcount in mid-level */
Vertex *bareback[max_n];   /* used for undoing barelist manipulations */
```

9. ⟨ Prepare the graph for backtracking 9 ⟩ ≡
```
d = infty;
bcount = ocount = 0;
for (v = g→vertices; v < g→vertices + g→n; v++) {
  v→type = bare;
  for (a = v→arcs, k = 0; a; a = a→next) k++;
  v→deg = k;
  if (k ≡ 2) barelist[bcount++] = v;
  if (k < d) d = k, curv[0] = v;
  v→llink = v→rlink = v→mate = v→comate = Λ;
}
head→rlink = head→llink = head;
head→name = "head";
if (d < 2) {
  printf("There␣are␣no␣Hamiltonian␣circuits,␣because␣%s␣has␣degree␣%d!\n", curv[0]→name, d);
  exit(0);
}
```
This code is used in section 1.

10. The arcs currently chosen appear in lists called *source* and *dest*. Some arcs are chosen when a bare vertex is being clothed; others are chosen at a level of backtracking when an outer vertex becomes inner.

⟨ Global variables 3 ⟩ +≡
```
Vertex *source[max_n], *dest[max_n];   /* the answers */
int ocount;   /* the current number of entries in source and dest */
int curo[max_n];   /* value of ocount at the beginning of each level */
```

11. Finally, a few other minor structures help us with backtracking or when we want to assess the progress of a potentially long calculation.

⟨ Global variables 3 ⟩ +≡
```
Vertex *curv[max_n];   /* outer vertex chosen for branching */
Arc *cura[max_n];   /* edge chosen for branching */
int curi[max_n];   /* index of the choice */
int maxi[max_n];   /* total number of choices */
int profile[max_n];   /* number of times we reached this level */
int l;   /* the current level of backtracking */
int maxl;   /* the largest l seen so far */
unsigned int total;   /* this many solutions so far */
```

12. Hamiltonian path problems often take a long time. The following subroutine can be called with an online debugger, to assess how far the work has progressed.

⟨ Subroutines 5 ⟩ +≡

```
void print_state()
{
  register int i, j, k;
  for (j = k = 0; k ≤ l; j++, k++) {
    while (j < curo[k]) {
      printf("␣␣␣␣␣␣%s--%s\n", source[j]→name, dest[j]→name);
      j++;
    }
    if (k) {
      if (j < g→n)
        printf("␣%3d:␣%s--%s␣(%d␣of␣%d)\n", k, source[j]→name, dest[j]→name, curi[k], maxi[k]);
    } else ⟨ Print the state line for the bottom level 39 ⟩;
  }
}
```

13. ⟨ Print the results 13 ⟩ ≡

```
printf("Altogether␣%u␣solutions.\n", total);
if (verbose) {
  for (k = 1; k ≤ maxl; k++) printf("%3d:␣%d\n", k, profile[k]);
}
```

This code is used in section 1.

14. 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 levels $l \ge 1$ is in place and understood.

⟨ Backtrack through all solutions 14 ⟩ ≡
  ⟨ Bootstrap the backtrack process 36 ⟩;
advance: ⟨ Clothe everything on the bare list 18 ⟩;   /* here I said *sanity_check*(Λ) when debugging */
  l++;
  if (*verbose*) {
    if ($l > maxl$) $maxl = l$;
    printf("Entering␣level␣%d:", l);
    profile[l]++;
  }
  if ($ocount \ge g\to n - 1$) ⟨ Check for solution and **goto** *backup* 32 ⟩;
  ⟨ Choose an outer vertex v of minimum degree d 15 ⟩;
  if (*verbose*) *printf*("␣choosing␣%s(%d)\n", $v\to name$, d);
  if ($d \equiv 0$) **goto** *backup*;
  $curv[l] = v$, $curi[l] = 1$, $maxi[l] = d$, $curb[l] = bcount$, $curo[l] = ocount$;
  $source[ocount] = v$;
  $w = v\to mate$;
  ⟨ Promote v from *outer* to *inner* 16 ⟩;
  $a = v\to arcs$;
try_move: **for** (; ; $a = a\to next$) {
    $u = a\to tip$;
    if ($u\to type \ne inner \land u \ne w$) **break**;
  }
  $cura[l] = a$;
  ⟨ Update data structures to account for choosing edge $cura[l]$ 17 ⟩;
  goto *advance*;
backup: l−−;
  if (*verbose*) *printf*("␣back␣to␣level␣%d:\n", l);
  ⟨ Unclothe everything clothed on level l 25 ⟩;
  if (l) {
    ⟨ Downdate data structures to deaccount for choosing edge $cura[l]$ 30 ⟩;
      /* here I said *sanity_check*(v) when debugging */
    if ($curi[l] < maxi[l]$) {
      $curi[l]$++;
      $w = v\to mate$; $a = cura[l]\to next$;
      goto *try_move*;
    }
    ⟨ Demote v from *inner* to *outer* 31 ⟩;
    if ($l > 1$) **goto** *backup*;
  }
  ⟨ Advance at bottom level 38 ⟩;

This code is used in section 1.

15. All the outer vertices are in the doubly linked list, and it is not empty.

⟨ Choose an outer vertex v of minimum degree d 15 ⟩ ≡

```
for (u = head→rlink, d = infty; u ≠ head; u = u→rlink) {
  if (verbose) printf("␣%s(%d)", u→name, u→deg);
  if (u→deg < d) d = u→deg, v = u;
}
```

This code is used in section 14.

16. At the beginning of a level, when we're about to choose a neighbor for the outer vertex v, we convert v to *inner* type because this conversion will be valid regardless of which edge we choose.

```
#define dancing_delete(u) u→llink→rlink = u→rlink, u→rlink→llink = u→llink
#define decrease_deg(u, w)
        if (u→type ≡ bare) {
          u→deg−−;
          if (u→deg ≡ 2) barelist[bcount++] = u;
        } else if (u ≠ w) u→deg−−    /* u is an outer neighbor of v with v→mate = w */
```

⟨ Promote v from *outer* to *inner* 16 ⟩ ≡

```
for (a = v→arcs; a; a = a→next) {
  u = a→tip;
  if (u→type > inner) decrease_deg(u, w);
}
v→type = inner;
dancing_delete(v);
curbb[l] = bcount;
```

This code is used in section 14.

17. At this point, v is a formerly outer vertex that we're joining to vertex u. Also, $w = v\to mate$.

If u is type *outer*, we're joining two segments into one, making u of type *inner*. But if u is bare, we're lengthening a segment, and u becomes *outer*.

```
#define make_outer(u)
        {
          u→rlink = head→rlink, head→rlink→llink = u;
          u→llink = head, head→rlink = u;
          u→type = outer;
        }
#define vprint() if (verbose) printf("␣%s--%s\n", source[ocount − 1]→name, dest[ocount − 1]→name)
```

⟨ Update data structures to account for choosing edge *cura*[*l*] 17 ⟩ ≡

```
  dest[ocount++] = u; vprint();
  if (u→type ≡ outer) {
    for (a = w→arcs; a; a = a→next)
      if (a→tip ≡ u→mate) {
        u→mate→deg−−, w→deg−−;
        break;
      }
    w→mate = u→mate, u→mate→mate = w;
    dancing_delete(u);
    u→type = inner;
    for (a = u→arcs; a; a = a→next) {
      w = a→tip;
      if (w→type > inner) decrease_deg(w, u→mate);
    }
  } else {    /* u→type ≡ bare */
    for (a = w→arcs; a; a = a→next)
      if (a→tip ≡ u) {
        u→deg−−, w→deg−−;
        break;
      }
    w→mate = u, u→mate = w;
    make_outer(u);
  }
```

This code is used in section 14.

18. The situation might have changed since a vertex entered the bare list, because its type and/or degree may have been altered.

Also, giving clothes to one bare vertex might have a ripple effect, causing other vertices to enter the bare list. The value of *bcount* 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.

⟨ Clothe everything on the bare list 18 ⟩ ≡

```
for (k = curb[l]; k < bcount; k++) {
  v = barelist[k];
  if (v→type ≠ bare) bareback[k] = v, barelist[k] = Λ;
  else {
    if (v→deg ≠ 2) {
      if (verbose) printf("(oops,␣low␣degree;␣backing␣up)\n");
      goto emergency_backup;      /* see below */
    }
    ⟨ Find the two neighbors, u and w, of vertex v 19 ⟩;
    if (u→mate ≡ w ∧ ocount ≠ g→n − 2) {
      if (verbose) printf("(oops,␣short␣cycle;␣backing␣up)\n");
      goto emergency_backup;
    }
    v→mate = u, v→comate = w;
    v→type = inner;
    source[ocount] = u, dest[ocount++] = v; vprint();
    source[ocount] = v, dest[ocount++] = w; vprint();
    if (u→type ≡ bare)
      if (w→type ≡ bare) ⟨ Promote BBB to OIO 20 ⟩
      else ⟨ Promote BBO to OII 21 ⟩
    else if (w→type ≡ bare) ⟨ Promote OBB to IIO 22 ⟩
    else ⟨ Promote OBO to III 23 ⟩;
  }
}
```

This code is used in section 14.

19. ⟨ Find the two neighbors, u and w, of vertex v 19 ⟩ ≡

```
for (a = v→arcs; ; a = a→next) {
  u = a→tip;
  if (u→type ≠ inner) break;
}
for (a = a→next; ; a = a→next) {
  w = a→tip;
  if (w→type ≠ inner) break;
}
```

This code is used in section 18.

20. The clothing process involves four similar subcases (which, I admit, are slightly boring). We will see, however, that all of these manipulations are easily undone; and that fact, to me, is interesting indeed, almost climactic.

⟨Promote BBB to OIO 20⟩ ≡

```
{
  u→deg−−, w→deg−−;
  make_outer(u);
  make_outer(w);
  u→mate = w, w→mate = u;
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ w) {
      u→deg−−, w→deg−−;
      break;
    }
}
```

This code is used in section 18.

21. ⟨Promote BBO to OII 21⟩ ≡

```
{
  u→deg−−;
  make_outer(u);
  u→mate = w→mate, w→mate→mate = u;
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ w→mate) {
      u→deg−−, w→mate→deg−−;
      break;
    }
  for (a = w→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner) decrease_deg(v, w→mate);
  }
  w→type = inner;
  dancing_delete(w);
}
```

This code is used in section 18.

22. ⟨Promote OBB to IIO 22⟩ ≡

```
{
  w→deg−−;
  make_outer(w);
  w→mate = u→mate, u→mate→mate = w;
  for (a = w→arcs; a; a = a→next)
    if (a→tip ≡ u→mate) {
      w→deg−−, u→mate→deg−−;
      break;
    }
  for (a = u→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner) decrease_deg(v, u→mate);
  }
  u→type = inner;
  dancing_delete(u);
}
```

This code is used in section 18.

23. ⟨Promote OBO to III 23⟩ ≡

```
{
  for (a = u→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner) decrease_deg(v, u→mate);
  }
  u→type = inner;
  dancing_delete(u);
  for (a = w→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner) decrease_deg(v, w→mate);
  }
  w→type = inner;
  dancing_delete(w);
  if (u→mate ≠ w) {
    u→mate→mate = w→mate, w→mate→mate = u→mate;
    for (a = u→mate→arcs; a; a = a→next)
      if (a→tip ≡ w→mate) {
        u→mate→deg−−, w→mate→deg−−;
        break;
      }
  }
}
```

This code is used in section 18.

24. Backtracking. The fascinating thing about dancing links is the almost magical way in which the linked data structures snap back into place when we run the updating algorithm backwards. We do need constant vigilance, though, because the validity of the algorithms hangs by a slender thread.

```
#define dancing_undelete(v)  v→llink→rlink = v→rlink→llink = v
#define make_bare(v)  dancing_delete(v), v→type = bare, v→mate = Λ
```

25. The *emergency_backup* label in this section provides an interesting example of a case where it is right and proper to **goto** a statement in the middle of one loop from the middle of another. [See the discussion in Examples 6c and 7a of my paper "Structured programming with **go to** statements, *Computing Surveys* **6** (December 1974), 261–301.] The program jumps to *emergency_backup* when it is running through the bare list and finds a situation that cannot be completed to a Hamiltonian circuit; it will then undo whatever actions it had completed so far in the clothing loop, because the unclothing loop operates in reverse order.

```
⟨Unclothe everything clothed on level l 25⟩ ≡
  for (k = bcount − 1; k ≥ curb[l]; k−−) {
    v = barelist[k];
    if (¬v) barelist[k] = bareback[k];
    else {
      u = v→mate, w = v→comate;
      v→type = bare, v→mate = Λ;
      v→comate = Λ;    /* this isn't necessary, but I'm feeling tidy today */
      if (u→type ≡ outer)
        if (w→type ≡ outer) ⟨Demote OIO to BBB 26⟩
        else ⟨Demote OII to BBO 27⟩
      else if (w→type ≡ outer) ⟨Demote IIO to OBB 28⟩
      else ⟨Demote III to OBO 29⟩;
    }
  emergency_backup: ;
  }
```

This code is used in section 14.

26.
```
⟨Demote OIO to BBB 26⟩ ≡
  {
    u→deg++, w→deg++;
    make_bare(u);
    make_bare(w);
    for (a = u→arcs; a; a = a→next)
      if (a→tip ≡ w) {
        u→deg++, w→deg++;
        break;
      }
  }
```

This code is used in section 25.

27. The first statement here, '$v\to deg$ −−', compensates for the spurious increases that will occur because v is a neighbor of w and $v\to type$ is no longer *inner*.

⟨ Demote OII to BBO 27 ⟩ ≡

```
{
  v→deg −−;
  w→mate→mate = w;
  dancing_undelete(w);
  w→type = outer;
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ w→mate) {
      u→deg ++, w→mate→deg ++;
      break;
    }
  for (a = w→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner ∧ v ≠ w→mate) v→deg ++;
  }
  u→deg ++;
  make_bare(u);
}
```

This code is used in section 25.

28. ⟨ Demote IIO to OBB 28 ⟩ ≡

```
{
  v→deg −−;
  u→mate→mate = u;
  dancing_undelete(u);
  u→type = outer;
  for (a = w→arcs; a; a = a→next)
    if (a→tip ≡ u→mate) {
      w→deg ++, u→mate→deg ++;
      break;
    }
  for (a = u→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner ∧ v ≠ u→mate) v→deg ++;
  }
  w→deg ++;
  make_bare(w);
}
```

This code is used in section 25.

29. ⟨ Demote III to OBO 29 ⟩ ≡

```
{
  v→deg −= 2;    /* compensate for two spurious increases below */
  if (u→mate ≠ w) {
    u→mate→mate = u, w→mate→mate = w;
    for (a = u→mate→arcs; a; a = a→next)
      if (a→tip ≡ w→mate) {
        u→mate→deg ++, w→mate→deg ++;
        break;
      }
  }
  dancing_undelete(w);
  w→type = outer;
  for (a = w→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner ∧ v ≠ w→mate) v→deg ++;
  }
  dancing_undelete(u);
  u→type = outer;
  for (a = u→arcs; a; a = a→next) {
    v = a→tip;
    if (v→type ≠ inner ∧ v ≠ u→mate) v→deg ++;
  }
}
```

This code is used in section 25.

30. A somewhat subtle point deserve special mention here: We want to reset *bcount* to *curbb*[*l*], not to *curb*[*l*], because entries that were put onto the *barelist* while *v* was becoming *inner* should remain there.

⟨ Downdate data structures to deaccount for choosing edge *cura*[*l*] 30 ⟩ ≡

```
v = curv[l];
ocount = curo[l];
u = dest[ocount];      /* cura[l]→tip */
if (u→type ≡ inner) {
  for (a = u→arcs; a; a = a→next) {
    w = a→tip;
    if (w→type ≠ inner ∧ w ≠ u→mate) w→deg++;
  }
  u→type = outer;
  dancing_undelete(u);
  w = v→mate;
  u→mate→mate = u, w→mate = v;
  for (a = w→arcs; a; a = a→next)
    if (a→tip ≡ u→mate) {
      u→mate→deg++, w→deg++;
      break;
    }
} else {      /* u→type ≡ outer */
  make_bare(u);
  w = v→mate;
  w→mate = v;
  for (a = w→arcs; a; a = a→next)
    if (a→tip ≡ u) {
      u→deg++, w→deg++;
      break;
    }
}
bcount = curbb[l];
```

This code is used in section 14.

31. ⟨ Demote *v* from *inner* to *outer* 31 ⟩ ≡

```
bcount = curb[l];
dancing_undelete(v);
v→type = outer;
for (a = v→arcs; a; a = a→next) {
  u = a→tip;
  if (u→type ≠ inner ∧ u ≠ w) u→deg++;
}
```

This code is used in section 14.

32. 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 circuit, although its last link usually still needs to be filled in.

⟨ Check for solution and **goto** *backup* 32 ⟩ ≡

```
{
  if (ocount < g→n) ⟨ If the two outer vertices aren't adjacent, goto backup 33 ⟩;
  total++;
  if (total % abs(modulus) ≡ 0 ∨ verbose) {
    curo[l] = ocount;
    source[ocount] = head→rlink, dest[ocount] = head→llink;
    curi[l] = maxi[l] = 1;
    if (modulus < 0) {
      printf("\n%d:\n", total); print_state();
    } else ⟨ Unscramble and print the current solution 34 ⟩;
  }
  goto backup;
}
```

This code is used in section 14.

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

⟨ If the two *outer* vertices aren't adjacent, **goto** *backup* 33 ⟩ ≡

```
{
  u = head→llink, v = head→rlink;
  for (a = u→arcs; a; a = a→next)
    if (a→tip ≡ v) goto yes;
  goto backup;
yes: ;
}
```

This code is used in section 32.

34. **#define** $index(v)$ $((v) - g\rightarrow vertices)$

⟨ Unscramble and print the current solution 34 ⟩ ≡

```
{
  register int i, j, k;
  for (k = 0; k < g→n; k++) v1[k] = −1;
  for (k = 0; k < g→n; k++) {
    i = index(source[k]);
    j = index(dest[k]);
    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;
  }
  if (verbose) printf("\n");
  printf("%d:", total);
  for (k = 0; k ≤ g→n; k++) printf("␣%s", (g→vertices + path[k])→name);
  printf("\n");
}
```

This code is used in section 32.

35. ⟨ Global variables 3 ⟩ +≡

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

36. 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 *barelist* will provide us with at least two *outer* 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 $maxi[0] = 0$. In the latter case, we take a vertex v of minimum degree d; we set $maxi[0] = d-1$, and try each neighbor of v in turn. (More precisely, after we've found all Hamiltonian cycles that contain an edge from v to some other vertex, u, we'll remove that edge physically from the graph, and repeat the process until v or some other vertex has only two neighbors left.)

```
⟨Bootstrap the backtrack process 36⟩ ≡
  l = 0;
  if (d > 2) {
     maxi[0] = d − 1;
     source[0] = v = curv[0];
     make_outer(v);
  force: cura[0] = a = v→arcs;
     v→arcs = a→next;
     curi[0]++;
     dest[0] = u = a→tip;
     ocount = 1; vprint();
     make_outer(u);
     v→deg−−;
     u→deg−−;
     ⟨Remove the arc from u to v 37⟩;
     v→mate = u, u→mate = v;
  }
```

This code is used in section 14.

37. ⟨Remove the arc from u to v 37⟩ ≡

```
  if (u→arcs→tip ≡ v) u→arcs = u→arcs→next;
  else {
     for (a = u→arcs; a→next→tip ≠ v; a = a→next) ;
     a→next = a→next→next;
  }
```

This code is used in section 36.

38. When the edge between u and v is removed, and u reverts to a *bare* vertex, it might now have degree 2. In such cases we don't need v as a seed vertex, so we revert to the simpler algorithm.

⟨ Advance at bottom level 38 ⟩ ≡

```
if (curi[0] < maxi[0]) {
  if (verbose) printf("␣back␣to␣level␣0:\n");
  l = 0;
  ocount = 0;
  u = dest[0];
  dancing_delete(u);
  u→type = bare;
  if (u→deg ≡ 2) barelist[0] = u, bcount = 1;
  else bcount = 0;      /* we never undo barelist conversions at level zero */
  v = source[0];
  if (v→deg ≡ 2) {
    v→type = bare;
    dancing_delete(v);
    barelist[bcount++] = v;
  }
  if (bcount ≡ 0) goto force;
  maxi[0] = curi[0] = curi[0] + 1;      /* cut to the chase */
  cura[0] = Λ;
  goto advance;
}
```

This code is used in section 14.

39. ⟨ Print the state line for the bottom level 39 ⟩ ≡

```
if (cura[0]) printf("␣%3d:␣%s--%s␣(%d␣of␣%d)\n", 0, source[0]→name, dest[0]→name, curi[0], maxi[0]);
else {
  j = −1;      /* this trick will make source[0] and dest[0] appear */
  if (maxi[0]) printf("␣%3d:␣(%d␣of␣%d)\n", 0, curi[0], maxi[0]);
}
```

This code is used in section 12.

40. Index.

⟨ Advance at bottom level 38 ⟩ Used in section 14.
⟨ Backtrack through all solutions 14 ⟩ Used in section 1.
⟨ Bootstrap the backtrack process 36 ⟩ Used in section 14.
⟨ Check for solution and **goto** *backup* 32 ⟩ Used in section 14.
⟨ Choose an outer vertex v of minimum degree d 15 ⟩ Used in section 14.
⟨ Clothe everything on the bare list 18 ⟩ Used in section 14.
⟨ Demote III to OBO 29 ⟩ Used in section 25.
⟨ Demote IIO to OBB 28 ⟩ Used in section 25.
⟨ Demote OII to BBO 27 ⟩ Used in section 25.
⟨ Demote OIO to BBB 26 ⟩ Used in section 25.
⟨ Demote v from *inner* to *outer* 31 ⟩ Used in section 14.
⟨ Downdate data structures to deaccount for choosing edge $cura[l]$ 30 ⟩ Used in section 14.
⟨ Find the two neighbors, u and w, of vertex v 19 ⟩ Used in section 18.
⟨ Global variables 3, 4, 8, 10, 11, 35 ⟩ Used in section 1.
⟨ If the two *outer* vertices aren't adjacent, **goto** *backup* 33 ⟩ Used in section 32.
⟨ Prepare the graph for backtracking 9 ⟩ Used in section 1.
⟨ Print the results 13 ⟩ Used in section 1.
⟨ Print the state line for the bottom level 39 ⟩ Used in section 12.
⟨ Process the command line, inputting the graph 2 ⟩ Used in section 1.
⟨ Promote BBB to OIO 20 ⟩ Used in section 18.
⟨ Promote BBO to OII 21 ⟩ Used in section 18.
⟨ Promote OBB to IIO 22 ⟩ Used in section 18.
⟨ Promote OBO to III 23 ⟩ Used in section 18.
⟨ Promote v from *outer* to *inner* 16 ⟩ Used in section 14.
⟨ Remove the arc from u to v 37 ⟩ Used in section 36.
⟨ Subroutines 5, 6, 7, 12 ⟩ Used in section 1.
⟨ Unclothe everything clothed on level l 25 ⟩ Used in section 14.
⟨ Unscramble and print the current solution 34 ⟩ Used in section 32.
⟨ Update data structures to account for choosing edge $cura[l]$ 17 ⟩ Used in section 14.

HAMDANCE