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

1.   Introduction.   This little program finds all the Hamiltonian cycles 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 cycle, with no three touching any vertex; we repeatedly choose further edges, preserving this condition while not completing any cycles that are too short.

   Note: This program assumes that the graph has no multiple edges. Otherwise it might get into a loop.

   (I wrote the original form of this program on 14 May 2001. Now, in 2025 as I prepare to finish Section 7.2.2.4 of TAOCP, I've polished it and instrumented it with mem counts, etc. I couldn't help noticing that, in hindsight, this program doesn't realy use the now-standard idea of "dancing links" when it manipulates *llink* and *rlink*; a more complex kind of choreography is involved. As in my other programs, one "mem" essentially means a memory access to a 64-bit word. The number of mems reported does not include the time needed to input the graph or to print 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 ∗/
#**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*;
     register int *i*, *j*, *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 cycle order.

#**define** max_n 20000   /∗ 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** $g = \Lambda$;
   if $(argc < 3 \vee sscanf(argv[2], \texttt{"\%d"}, \&modulus) \neq 1)$ $modulus = infty$;
   if $(\neg g \vee modulus \equiv 0)$ {
      $fprintf(stderr, \texttt{"Usage:}␣\texttt{\%s}␣\texttt{foo.gb}␣\texttt{[[-]modulus]}␣\texttt{[verbose]\\n"}, argv[0])$;
      $exit(-1)$;
   }
   if $(g\text{-}n > max\_n)$ {
      $fprintf(stderr, \texttt{"Sorry,}␣\texttt{I'm}␣\texttt{set}␣\texttt{up}␣\texttt{to}␣\texttt{handle}␣\texttt{at}␣\texttt{most}␣\texttt{\%d}␣\texttt{vertices!\\n"}, max\_n)$;
      $exit(-2)$;
   }
   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 ∗/
   unsigned long long $mems$;
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. ["Effectively removed" means that the edge is still in the graph, but it's not included in the current *deg* count.]

   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 cycle; 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 *outer* vertex of minimum degree, and make it inner in all possible ways. (A special routine is used to get started.)

   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=%ld", *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.

#**define** *sanity_checking*   0   /∗ set this to 1 if you suspect a bug ∗/

⟨ 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␣%ld!\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 $(v = g\text{→}vertices;\ v < g\text{→}vertices + g\text{→}n;\ v\text{++})$ {
    $o, v\text{→}type = bare$;
    for $(o, a = v\text{→}arcs, k = 0;\ a;\ o, a = a\text{→}next)\ k\text{++}$;
    $o, v\text{→}deg = k$;
    if $(k \equiv 2)\ o, barelist[bcount\text{++}] = v$;
    if $(k < d)\ o, d = k, curv[0] = v$;
    $oo, oo, v\text{→}llink = v\text{→}rlink = v\text{→}mate = v\text{→}comate = \Lambda$;
  }
  $oo, head\text{→}rlink = head\text{→}llink = head$;
  $head\text{→}name = \texttt{"head"}$;
  if $(d < 2)$ {
    $printf(\texttt{"There␣are␣no␣Hamiltonian␣cycles,␣because␣\%s␣has␣degree␣\%d!\\n"}, curv[0]\text{→}name, d)$;
    $exit(0)$;
  }
This code is used in section 1.

10.   The arcs currently chosen appear in lists called *source* and *dest*. (The kth chosen arc goes from *source*[k] to *dest*[k].) 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.

  The *source* and *dest* arrays are used only for printing a solution, not for making any decisions. Therefore I'm not charging any mems for storing into them. (Except for *dest*[0], which has special significance at root level.)

⟨ 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 long long *total*;   /∗ this many solutions so far ∗/
  unsigned long long *nodes*;

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* (**FILE** *∗stream*)
  {
    register int *i, j, k*;
    for (*j* = *k* = 0; *k* ≤ *l*; *j*++, *k*++) {
      while (*j* < *curo*[*k*]) {
        fprintf (*stream*, "␣␣␣␣␣␣%s−−%s\n", *source*[*j*]→*name*, *dest*[*j*]→*name*);
        j++;
      }
      if (*k*) {
        if (*j* < *g*→*n*) *fprintf* (*stream*, "␣%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 root level 39 ⟩;
    }
  }

13.   ⟨ Print the results 13 ⟩ ≡
  fprintf (*stderr*, "Altogether␣%llu␣solution%s,␣%llu␣nodes,␣%llu␣mems.\n", *total*,
    total ≡ 1 ? "" : "s", *nodes*, *mems*);
  if (*verbose*) {
    for (*k* = 1; *k* ≤ *maxl*; *k*++) *fprintf* (*stderr*, "%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 $l \geq 1$ is in place and understood.

⟨ Backtrack through all solutions 14 ⟩ ≡
  ⟨ Bootstrap the backtrack process 36 ⟩;
advance: ⟨ Clothe everything on the bare list 18 ⟩;
  if (*sanity_checking*) *sanity_check*(Λ);
  l++, *nodes*++;
  if (*verbose*) {
    if (*l* > *maxl*) *maxl* = *l*;
    fprintf(*stderr*, "Entering␣level␣%d:", *l*);
    profile[*l*]++;
  }
  if (*ocount* ≥ *g*→*n* − 1) ⟨ Check for solution and **goto** *backup* 32 ⟩;
  ⟨ Choose an outer vertex *v* of minimum degree *d* 15 ⟩;
  if (*verbose*) *fprintf*(*stderr*, "␣choosing␣%s(%d)\n", *v*→*name*, *d*);
  if (*d* ≡ 0) **goto** *backup*;
  mems += 5, *curv*[*l*] = *v*, *curi*[*l*] = 1, *maxi*[*l*] = *d*, *curb*[*l*] = *bcount*, *curo*[*l*] = *ocount*;
  source[*ocount*] = *v*;
  o, *w* = *v*→*mate*;
  ⟨ Promote *v* from *outer* to *inner* 16 ⟩;
  o, *a* = *v*→*arcs*;
try_move: **for** (; ; *o*, *a* = *a*→*next*) {
    o, *u* = *a*→*tip*;
    if ((*o*, *u*→*type* ≠ *inner*) ∧ *u* ≠ *w*) **break**;
  }
  o, *cura*[*l*] = *a*;
  ⟨ Update data structures to account for choosing edge *cura*[*l*] 17 ⟩;
  goto *advance*;
backup: *l*−−;
  if (*verbose*) *fprintf*(*stderr*, "␣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 ⟩;
    if (*sanity_checking*) *sanity_check*(*v*);
    if (*oo*, *curi*[*l*] < *maxi*[*l*]) {
      o, *curi*[*l*]++;
      oo, *w* = *v*→*mate*, *a* = *cura*[*l*]→*next*;
      goto *try_move*;
    }
    ⟨ Demote *v* from *inner* to *outer* 31 ⟩;
    if (*l* > 1) **goto** *backup*;
  }
  ⟨ Advance at root 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 $(o, u = head \rightarrow rlink, d = infty;\ u \neq head;\ o, u = u \rightarrow rlink)$ {
        if $(verbose)\ fprintf(stderr, "\_\%s(\%ld)", u \rightarrow name, u \rightarrow deg);$
        if $(o, u \rightarrow deg < d)\ d = u \rightarrow deg, v = u;$
    }

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** $dancing\_delete(u)$ $oo, oo, u \rightarrow llink \rightarrow rlink = u \rightarrow rlink, u \rightarrow rlink \rightarrow llink = u \rightarrow llink$
#**define** $decrease\_deg(u, w)$
        /∗ u is a neighbor of $w \rightarrow mate$, whose type is changing from *outer* to *inner* ∗/
      if $(o, u \rightarrow type \equiv bare)$ {
        $oo, u \rightarrow deg$ −−;
        if $(u \rightarrow deg \equiv 2)\ o, barelist[bcount {+}{+}] = u;$
      } **else if** $(u \neq w)\ oo, u \rightarrow deg$ −−   /∗ u itself is *outer* ∗/

⟨ Promote v from *outer* to *inner* 16 ⟩ ≡
    for $(o, a = v \rightarrow arcs;\ a;\ o, a = a \rightarrow next)$ {
        $o, u = a \rightarrow tip;$
        if $(o, u \rightarrow type \neq inner)\ decrease\_deg(u, w);$
    }
    $o, v \rightarrow type = inner;$
    $dancing\_delete(v);$
    $o, curbb[l] = bcount;$

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, $w = v\text{-}mate$.

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)
        {
            ooo, u→rlink = head→rlink, head→rlink→llink = u;
            oo, u→llink = head, head→rlink = u;
            o, u→type = outer;
        }
#define vprint()
        if (verbose) fprintf(stderr, "␣%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 (o, u→type ≡ outer) {
    for (oo, a = w→arcs; a; o, a = a→next)       /∗ extra mem to fetch u→mate outside the loop ∗/
      if (a→tip ≡ u→mate) {
        oo, u→mate→deg −−, oo, w→deg −−;
        break;
      }
    oo, w→mate = u→mate, u→mate→mate = w;
    dancing_delete(u);
    o, u→type = inner;
    for (o, a = u→arcs; a; o, a = a→next) {
      o, w = a→tip;
      if (o, w→type ≠ inner) decrease_deg(w, u→mate);
    }
  } else {      /∗ u→type ≡ bare ∗/
    for (o, a = w→arcs; a; o, a = a→next)
      if (o, a→tip ≡ u) {
        oo, u→deg −−, oo, w→deg −−;
        break;
      }
    oo, 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.

$\langle$ Clothe everything on the bare list 18 $\rangle \equiv$

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

This code is used in section 14.

19. $\langle$ Find the two neighbors, u and w, of vertex v 19 $\rangle \equiv$

```
for (o, a = v→arcs; ; o, a = a→next) {
    o, u = a→tip;
    if (o, u→type ≠ inner) break;
}
for (o, a = a→next; ; o, a = a→next) {
    o, w = a→tip;
    if (o, 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 ⟩ ≡
```
  {
    oo, u→deg −−, oo, w→deg −−;
    make_outer (u);
    make_outer (w);
    oo, u→mate = w, w→mate = u;
    for (o, a = u→arcs; a; o, a = a→next)
      if (o, a→tip ≡ w) {
        oo, u→deg −−, oo, w→deg −−;
        break;
      }
  }
```
This code is used in section 18.

21.   ⟨ Promote BBO to OII 21 ⟩ ≡
```
  {
    oo, u→deg −−;
    make_outer (u);
    ooo, u→mate = w→mate, w→mate→mate = u;
    for (o, a = u→arcs; a; o, a = a→next)
      if (o, a→tip ≡ w→mate) {
        oo, u→deg −−, oo, w→mate→deg −−;
        break;
      }
    for (o, a = w→arcs; a; o, a = a→next) {
      o, v = a→tip;
      if (o, v→type ≠ inner) decrease_deg (v, w→mate);
    }
    o, w→type = inner;
    dancing_delete (w);
  }
```
This code is used in section 18.

22.   (The same as BBO to OII, but with $u \leftrightarrow w$.)

⟨ Promote OBB to IIO 22 ⟩ ≡

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

This code is used in section 18.

23.   ⟨ Promote OBO to III 23 ⟩ ≡

```
{
    for (o, a = u→arcs; a; o, a = a→next) {
        o, v = a→tip;
        if (o, v→type ≠ inner) decrease_deg(v, u→mate);
    }
    o, u→type = inner;
    dancing_delete(u);
    for (o, a = w→arcs; a; o, a = a→next) {
        o, v = a→tip;
        if (o, v→type ≠ inner) decrease_deg(v, w→mate);
    }
    o, w→type = inner;
    dancing_delete(w);
    if (o, u→mate ≠ w) {       /∗ otherwise a complete cycle is in place ∗/
        ooo, u→mate→mate = w→mate, w→mate→mate = u→mate;
        for (o, a = u→mate→arcs; a; o, a = a→next)
            if (o, a→tip ≡ w→mate) {
                oo, u→mate→deg −−, oo, 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** $dancing\_undelete(v)$ $oo, oo, v{\rightarrow}llink{\rightarrow}rlink = v{\rightarrow}rlink{\rightarrow}llink = v$
#**define** $make\_bare\_from\_outer(v)$ $dancing\_delete(v), oo, v{\rightarrow}type = bare, v{\rightarrow}mate = \Lambda$

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 cycle; 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 $(o, k = bcount - 1; \ k \geq curb[l]; \ k{-}{-})$ {
    $o, v = barelist[k];$
    if $(\neg v)$ $oo, barelist[k] = bareback[k];$
    else {
      $oo, u = v{\rightarrow}mate, w = v{\rightarrow}comate;$
      $oo, v{\rightarrow}type = bare, v{\rightarrow}mate = \Lambda;$
      $v{\rightarrow}comate = \Lambda;$ /∗ this isn't necessary, but I'm feeling tidy today ∗/
      if $(o, u{\rightarrow}type \equiv outer)$
        if $(o, w{\rightarrow}type \equiv outer)$ ⟨ Demote OIO to BBB 26 ⟩
        else ⟨ Demote OII to BBO 27 ⟩
      else if $(o, w{\rightarrow}type \equiv outer)$ ⟨ 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 ⟩ ≡
  {
    for $(o, a = u{\rightarrow}arcs; \ a; \ o, a = a{\rightarrow}next)$
      if $(a{\rightarrow}tip \equiv w)$ {
        $oo, u{\rightarrow}deg{+}{+}, oo, w{\rightarrow}deg{+}{+};$
        break;
      }
    $make\_bare\_from\_outer(w);$
    $make\_bare\_from\_outer(u);$
    $oo, u{\rightarrow}deg{+}{+}, oo, w{\rightarrow}deg{+}{+};$
  }

This code is used in section 25.

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

⟨ Demote OII to BBO 27 ⟩ ≡
```
{
  oo, v→deg −−;
  o, w→mate→mate = w;
  dancing_undelete(w);
  o, w→type = outer;
  for (o, a = u→arcs; a; o, a = a→next)
    if (a→tip ≡ w→mate) {
      oo, u→deg ++, oo, w→mate→deg ++;
      break;
    }
  for (o, a = w→arcs; a; o, a = a→next) {
    o, v = a→tip;
    if (o, v→type ≠ inner ∧ v ≠ w→mate) oo, v→deg ++;
  }
  oo, u→deg ++;
  make_bare_from_outer(u);
}
```
This code is used in section 25.

28.   ⟨ Demote IIO to OBB 28 ⟩ ≡
```
{
  oo, v→deg −−;
  o, u→mate→mate = u;
  dancing_undelete(u);
  o, u→type = outer;
  for (o, a = w→arcs; a; o, a = a→next)
    if (a→tip ≡ u→mate) {
      oo, w→deg ++, oo, u→mate→deg ++;
      break;
    }
  for (o, a = u→arcs; a; o, a = a→next) {
    o, v = a→tip;
    if (o, v→type ≠ inner ∧ v ≠ u→mate) oo, v→deg ++;
  }
  oo, w→deg ++;
  make_bare_from_outer(w);
}
```
This code is used in section 25.

29. ⟨ Demote III to OBO 29 ⟩ ≡
  {
     $oo, v \rightarrow deg\ -=2;$ /∗ compensate for two spurious increases below ∗/
     if $(u \rightarrow mate \neq w)$ {
        $oo, u \rightarrow mate \rightarrow mate = u, w \rightarrow mate \rightarrow mate = w;$
        for $(o, a = u \rightarrow mate \rightarrow arcs;\ a;\ o, a = a \rightarrow next)$
           if $(a \rightarrow tip \equiv w \rightarrow mate)$ {
              $oo, u \rightarrow mate \rightarrow deg\ ++, oo, w \rightarrow mate \rightarrow deg\ ++;$
              break;
           }
     }
     $dancing\_undelete(w);$
     $o, w \rightarrow type = outer;$
     for $(o, a = w \rightarrow arcs;\ a;\ o, a = a \rightarrow next)$ {
        $v = a \rightarrow tip;$
        if $(o, v \rightarrow type \neq inner \wedge v \neq w \rightarrow mate)\ \ v \rightarrow deg\ ++;$
     }
     $dancing\_undelete(u);$
     $o, u \rightarrow type = outer;$
     for $(o, a = u \rightarrow arcs;\ a;\ o, a = a \rightarrow next)$ {
        $o, v = a \rightarrow tip;$
        if $(o, v \rightarrow type \neq inner \wedge v \neq u \rightarrow mate)\ \ v \rightarrow deg\ ++;$
     }
  }

This code is used in section 25.

30. A somewhat subtle point deserves 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 ⟩ ≡
```
  o, v = curv[l];
  o, ocount = curo[l];
  o, u = dest[ocount];      /* cura[l]→tip */
  if (o, u→type ≡ inner) {
    for (o, a = u→arcs; a; o, a = a→next) {
      o, w = a→tip;
      if (o, w→type ≠ inner ∧ w ≠ u→mate) w→deg ++;
    }
    o, u→type = outer;
    dancing_undelete(u);
    o, w = v→mate;
    oo, u→mate→mate = u, w→mate = v;
    for (o, a = w→arcs; a; o, a = a→next)
      if (o, a→tip ≡ u→mate) {
        oo, u→mate→deg ++, oo, w→deg ++;
        break;
      }
  } else {      /* u→type ≡ outer */
    make_bare_from_outer(u);
    o, w = v→mate;
    o, w→mate = v;
    for (o, a = w→arcs; a; o, a = a→next)
      if (o, a→tip ≡ u) {
        oo, u→deg ++, oo, w→deg ++;
        break;
      }
  }
  o, bcount = curbb[l];
```
This code is used in section 14.

31. ⟨ Demote *v* from *inner* to *outer* 31 ⟩ ≡
```
  o, bcount = curb[l];
  dancing_undelete(v);
  o, v→type = outer;
  for (o, a = v→arcs; a; o, a = a→next) {
    o, u = a→tip;
    if (o, 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 cycle, 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* mod *abs*(*modulus*) ≡ 0 ∨ *verbose*) {
       curo[*l*] = *ocount*;
       source[*ocount*] = *head→rlink*, *dest*[*ocount* ++] = *head→llink*;
       vprint( );
       curi[*l*] = *maxi*[*l*] = 1;
       if (*modulus* < 0) {
          printf("\n%llu:\n", *total*); *print_state*(*stdout*);
       } **else** ⟨ Unscramble and print the current solution 34 ⟩;
    }
    fflush(*stdout*);
    goto *backup*;
  }

This code is used in section 14.

33.   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* 33 ⟩ ≡
  {
    oo, *u* = *head→llink*, *v* = *head→rlink*;
    for (*o*, *a* = *u→arcs*; *a*; *o*, *a* = *a→next*)
       if (*o*, *a→tip* ≡ *v*) **break**;
    if (¬*a*) **goto** *backup*;
  }

This code is used in section 32.

34.   #**define** $index(v)$ $((v) - g\text{-}vertices)$

⟨ 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)  fprintf(stderr, "\n");
  for (k = 0; k ≤ g→n; k++)  printf("%s␣", (g→vertices + path[k])→name);
  printf("#%llu\n", total);
}
```

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 cycle, 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 $(d > 2)$ {
     $o, maxi[0] = d - 1$;
     $o, source[0] = v = curv[0]$;
     $make\_outer(v)$;
  $force$: $oo, cura[0] = a = v{\rightarrow}arcs$;
     $oo, v{\rightarrow}arcs = a{\rightarrow}next$;
     $oo, curi[0]{+}{+}$;
     $oo, dest[0] = u = a{\rightarrow}tip$;
     $ocount = 1$; $vprint(\,)$;
     $make\_outer(u)$;
     $oo, v{\rightarrow}deg{-}{-}$;
     $oo, u{\rightarrow}deg{-}{-}$;
     ⟨ Remove the arc from u to v 37 ⟩;
     $oo, v{\rightarrow}mate = u, u{\rightarrow}mate = v$;
  }

This code is used in section 14.

37.   ⟨ Remove the arc from u to v 37 ⟩ ≡
  if $(oo, u{\rightarrow}arcs{\rightarrow}tip \equiv v)$ $oo, u{\rightarrow}arcs = u{\rightarrow}arcs{\rightarrow}next$;
  else {
     for $(o, a = u{\rightarrow}arcs;\ oo, a{\rightarrow}next{\rightarrow}tip \neq v;\ a = a{\rightarrow}next)$;
     $oo, a{\rightarrow}next = a{\rightarrow}next{\rightarrow}next$;
  }

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.

$\langle$ Advance at root level $38 \rangle \equiv$

 if $(oo, curi[0] < maxi[0])$ {

  if $(verbose)$ $fprintf(stderr,$ "␣back␣to␣level␣0:\n"$);$

  $l = 0;$

  $ocount = 0;$

  $o, u = dest[0];$

  $dancing\_delete(u);$

  $o, u{\rightarrow}type = bare;$

  if $(o, u{\rightarrow}deg \equiv 2)$ $o, barelist[0] = u, bcount = 1;$

  else $bcount = 0;$ $/*$ we never undo $barelist$ conversions at level zero $*/$

  $o, v = source[0];$

  if $(o, v{\rightarrow}deg \equiv 2)$ {

   $o, v{\rightarrow}type = bare;$

   $dancing\_delete(v);$

   $o, barelist[bcount{+}{+}] = v;$

  }

  if $(bcount \equiv 0)$ **goto** $force;$

  $ooo, maxi[0] = curi[0] = curi[0] + 1;$ $/*$ cut to the chase $*/$

  $o, cura[0] = \Lambda;$

  goto $advance;$

 }

This code is used in section 14.

39. $\langle$ Print the state line for the root level $39 \rangle \equiv$

 if $(cura[0])$

  $fprintf(stream,$ "␣%3d:␣%s−−%s␣(%d␣of␣%d)\n"$, 0, source[0]{\rightarrow}name, dest[0]{\rightarrow}name, curi[0], maxi[0]);$

 else {

  $j = -1;$ $/*$ this trick will make $source[0]$ and $dest[0]$ appear $*/$

  if $(maxi[0])$ $fprintf(stream,$ "␣%3d:␣(%d␣of␣%d)\n"$, 0, curi[0], maxi[0]);$

 }

This code is used in section 12.

40.   Index.

HAMDANCE