1. Intro. This program tries to find Hamiltonian cycles using the method described by Euler in 1759. Given a path in a graph G, Euler used four tricks: (1) extend the path by adding an edge from the last vertex to a new vertex; (2) reverse the path's direction; (3) change a cyclic path of the form $x — y — \cdots — z — x$ to the cyclic path $y — \cdots — z — x — y$; (4) change a path of the form $w — \cdots — x — y — \cdots — z$, where $z — x$, to $w — \cdots — x — z — \cdots — y$.

When a transformation of type (1) is possible, thus increasing the length of the longest known path, we forget all previous paths and start over. Otherwise we keep exploring until either we either run out of memory or the four tricks don't lead to anything new.

We save memory by using (2) and (3) to put paths into a canonical form. Thus we remember only noncyclic paths whose first vertex is less than its last vertex, and we remember only cyclic paths whose first vertex is the smallest, and whose second vertex is less than its last. The remembered paths of G can be regarded as vertices of a giant graph H, in which we are doing a breadth-first search. We'll call them "supervertices."

(Euler was interested in knight's tours. If G is a knight graph, a noncyclic supervertex might have up to 14 neighbors in H; a cyclic supervertex has fewer than $12n$ neighbors.)

```
#define maxn 1024    /* at most this many vertices in G */
#define bits_per_vert 10    /* we must have (1 ≪ bits_per_vert) ≥ maxn */
#define verts_per_octa 6    /* we must have bits_per_vert * verts_per_octa ≤ 64 */
#define logmemsize 27
#define memsize (1 ≪ logmemsize)
#define memmask (memsize − 1)
#define loghashsize 12
#define hashsize (1 ≪ loghashsize)
#define hashmask (hashsize − 1)

#include "gb_graph.h"    /* use the Stanford GraphBase conventions */
#include "gb_save.h"    /* and its routine for inputting graphs */
#include "gb_flip.h"
  ⟨Preprocessor definitions⟩
  ⟨Global variables 4⟩
  Graph *g;    /* the given graph */
  int seed;    /* command-line parameter */
  ⟨Subroutines 11⟩
  int main(int argc, char *argv[])
  {
    register Vertex *u, *v;
    register Arc *a, *b;
    register int i, j, k, l, iu, iv, t;
    register unsigned long long acc;
    if ((1 ≪ bits_per_vert) < maxn ∨ bits_per_vert * verts_per_octa > 64) {
      fprintf(stderr, "Recompile␣me␣with␣correct␣parameters!\n");
      exit(−666);
    }
    ⟨Process the command line, inputting the graph 2⟩;
    ⟨Prepare the graph 3⟩;
    ⟨Carry out Euler's method 17⟩;
  done: fprintf(stderr, "Altogether␣%lld␣updates;", updates);
    fprintf(stderr, "␣found␣%lld␣cycle%s␣and␣%lld␣noncycle%s␣of␣size␣%d.\n", cycles,
        cycles ≡ 1 ? "" : "s", noncycles, noncycles ≡ 1 ? "" : "s", pathlen);
    fprintf(stderr, "Dictionary␣size␣%.1f␣(mean),␣%d␣(max).\n", dictave, dictmax);
  }
```

2. **#define** *vert*(*k*) (*g*→*vertices* + (*k*))

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

```
if (argc > 2) g = restore_graph(argv[1]); else g = Λ;
if (¬g) {
  fprintf(stderr, "Usage:␣%s␣foo.gb␣seed␣[v0]␣[v1]␣...\n", argv[0]);
  exit(−1);
}
n = g→n;
if (n > maxn) {
  fprintf(stderr, "Sorry,␣I␣allow␣only␣%d␣vertices,␣not␣%d!\n", maxn, n);
  exit(−2);
}
if (sscanf(argv[2], "%d", &seed) ≠ 1) {
  fprintf(stderr, "bad␣random␣seed␣‘%s’!\n", argv[2]);
  exit(−7);
}
gb_init_rand(seed);
for (k = 0; k < n ∧ argv[k + 3]; k++) {
  for (j = 0; j < n; j++)
    if (strcmp(argv[k + 3], vert(j)→name) ≡ 0) break;
  if (j ≡ n) {
    fprintf(stderr, "Vertex␣‘%s’␣isn’t␣in␣the␣graph!\n", argv[k + 3]);
    exit(−3);
  }
  path[k] = j;
}
if (¬k) k = 1, path[0] = gb_unif_rand(n);    /* if no path given, we use a random one-vertex path */
pathlen = k;    /* this is the number of vertices in the path, not its length */
```

This code is used in section 1.

3. The neighbors of each vertex are put into random order.

We also attach a random number to each edge of the graph, because the sum of those numbers will make a good hash key.

It's actually best to work with the full adjacency matrix, *adj*, and to store those edge weights in *adj*.

```
#define tmp u.A
#define ivert(v) ((v) - g->vertices)
```

⟨ Prepare the graph 3 ⟩ ≡

```
for (v = g->vertices; v < g->vertices + n; v++) {
  for (j = 0, a = v->arcs; a; j++, a = a->next) {
    vert(j)->tmp = a;
    if (a->tip > v) adj[ivert(v)][ivert(a->tip)] = adj[ivert(a->tip)][ivert(v)] = gb_next_rand() | (1 << 30);
  }
  for (i = 0; i < j; i++) {
    k = gb_unif_rand(j - i);
    if (i) b->next = vert(k)->tmp; else v->arcs = vert(k)->tmp;
    b = vert(k)->tmp;
    vert(k)->tmp = vert(j - i - 1)->tmp;
  }
  b->next = Λ;
}
for (pathhash = 0, j = 1; j < pathlen; j++) {
  if (¬adj[path[j - 1]][path[j]]) {
    fprintf(stderr, "Oops:␣`%s'␣isn't␣adjacent␣to␣`%s'!\n", vert(path[j - 1])->name,
        vert(path[j])->name);
    exit(-4);
  }
  pathhash += adj[path[j - 1]][path[j]];
}
```

This code is used in section 1.

4. ⟨ Global variables 4 ⟩ ≡

```
int n; /* the number of vertices in G */
int pathlen; /* the number of vertices in the current paths */
int adj[maxn][maxn]; /* the adjacency matrix of edge weights */
int path[maxn]; /* the current path */
int oldpath[maxn]; /* previous path used to generate new ones */
int where[maxn]; /* inverse permutation of oldpath */
int save[maxn]; /* temporary storage */
unsigned int pathhash; /* the full hash code for path */
unsigned int oldhash; /* the full hash code for oldpath */
```

See also section 5.

This code is used in section 1.

5. Data structures. We need to remember enough of what we've already done to avoid generating the same path twice. This means, when we are looking at all supervertices at distance d from the initial supervertex, we need to know all of the supervertices previously seen at distances $d-1$ and d, as we generate the ones at distance $d+1$. (We can, however, safely forget the supervertices at distance less than $d-1$.)

The remembered supervertices are stored as blocks of consecutive octabytes in *mem*, which is a big array of **unsigned long long** integers. Each supervertex block begins with one octabyte that contains its full hash code and a link to other supervertices (if any) that have the same truncated hash code. That initial octabyte is followed by $\lceil m/t \rceil$ others, where m is the number of vertices in the current paths and $t = \mathit{verts\_per\_octa}$ is the number of vertices that can be packed into an octabyte. The memory is treated as a cyclic queue, wrapping around from $\mathit{mem}[\mathit{memsize}-1]$ to $\mathit{mem}[0]$.

Link l therefore points to the block of b octas that begin at location $(l*b) \,\%\, \mathit{memsize}$, where $b = 1 + \lceil m/t \rceil$. This link is regarded as Λ if l is less than the first block for supervertices at distance $d-1$.

The first word of a block consists, more precisely, of a 32-bit link, followed by 32 bits of full hash code.

⟨ Global variables 4 ⟩ +≡
- **unsigned long long** *mem*[*memsize*]; /* the big memory array */
- **int** *prevstart*; /* first block for distance $d-1$ */
- **int** *curstart*; /* first block for distance d */
- **int** *curptr*; /* the block for the current supervertex */
- **int** *nextstart*; /* first block for distance $d+1$ */
- **int** *nextptr*; /* the block for the next supervertex */
- **unsigned int** *curlink*; /* link that corresponds to *curptr* */
- **unsigned int** *nextlink*; /* link that corresponds to *nextptr* */
- **int** *curd*; /* d */
- **int** *cutoff*; /* links less than this are treated as Λ */
- **int** *nextcutoff*; /* the *cutoff* to use when d increases */
- **int** *nextnextcutoff*; /* the *nextcutoff* to use when d increases */
- **int** *blocksize*; /* the size of each block, based on *pathlen* */
- **int** *cyclic*; /* is the current path cyclic? */
- **int** *hashhead*[*hashsize*]; /* heads of the hash lists */
- **long long** *updates*, *cycles*, *noncycles*;
- **int** *dictsize*; /* items currently in the dictionary */
- **int** *dictmax*; /* the maximum *dictsize* so far */
- **double** *dictave*; /* mean *dictsize* per update */

6. When we begin to process a supervertex, we unpack its path into the array *oldpath*.

\#**define** *mmod*(*x*) ((*x*) & *memmask*)
\#**define** *debugging* 0

⟨ Unpack the block at *curptr* 6 ⟩ ≡

```
oldhash = (unsigned int) mem[curptr];
for (j = 1, i = k = 0, acc = mem[mmod(curptr + 1)]; k < pathlen; k++) {
  oldpath[k] = acc & ((1 << bits_per_vert) - 1);
  where[oldpath[k]] = k;
  acc >>= bits_per_vert;
  if (++i == verts_per_octa) i = 0, acc = mem[mmod(curptr + (++j))];
}
cyclic = (adj[oldpath[0]][oldpath[pathlen - 1]] != 0);
if (debugging) ⟨ Do a sanity check on oldpath and oldhash 7 ⟩;
```

This code is used in section 14.

7. ⟨ Do a sanity check on *oldpath* and *oldhash* 7 ⟩ ≡

```
{
  register unsigned int h = 0;
  for (k = 1; k < pathlen; k++) h += adj[oldpath[k - 1]][oldpath[k]];
  if (cyclic) h += adj[oldpath[k - 1]][oldpath[0]];
  if (oldhash ≠ h) {
    fprintf(stderr, "Sanity␣check␣failure!\n");
    exit(−6666);
  }
}
```

This code is used in section 6.

8. When we've created a path that's possibly new, we pack it into the block *nextptr*.

⟨ Pack *path* into the block at *nextptr* 8 ⟩ ≡

```
for (j = 1, i = k = 0, acc = 0; k < pathlen; k++) {
  acc += (unsigned long long) path[k] ≪ (i ∗ bits_per_vert);
  if (++i ≡ verts_per_octa) {
    if (mmod(nextptr + j) ≡ prevstart) {
    memoverflow: fprintf(stderr, "Overflow␣(memsize=%d,␣dictsize=%d)!\n", memsize, dictsize);
      exit(−9);
    }
    mem[mmod(nextptr + j)] = acc, acc = 0, i = 0, j++;
  }
}
if (i) {
  if (mmod(nextptr + j) ≡ prevstart) goto memoverflow;
  mem[mmod(nextptr + j)] = acc;
}
```

This code is used in section 11.

9. A path isn't packed until it has been put into canonical form. (As mentioned earlier, a noncyclic path is equivalent to its reverse; a cyclic path is equivalent to all of its cyclic shifts and to all of its reverse's cyclic shifts.)

⟨ Canonize the *path* 9 ⟩ ≡

```
if (adj[path[0]][path[pathlen − 1]]) {
  cyclic = 1;
  pathhash += adj[path[0]][path[pathlen − 1]];
  for (j = 0, k = 1; k < pathlen; k++)
    if (path[k] < path[j]) j = k;
  if (j) ⟨ Shift the path cyclically left j 10 ⟩;
  if (path[1] > path[pathlen − 1])
    for (i = 1, j = pathlen − 1; i < j; i++, j−−) t = path[i], path[i] = path[j], path[j] = t;
} else {
  cyclic = 0;
  if (path[0] > path[pathlen − 1])
    for (i = 0, j = pathlen − 1; i < j; i++, j−−) t = path[i], path[i] = path[j], path[j] = t;
}
```

This code is used in section 11.

10. I know that there are tricky ways to shift a path cyclically in place. But I'm not short of memory space; and I'm short of personal time. So I use an auxiliary array.

⟨ Shift the path cyclically left j 10 ⟩ ≡

```
for (i = 0; i < j; i++) save[i] = path[i];
for ( ; i < pathlen; i++) path[i − j] = path[i];
for ( ; i − j < pathlen; i++) path[i − j] = save[i − pathlen];
```

This code is used in section 9.

11. The basic operation of the breadth-first search that we'll be doing consists of generating a path that's a neighbor of the current supervertex, and adding it to the collection of known supervertices if it hasn't been seen before. The *update* subroutine handles the latter task.

⟨ Subroutines 11 ⟩ ≡

```
void upd(void)
{
  register int h, i, j, k, l, ll, nextl, t;
  register unsigned long long acc;
  updates++;
  ⟨ Canonize the path 9 ⟩;
  ⟨ Pack path into the block at nextptr 8 ⟩;
  h = pathhash & hashmask;
  for (l = hashhead[h]; l ≥ cutoff; l = nextl) {
    ll = (blocksize ∗ l) & memmask;
    nextl = mem[ll] ≫ 32;
    if ((mem[ll] ⊕ pathhash) & #ffffffff) continue;    /∗ no match at ll ∗/
    for (j = 1; j < blocksize; j++)
      if (mem[(ll + j) & memmask] ≠ mem[(nextptr + j) & memmask]) break;
    if (j < blocksize) continue;
    break;    /∗ match found ∗/
  }
  if (l < cutoff) {    /∗ this supervertex is new ∗/
    if (cyclic) cycles++; else noncycles++;
    ⟨ Print path 12 ⟩;
    mem[nextptr] = ((unsigned long long) hashhead[h] ≪ 32) + pathhash;
    hashhead[h] = nextlink;
    if (nextlink ≡ #ffffffff) {
      fprintf(stderr, "Link␣overflow!\n");
      exit(−667);
    }
    nextlink++, nextptr = mmod(nextptr + blocksize), dictsize++;
    if (nextptr ≡ prevstart) goto memoverflow;
  }
  ⟨ Update the stats 13 ⟩;
}
```

This code is used in section 1.

12. ⟨ Print path 12 ⟩ ≡

```
for (k = 0; k < pathlen; k++) printf("%s%s", k ∨ ¬cyclic ? "␣" : "", vert(path[k])→name);
printf("␣#%u>%u\n", nextlink, curlink);
```

This code is used in section 11.

13. ⟨ Update the stats 13 ⟩ ≡

if ($dictsize > dictmax$) $dictmax = dictsize$;
$dictave$ += ((**double**) $dictsize - dictave$)/(**double**) $updates$;

This code is used in section 11.

14. Breadth-first search. OK, let's specify how a supervertex at distance d is processed.

⟨ Explore the neighbors of supervertex *curptr* 14 ⟩ ≡

```
{
  ⟨ Unpack the block at curptr 6 ⟩;
  if (¬cyclic) ⟨ Explore the neighbors of the noncyclic oldpath 15 ⟩
  else ⟨ Explore the neighbors of the cyclic oldpath 16 ⟩;
}
```

This code is used in section 17.

15. ⟨ Explore the neighbors of the noncyclic *oldpath* 15 ⟩ ≡

```
{
  iv = oldpath[pathlen − 1], v = vert(iv);
  for (a = v→arcs; a; a = a→next) {
    u = a→tip, iu = ivert(u);
    k = where[iu];    /* if k ≥ 0, we have iu = oldpath[k] */
    if (k < 0) {
      for (j = 0; j < pathlen; j++) path[j] = oldpath[j];
      path[pathlen] = iu;
      pathhash = oldhash + adj[iu][iv];
      goto breakthru;
    }
    if (k ≡ pathlen − 2) continue;    /* we already knew that u —— v */
    for (j = 0; j ≤ k; j++) path[j] = oldpath[j];
    for (i = pathlen − 1; i > k; i−−, j++) path[j] = oldpath[i];
    pathhash = oldhash + adj[iu][iv] − adj[iu][oldpath[k + 1]];
    update();
  }
  iv = oldpath[0], v = vert(iv);
  for (a = v→arcs; a; a = a→next) {
    u = a→tip, iu = ivert(u);
    k = where[iu];
    if (k < 0) {
      for (j = 0; j < pathlen; j++) path[j + 1] = oldpath[j];
      path[0] = iu;
      pathhash = oldhash + adj[iu][iv];
      goto breakthru;
    }
    if (k ≡ 1) continue;    /* we already knew that u —— v */
    for (i = 0; i < k; i++) path[i] = oldpath[k − 1 − i];
    for (j = k; j < pathlen; j++) path[j] = oldpath[j];
    pathhash = oldhash + adj[iu][iv] − adj[iu][oldpath[k − 1]];
    update();
  }
}
```

This code is used in section 14.

16. ⟨ Explore the neighbors of the cyclic *oldpath* 16 ⟩ ≡

```
{
  for (j = 0; j < pathlen; j++) {
    iv = oldpath[j], v = vert(iv);
    for (a = v→arcs; a; a = a→next) {
      u = a→tip, iu = ivert(u);
      k = where[iu];
      if (k < 0) {
        for (i = j; i < pathlen; i++) path[i + 1 − j] = oldpath[i];
        for (i = 0; i < j; i++) path[pathlen − j + i + 1] = oldpath[i];
        path[0] = iu;
        pathhash = oldhash + adj[iu][iv] − adj[j ? oldpath[j − 1] : oldpath[pathlen − 1]][oldpath[j]];
        goto breakthru;
      }
      if (k ≡ j − 1 ∨ k ≡ j + 1 ∨ k ≡ j − 1 + pathlen ∨ k ≡ j + 1 − pathlen) continue;
      for (t = 0, i = k − 1; ; i−−) {
        if (i < 0) i = pathlen − 1;
        path[t++] = oldpath[i];
        if (i ≡ j) break;
      }
      for (i = k; t < pathlen; i++) {
        if (i ≥ pathlen) i = 0;
        path[t++] = oldpath[i];
      }
      pathhash = oldhash + adj[iu][iv] − adj[iu][k ? oldpath[k − 1] : oldpath[pathlen − 1]]
       − adj[oldpath[j]][j ? oldpath[j − 1] : oldpath[pathlen − 1]];
      update();
      for (t = 0, i = j + 1; ; i++) {
        if (i ≥ pathlen) i = 0;
        path[t++] = oldpath[i];
        if (i ≡ k) break;
      }
      for (i = j; t < pathlen; i−−) {
        if (i < 0) i = pathlen − 1;
        path[t++] = oldpath[i];
      }
      pathhash = oldhash + adj[iu][iv] − adj[iu][k < pathlen − 1 ? oldpath[k + 1] : oldpath[0]]
       − adj[oldpath[j]][j < pathlen − 1 ? oldpath[j + 1] : oldpath[0]];
      update();
    }
  }
}
```

This code is used in section 14.

17. Putting it all together. We've implemented the basic functionality. The only thing left is to connect up the pieces. (I suppose Dijkstra would have done this first; perhaps I should have done so too.)

Subtle point: I want the first link to be 1, not 0. So the first block of *mem* to be filled starts at *blocksize*.

When a cycle is found, we know that we can always make a breakthru unless every vertex of that cycle has neighbors only in the cycle. We assume that the given graph is connected. Therefore we don't need to put a cycle in the dictionary unless $pathlen = n$.

```
#define update()  upd(); if (cyclic & pathlen < n) goto shortcut
⟨Carry out Euler's method 17⟩ ≡
  goto firstpath;
shortcut: for (j = 0; j < pathlen; j++) oldpath[j] = path[j], where[oldpath[j]] = j;
  for (j = 0; j < pathlen; j++) {
    iv = oldpath[j], v = vert(iv);
    for (a = v->arcs; a; a = a->next) {
      u = a->tip, iu = ivert(u);
      if (where[iu] < 0) break;
    }
    if (where[iu] < 0) break;
  }
  if (where[iu] ≥ 0) {
    fprintf(stderr, "*␣The␣graph␣isn't␣connected!\n");
    exit(0);    /* we've printed a Hamiltonian cycle of a connected component */
  }
  for (i = j; i < pathlen; i++) path[i + 1 - j] = oldpath[i];
  for (i = 0; i < j; i++) path[pathlen - j + i + 1] = oldpath[i];
  path[0] = iu;
  pathhash = pathhash + adj[iu][iv] - adj[j ? oldpath[j - 1] : oldpath[pathlen - 1]][oldpath[j]];
  printf("*␣");    /* a shortcut is a special kind of breakthru */
breakthru: printf("Breakthru␣after␣%lld␣cycles,␣%lld␣noncycles!\n", cycles, noncycles);
  pathlen++;
firstpath: blocksize = 1 + (int)((pathlen + verts_per_octa - 1)/verts_per_octa);
  for (k = 0; k < n; k++) where[k] = -1;
  for (k = 0; k < hashsize; k++) hashhead[k] = 0;
  cycles = noncycles = curlink = dictsize = 0;
  prevstart = curstart = nextptr = blocksize;
  cutoff = nextcutoff = nextlink = 1;
  printf("Paths␣and␣cycles␣of␣length␣%d:\n", pathlen);
  update();
  nextstart = nextptr, nextnextcutoff = nextlink, curlink = 1;
  for (curd = 0; ; curd++) {
    fprintf(stderr, "␣len␣%d␣after␣distance␣%d:␣%lld␣cycle%s,␣%lld␣noncycle%s\n", pathlen, curd,
        cycles, cycles ≡ 1 ? "" : "s", noncycles, noncycles ≡ 1 ? "" : "s");
    if (curstart ≡ nextstart) break;
    for (curptr = curstart; curptr ≠ nextstart; curptr = mmod(curptr + blocksize), curlink++)
      ⟨Explore the neighbors of supervertex curptr 14⟩;
    prevstart = curstart, curstart = nextstart, nextstart = nextptr;
    dictsize -= nextcutoff - cutoff;
    cutoff = nextcutoff, nextcutoff = nextnextcutoff, nextnextcutoff = nextlink;
  }
```

This code is used in section 1.

18. Index.

HAM-EULER