

HAM

1. Hamiltonian circuits. This program finds all Hamiltonian circuits of an undirected graph, using conventions of the Stanford GraphBase.

If the user says, for example, `'ham foo.gb'`, the standard output will list every Hamiltonian circuit of the graph `foo`, which should be represented in file `foo.gb` using the Stanford GraphBase's portable ASCII format. The total number of solutions is reported at the end of the output.

An optional second parameter specifies an interval between outputs, so that the list contains only a sample of the solutions. For example, `'ham foo.gb 1000'` will list only one of every 1000 Hamiltonian circuits. If the second parameter is zero, only the total number of circuits will be output.

```
#include <stdio.h>      /* standard C input/output functions */
#include "gb_graph.h"    /* the GraphBase data structures */
#include "gb_save.h"     /* the restore_graph routine */

2. We use a utility field deg to record vertex degrees.

#define deg u.I          /* the current number of arcs to and from this vertex */
int main(int argc, char *argv[])
{
    Graph *g;            /* the user's graph */
    register Vertex *t, *u, *v; /* key vertices */
    Vertex *x, *y, *z;    /* vertices used less often */
    register Arc *a, *aa; /* arcs used often */
    Arc *b, *bb;          /* arcs used less often */
    int count = 0;        /* solutions found so far */
    int interval = 1;     /* the reporting interval */

    < Scan the command line arguments and input g 3 >;
    < Prepare g for backtracking, and find a vertex x of minimum degree 4 >;
    < Abort the run if g is malformed or x-deg < 2 5 >;
    for (b = x->arcs; b->next; b = b->next)
        for (bb = b->next; bb; bb = bb->next) {
            v = b->tip;
            z = bb->tip;
            < Find all simple paths of length g-n - 2 from v to z, avoiding x 7 >;
        }
    printf("Altogether %d solutions.\n", count);
    return 0; /* normal exit */
}

3. < Scan the command line arguments and input g 3 > ≡
if (argc > 2 ∧ sscanf(argv[2], "%d", &interval) ≡ 1) {
    argc--;
    if (interval < 0) interval = -interval;
    else if (interval ≡ 0) interval = -10000;
    /* suppress output when 0 is specified */
}
if (argc ≠ 2) {
```

```

    printf("Usage: %s foo.gb [interval]\n", argv[0]);
    return -1;
}
g = restore_graph(argv[1]);

```

This code is used in section 2.

4. Vertices that have already appeared in the path are said to be “taken,” and their *taken* field is nonzero. Initially we make all those fields zero.

```

#define taken v.I /* does this vertex appear in the current path? */
#define not_taken(v) ((v)->taken == 0)
⟨Prepare g for backtracking, and find a vertex x of minimum degree 4⟩ ≡
    if (g) { int dmin = g-n;
        for (v = g-vertices; v < g-vertices + g-n; v++) {
            register int d = 0; /* the degree of v */
            v->taken = 0;
            for (a = v->arcs; a; a = a->next) d++;
            v->deg = d;
            if (d < dmin) dmin = d, x = v;
        }
    }

```

This code is used in section 2.

5. A vertex with fewer than two neighbors cannot be part of a Hamiltonian circuit, so we give such cases short shrift.

```

⟨Abort the run if g is malformed or x->deg < 2 5⟩ ≡
    if (!g) {
        printf("Graph %s is malformed (error code %ld)!\n", argv[1], panic_code);
        return -2;
    }
    if (x->deg < 2) {
        printf("No solutions (vertex %s has degree %ld).\n", x->name, x->deg);
        return -3;
    }

```

This code is used in section 2.

Arc = struct arc_struct,	<i>name</i> : char *, GB_GRAPH §9.	GB_GRAPH §10.
GB_GRAPH §10.	<i>next</i> : struct area_pointers	<i>u</i> : util, GB_GRAPH §9.
<i>arcs</i> : static Arc *,	*, GB_GRAPH §12.	<i>v</i> : register Vertex *, ham §2.
GB_SAVE §8.	<i>panic_code</i> : long,	<i>v</i> : util, GB_GRAPH §9.
<i>deg</i> = u.I, ham §2.	GB_GRAPH §5.	Vertex = struct
Graph = struct	<i>printf</i> : int (), <stdio.h>.	vertex_struct ,
graph_struct ,	<i>restore_graph</i> : Graph *(),	GB_GRAPH §9.
GB_GRAPH §20.	GB_SAVE §4.	<i>vertices</i> : Vertex *,
<i>I</i> : long, GB_GRAPH §8.	<i>sscanf</i> : int (), <stdio.h>.	GB_GRAPH §20.
<i>n</i> : long, GB_SAVE §31.	<i>tip</i> : struct vertex_struct *,	

6. The algorithm. Unproductive branches of the search tree are cut off by using a simple rule: If one of the vertices we could move to next is adjacent to only one other unused vertex, we must move to it now.

The moves will be recorded in the vertex array of g . More precisely, the k th vertex of the path will be $t\text{-vert}$ when t is the k th vertex of the graph. If the move was not forced, $t\text{-ark}$ will point to the **Arc** record representing the arc from $t\text{-vert}$ to $(t + 1)\text{-vert}$; otherwise $t\text{-ark}$ will be Λ .

```
#define vert w.V      /* vertex on current path */
#define ark x.A        /* arc to its current successor */
```

7. This program is a typical application of the backtrack method; in other words, it essentially does a depth-first search in the tree of all solutions. The author, being a member of the Old School, is most comfortable writing such programs with labels and **goto** statements, rather than with **while** loops. Perhaps some day he will learn his lesson; but backtrack programs do need to be streamlined for speed.

A complication arises because we may discover that a move is unproductive before we have completely updated the data structures recording that move.

```
<Find all simple paths of length  $g\text{-}n - 2$  from  $v$  to  $z$ , avoiding  $x$  7>  $\equiv$ 
{ Vertex  $t\text{max}$ ;      /* the deepest level */
   $t = g\text{-vertices}$ ;  $t\text{max} = t + g\text{-}n - 1$ ;    /*  $t$  represents the current level */
   $x\text{-taken} = 1$ ;  $t\text{-vert} = x$ ;  $t\text{-ark} = \Lambda$ ;
  advance:
    <Increase  $t$ , updating the data structures to show that vertex  $v$  is now taken, and set  $y$ 
      to a forced move, if any; but goto backtrack if no further moves are possible 8>;
    if ( $y$ ) { /* move is forced */
       $t\text{-ark} = \Lambda$ ;  $v = y$ ; goto advance;
    }
     $a = v\text{-arcs}$ ;
  search: <Look at arc  $a$  and its successors, advancing if a valid move is found 10>;
  restore:  $aa = \Lambda$ ;
  restore_to_aa: <Downdate the data structures to the state they were in when level  $t$  was
    entered, stopping at arc  $aa$  9>;
  backtrack: <Decrease  $t$ , if possible, and search for another possibility 11>;
}
```

This code is used in section 2.

8. When a vertex becomes taken, we pretend that it has been removed from the graph.

```
<Increase  $t$ , updating the data structures to show that vertex  $v$  is now taken, and set  $y$  to a
  forced move, if any; but goto backtrack if no further moves are possible 8>  $\equiv$ 
 $t++$ ;
 $t\text{-vert} = v$ ;
 $v\text{-taken} = 1$ ;
if ( $v \equiv z$ ) {
  if ( $t \equiv t\text{max}$ ) <Record a solution 12>;
  goto backtrack;
}
```

```

for ( $aa = v \rightarrow arcs, y = \Lambda; aa; aa = aa \rightarrow next$ ) { register int  $d$ ;
     $u = aa \rightarrow tip$ ;
     $d = u \rightarrow deg - 1$ ;
    if ( $d \equiv 1 \wedge not\_taken(u)$ ) {      /* we must move next to  $u$  */
        if ( $y$ ) goto restore_to_aa;    /* two forced moves can't both be made */
         $y = u$ ;
    }
     $u \rightarrow deg = d$ ;      /*  $u$  can no longer move to  $v$  */
}

```

This code is used in section 7.

9. We didn't change the graph drastically at level t ; all we did was decrease the degrees of vertices reachable from $t \rightarrow vert$. Therefore we can easily undo previous changes when we are backing up.

\langle Downdate the data structures to the state they were in when level t was entered, stopping at arc aa $\rangle \equiv$

```

for ( $a = t \rightarrow vert \rightarrow arcs; a \neq aa; a = a \rightarrow next$ )  $a \rightarrow tip \rightarrow deg ++$ ;

```

This code is used in section 7.

10. \langle Look at arc a and its successors, advancing if a valid move is found $\rangle \equiv$

```

while ( $a$ ) {
     $v = a \rightarrow tip$ ;
    if ( $not\_taken(v)$ ) {
         $t \rightarrow ark = a$ ;
        goto advance;      /* move to  $v$  */
    }
     $a = a \rightarrow next$ ;
}

```

This code is used in section 7.

11. \langle Decrease t , if possible, and search for another possibility $\rangle \equiv$

```

 $t \rightarrow vert \rightarrow taken = 0$ ;
 $t --$ ;
if ( $t \rightarrow ark$ ) {
     $a = t \rightarrow ark \rightarrow next$ ;
    goto search;
}
if ( $t \neq g \rightarrow vertices$ ) goto restore;      /* the move was forced, so we bypass search */

```

This code is used in section 7.

a : register Arc *, §2. A : struct <i>arc_struct</i> *, GB_GRAPH §8. aa : register Arc *, §2. Arc : struct <i>arc_struct</i> , GB_GRAPH §10. $arcs$: static Arc *, GB_SAVE §8. $ark = x.A$, ham §6. $deg = u.I$, ham §2. g : Graph *, §2. n : long , GB_SAVE §31.	$next$: struct <i>area_pointers</i> *, GB_GRAPH §12. $not_taken = \text{macro } (), §4$. t : register Vertex *, §2. $taken = v.I$, ham §4. tip : struct <i>vertex_struct</i> *, GB_GRAPH §10. u : register Vertex *, §2. v : register Vertex *, §2. V : struct <i>vertex_struct</i> *, GB_GRAPH §8.	$vert = w.V$, ham §6. $Vertex = \text{struct}$ <i>vertex_struct</i> , GB_GRAPH §9. $vertices$: Vertex *, GB_GRAPH §20. w : util , GB_GRAPH §9. x : Vertex *, §2. x : util , GB_GRAPH §9. y : Vertex *, §2. z : Vertex *, §2.
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

12. We print a solution by simply listing the vertex names in the current path.

$\langle \text{Record a solution } 12 \rangle \equiv$

```

{
    count++;
    if ( $count \% interval \equiv 0 \wedge interval > 0$ ) {
        printf("%d:\n", count);
        for ( $u = g\text{-vertices}; u \leq tmax; u++$ ) printf("%s\n",  $u\text{-vert-name}$ );
        printf("\n");
    }
}

```

This code is used in section 8.

13. Index.

count: **int**, §2.
g: **Graph** *, §2.
interval: **int**, §2.
name: **char** *, GB_GRAPH §9.

printf: **int** (), <stdio.h>.
tmax: **Vertex** *, §7.
w: **register Vertex** *, §2.

vert = *w.V*, ham §6.
vertices: **Vertex** *,
GB_GRAPH §20.