

(Downloaded from <https://cs.stanford.edu/~knuth/programs.html> and typeset on May 28, 2023)

**1. Hamiltonian cycles.** This program finds all Hamiltonian cycles of an undirected graph. [It's a slight revision of the program published in my paper "Mini-indexes for literate programs," *Software—Concepts and Tools* **15** (1994), 2–11.] The input graph should be in Stanford GraphBase format, and should be named on the command line as, for example, `foo.gb`. An optional second command-line parameter is a modulus  $m$ , which causes every  $m$ th solution to be printed.

We use a utility field to record the vertex degrees.

```
#define deg u.I
#include "gb_graph.h" /* the GraphBase data structures */
#include "gb_save.h" /* restore_graph */
main(int argc, char *argv[])
{
    Graph *g;
    Vertex *x, *y, *z, *tmax;
    register Vertex *t, *u, *v;
    register Arc *a, *aa;
    register int d;
    Arc *b, *bb;
    int count = 0;
    int dmin, modulus;

    ⟨Process the command line, inputting the graph 2⟩;
    ⟨Prepare g for backtracking, and find a vertex x of minimum degree 3⟩;
    for (v = g-vertices; v < g-vertices + g-n; v++) printf("%d", v-deg);
    printf("\n"); /* TEMPORARY CHECK */
    if (x-deg < 2) {
        printf("The minimum degree is %d (vertex %s)!\n", x-deg, x-name);
        return -1;
    }
    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 4⟩;
        }
    printf("Altogether %d solutions.\n", count);
    for (v = g-vertices; v < g-vertices + g-n; v++) printf("%d", v-deg);
    printf("\n"); /* TEMPORARY CHECK, SHOULD AGREE WITH FORMER VALUES */
}
```

**2.** ⟨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 = 1000000000;
if (¬g ∨ modulus ≤ 0) {
    fprintf(stderr, "Usage: %s foo.gb [modulus]\n", argv[0]);
    exit(-1);
}
```

This code is used in section 1.

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

**#define** *taken* *v.I*

⟨ Prepare *g* for backtracking, and find a vertex *x* of minimum degree [3](#) ⟩ ≡

```

    dmin = g→n;
    for (v = g→vertices; v < g→vertices + g→n; v++) {
        v→taken = 0;
        d = 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 [1](#).

**4. The data structures.** I use one simple rule to cut off unproductive branches of the search tree: 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 edge from  $t\text{-vert}$  to  $(t+1)\text{-vert}$ ; otherwise  $t\text{-ark}$  will be  $\Lambda$ .

This program is a typical backtrack program. I am more comfortable doing it with labels and goto statements than with while loops, but some day I may learn my lesson.

```
#define vert w.V
```

```
#define ark x.A
```

```
< Find all simple paths of length  $g\text{-}n - 2$  from  $v$  to  $z$ , avoiding  $x$  4 >  $\equiv$ 
```

```
   $t = g\text{-vertices}$ ;  $tmax = t + g\text{-}n - 1$ ;
```

```
   $x\text{-taken} = 1$ ;  $t\text{-vert} = x$ ;
```

```
   $t\text{-ark} = \Lambda$ ;
```

```
advance: < Increase  $t$  and update the data structures to show that vertex  $v$  is now taken; goto backtrack if no further moves are possible 5 >;
```

```
try: < Look at edge  $a$  and its successors, advancing if it is a valid move 7 >;
```

```
restore: < Dnupdate the data structures to the state they were in when level  $t$  was entered 6 >;
```

```
backtrack: < Decrease  $t$ , if possible, and try the next possibility; or goto done 8 >;
```

```
done:
```

This code is used in section 1.

**5.** < Increase  $t$  and update the data structures to show that vertex  $v$  is now taken; **goto** *backtrack* if no further moves are possible 5 >  $\equiv$

```
   $t++$ ;
```

```
   $t\text{-vert} = v$ ;
```

```
   $v\text{-taken} = 1$ ;
```

```
  if ( $v \equiv z$ ) {
```

```
    if ( $t \equiv tmax$ ) < Record a solution 9 >;
```

```
    goto backtrack;
```

```
  }
```

```
  for ( $aa = v\text{-arcs}$ ,  $y = \Lambda$ ;  $aa$ ;  $aa = aa\text{-next}$ ) {
```

```
     $u = aa\text{-tip}$ ;
```

```
     $d = u\text{-deg} - 1$ ;
```

```
    if ( $d \equiv 1 \wedge u\text{-taken} \equiv 0$ ) {
```

```
      if ( $y$ ) goto restore; /* restoration will stop at  $aa$  */
```

```
       $y = u$ ;
```

```
    }
```

```
     $u\text{-deg} = d$ ;
```

```
  }
```

```
  if ( $y$ ) {
```

```
     $t\text{-ark} = \Lambda$ ;
```

```
     $v = y$ ;
```

```
    goto advance;
```

```
  }
```

```
   $a = v\text{-arcs}$ ;
```

This code is used in section 4.

**6.** < Dnupdate the data structures to the state they were in when level  $t$  was entered 6 >  $\equiv$

```
  for ( $a = t\text{-vert}\text{-arcs}$ ;  $a \neq aa$ ;  $a = a\text{-next}$ )  $a\text{-tip}\text{-deg}++$ ;
```

This code is used in section 4.

7.  $\langle$  Look at edge  $a$  and its successors, advancing if it is a valid move 7  $\rangle \equiv$

```

while ( $a$ ) {
     $v = a \rightarrow tip$ ;
    if ( $v \rightarrow taken \equiv 0$ ) {
         $t \rightarrow ark = a$ ;
        goto advance;
    }
     $a = a \rightarrow next$ ;
}
restore_all:  $aa = \Lambda$ ;    /* all moves tried; we fall through to restore */

```

This code is used in section 4.

8.  $\langle$  Decrease  $t$ , if possible, and try the next possibility; or **goto** *done* 8  $\rangle \equiv$

```

 $t \rightarrow vert \rightarrow taken = 0$ ;
 $t--$ ;
if ( $t \rightarrow ark$ ) {
     $a = t \rightarrow ark \rightarrow next$ ;
    goto try;
}
if ( $t \equiv g \rightarrow vertices$ ) goto done;
goto restore_all;    /* the move was forced */

```

This code is used in section 4.

9.  $\langle$  Record a solution 9  $\rangle \equiv$

```

{
     $count++$ ;
    if ( $count \% modulus \equiv 0$ ) {
         $printf("%d:\square", count)$ ;
        for ( $u = g \rightarrow vertices$ ;  $u \leq tmax$ ;  $u++$ )  $printf("%s\square", u \rightarrow vert \rightarrow name)$ ;
         $printf("\n")$ ;
    }
}

```

This code is used in section 5.

**10. Index.**

*a*: [1](#).  
*aa*: [1](#), [5](#), [6](#), [7](#).  
*advance*: [4](#), [5](#), [7](#).  
**Arc**: [1](#).  
*arcs*: [1](#), [3](#), [5](#), [6](#).  
*argc*: [1](#), [2](#).  
*argv*: [1](#), [2](#).  
*ark*: [4](#), [5](#), [7](#), [8](#).  
*b*: [1](#).  
*backtrack*: [4](#), [5](#).  
*bb*: [1](#).  
*count*: [1](#), [9](#).  
*d*: [1](#).  
*deg*: [1](#), [3](#), [5](#), [6](#).  
*dmin*: [1](#), [3](#).  
*done*: [4](#), [8](#).  
*exit*: [2](#).  
*fprintf*: [2](#).  
*g*: [1](#).  
**Graph**: [1](#).  
*main*: [1](#).  
*modulus*: [1](#), [2](#), [9](#).  
*name*: [1](#), [9](#).  
*next*: [1](#), [3](#), [5](#), [6](#), [7](#), [8](#).  
*printf*: [1](#), [9](#).  
*restore*: [4](#), [5](#), [7](#).  
*restore\_all*: [7](#), [8](#).  
*restore\_graph*: [1](#), [2](#).  
*sscanf*: [2](#).  
*stderr*: [2](#).  
*t*: [1](#).  
*taken*: [3](#), [4](#), [5](#), [7](#), [8](#).  
*tip*: [1](#), [5](#), [6](#), [7](#).  
*tmax*: [1](#), [4](#), [5](#), [9](#).  
*try*: [4](#), [8](#).  
*u*: [1](#).  
*v*: [1](#).  
*vert*: [4](#), [5](#), [6](#), [8](#), [9](#).  
**Vertex**: [1](#).  
*vertices*: [1](#), [3](#), [4](#), [8](#), [9](#).  
*x*: [1](#).  
*y*: [1](#).  
*z*: [1](#).

- ⟨ Decrease  $t$ , if possible, and try the next possibility; or **goto** *done* 8 ⟩    Used in section 4.
- ⟨ Downdate the data structures to the state they were in when level  $t$  was entered 6 ⟩    Used in section 4.
- ⟨ Find all simple paths of length  $g \rightarrow n - 2$  from  $v$  to  $z$ , avoiding  $x$  4 ⟩    Used in section 1.
- ⟨ Increase  $t$  and update the data structures to show that vertex  $v$  is now taken; **goto** *backtrack* if no further moves are possible 5 ⟩    Used in section 4.
- ⟨ Look at edge  $a$  and its successors, advancing if it is a valid move 7 ⟩    Used in section 4.
- ⟨ Prepare  $g$  for backtracking, and find a vertex  $x$  of minimum degree 3 ⟩    Used in section 1.
- ⟨ Process the command line, inputting the graph 2 ⟩    Used in section 1.
- ⟨ Record a solution 9 ⟩    Used in section 5.

# HAM

	Section	Page
Hamiltonian cycles .....	<a href="#">1</a>	1
The data structures .....	<a href="#">4</a>	3
Index .....	<a href="#">10</a>	5