Experimental Results on Hamiltonian-Cycle-Finding Algorithms

Gabriel Nivasch

May 9, 2003

1 Introduction

Frieze [1] introduced a heuristic polynomial-time algorithm, Ham, for finding Hamiltonian cycles in random graphs with high probability. We wanted to see how this algorithm performs in practice, and whether it could be improved by modifying it.

For this purpose, we borrowed an idea from an algorithm by Keydar called SemiHam [2]. SemiHam is a modification of Ham that finds Hamiltonian cycles in semi-random graphs with high probability. (The notions of random and semi-random graphs are beyond the scope of this report.)

For simplicity, we call our implementations by the borrowed names Ham and SemiHam, even though we don't follow completely the original theoretical algorithms.

We tested our algorithms on knight's tour problems. By analogy from chess, we define a *knight* as a piece that moves in a board either $\pm a$ cells horizontally and $\pm b$ cells vertically, or $\pm b$ cells horizontally and $\pm a$ cells vertically, for given a and b. Then, given an $m \times n$ rectangular board, we ask whether there exists a sequence of moves in which a knight visits each cell of the board exactly once, and then returns to its starting cell. This is the *knight's tour problem*.

We denote an instance of this problem by listing its parameters in the form $m \times n$–(a, b). For example, the classical knight's tour problem involves a regular knight and a regular chessboard, and it is denoted 8×8–$(1, 2)$.

A knight's tour problem corresponds in a trivial way to a Hamiltonian cycle problem on a graph: Each cell of the board corresponds to a vertex, and each legal knight move between cells corresponds to an edge. Then a knight's tour of the board corresponds to a Hamiltonian cycle in the graph.

Vandegriend in [3] analyzes the knight's tour problem in some detail. He gives several existence and non-existence theorems for different parameter values. He also reports on extensive computer experiments, in which he used an exhaustive Hamiltonian cycle algorithm. (Our algorithms, in contrast, are heuristic, as we mentioned). We will compare our results to his.

2 The Test Graphs

We wanted to find whether the improvements introduced in SemiHam make it perform better than the original Ham. So we looked for Hamiltonian graphs with different levels of difficulty, and for this we turned to knight's tours. In order to make the problems more difficult, we also allowed the knight some *fake moves*.

Consider coloring the cells of the board alternatingly black and white, like a regular chessboard. Then, if the knight's movement is given by (a, b) where $a + b$ is even, the knight always moves among cells of the same color. Then the cells of the other color cannot be visited, so the problem has no solution.

On the other hand, if $a + b$ is odd, then the knight always alternates cell colors. Now choose any number of pairs of white cells, and for each pair of cells allow the knight to move from one white cell to the other. If the knight uses any of these moves, he will use up an additional white cell instead of a black cell, thus creating a surplus of remaining black cells that he will never be able to balance. Therefore, these additional white-to-white moves do not make the problem any easier.

These *fake moves* correspond to *fake edges* in the corresponding graph—edges that provably cannot be part of a Hamiltonian cycle. We expected these fake edges to make the Hamiltonian problems more difficult for our algorithms. We will see what the actual results were.

We tested Ham and SemiHam on several sets of parameters. The parameters are the board dimensions $m \times n$, the knight's move (a, b), and the number of fake edges f, where $0 \le f \le \binom{mn/2}{2}$. For each set of parameters, we ran many trials and compared the algorithms' success rates.

Since both algorithms are completely deterministic, as we will describe, we introduced randomness by choosing the locations of the f fake edges uniformly at random, and then applying a uniformly random permutation to the vertices of the graph.

3 The Algorithms

We proceed to describe our implementations of Ham and SemiHam. We start with Ham.

3.1 Ham

The input to Ham is an N-vertex undirected graph, where the vertices are numbered 0 through $N - 1$. The edges are specified by a symmetric $N \times N$ adjacency matrix.

Ham works in stages. For each stage i, for $i = 1, \ldots, N$, the input to the stage is a path with $i - 1$ edges, and the output, if the stage is successful, is a path with i edges. We start at stage 1 with a path that consists just of vertex 0 and no edges. Then we proceed with the next stages in sequence.

If we manage to complete stage N, we have found a Hamiltonian path in the graph. Then comes stage $N+1$, in which we try to close the path into a Hamiltonian cycle. If the final stage is successful, then the found cycle is our output.

A path is represented by a structure that includes an array v of length N, and an integer *length* that specifies the number of vertices in the path. The vertices are stored in $v[0]$ through $v[length-1]$. We call $v[0]$ the *head* of the path, and $v[length-1]$ the *tail*.

For convenience, the algorithm keeps track of which vertices are already used in the path, with a boolean array *vertexUsed* of length N.

The Stages

Each stage from 1 to N proceeds as follows:

We first build a boolean array *externalNbr* of length N, which specifies, for each vertex in the path, the vertex's smallest neighbor that is external to the path. If a vertex in the path has no external neighbors, its corresponding entry in *externalNbr* is -1. For vertices not in the path, the entry in *externalNbr* is undefined.

We first try to extend the path in the most obvious ways:

(E1) If *externalNbr* of the head is not -1, then we extend the path along the head to the given external vertex, the external vertex becoming the new head.

(E2) If *externalNbr* of the tail is not -1, then we extend the path along the tail to the given external vertex, the external vertex becoming the new tail.

(E3) If the head and the tail are joined by an edge, so that the current path is actually a cycle, then we can extend the path as follows:

We examine the path vertices $v[0], v[1], \ldots$ sequentially until we find a vertex $v[i]$ whose corresponding *externalNbr* entry is $t \neq -1$. Such a vertex is guaranteed to exist—otherwise our path would be disconnected from the rest of the graph.

Once we find i and t, we rearrange the vertices in the following order: t, followed by $v[i], \ldots, v[0]$, followed by $v[len-1], \ldots, v[i+1]$.

An extension according to (E1) or (E2) is known as a *simple extension*, and an extension according to (E3) is known as a *cycle extension*.

If none of these extensions is possible on the original path, then we proceed to build new paths by applying *rotations* to the original path, and further rotations to the "rotated" paths, until we find a path that can be extended with either a simple or a cycle extension. All the rotated paths have the same vertices as the original path—just in a different order—so the arrays *vertexUsed* and *externalNbr* are valid for all of them.

The rotations are performed as follows:

We allocate an array of paths *pathList*, of size *MaxPaths*, where *MaxPaths* is an adjustable parameter. At the beginning, the the original path is stored at *pathList*[0]. There are two indices to *pathList*: *listCurr* and *listLen*, which indicate, respectively, the current path being examined and the total number of paths on the list. At the beginning, $listCurr = 0$ and $listLen = 1$.

We repeat the following steps until we succeed in extending a path, or until we run out of space in the *pathList* array:

(S1) Take the path *pathList*[*listCurr*], and successively examine its vertices $v[2], \ldots, v[length - 2]$, to see which ones are connected to the head by an edge. For each such vertex $v[i]$ do the following:

  (a) Create a new path in which the order of the vertices is $v[i-1], \ldots, v[0]$, followed by $v[i], \ldots, v[len - 1]$.

  (b) Add this path to entry *pathList*[*listLen*], and increment *listLen*.

  (c) Check if this new path can be extended according to (E1) or (E3) above.

(S2) Again take the path *pathList*[*listCurr*], and now successively examine its vertices $v[1], \ldots, v[length - 3]$, to see which ones are connected to the tail by an edge. For each such vertex $v[i]$ do the following:

  (a) Create a new path in which the order of the vertices is $v[0], \ldots, v[i]$, followed by $v[len - 1], \ldots, v[i + 1]$.

  (b) Add this path to entry *pathList*[*listLen*], and increment *listLen*.

  (c) Check if this new path can be extended according to (E2) or (E3) above.

(S3) Increment *listCurr*, and go back to step (S1).

If at any point we succeed in extending a path, then the current stage immediately ends in success. On the other hand, if *pathList* becomes completely full, or if there are no more paths to rotate because *listCurr* = *listLen*, and no extension was found, then the current stage—and the whole execution of Ham—ends in failure.

(Note that Ham wastes some effort in performing rotations that are exact undoings of previous rotations, thus examining some paths multiple times.)

The final stage $N + 1$ is almost the same as the preceding stages. First we check whether the input path is already a cycle. If it isn't, we perform the rotations as before, except that for each rotated path we only check whether it is a cycle or not.

3.2 SemiHam

SemiHam works in stages 1 through $N + 1$ exactly like Ham. At each stage, it first tries to perform a simple or cycle extension on the path, just as Ham.

However, when it comes to generate the rotations, SemiHam imposes an additional constraint: *pathList* can contain at most one path with a given pair of head and tail endpoints. We enforce this constraint with an $N \times N$ boolean array *pathEnds*, which keeps track of which $\langle head, tail \rangle$ pairs are already present in *pathList*.

Before creating each new path in steps (S1) and (S2) above, SemiHam checks in *pathEnds* that the new path's $\langle head, tail \rangle$ pair has not been seen already.

Also, in SemiHam *MaxPaths* is given the value N^2, which provides for more than enough space for rotations (since the maximum number of rotations is $N(N-1)$, because paths with $head = tail$ are not possible).

In order to make the comparison between Ham and SemiHam fair, in Ham we also let $MaxPaths = N^2$ (even though this gives Ham a small advantage).

Note that, as we have said, both Ham and SemiHam are completely deterministic.

We include in Appendix A our source code for initializing the graph to a knight's tour problem, and for the SemiHam algorithm. The Ham algorithm is obtained through some easy modifications to SemiHam.

4 Experimental Results

We found that, in general, SemiHam always has a higher success rate than Ham (except when both have a 100% success rate).

For the classical knight's tour problem (8×8–$(1, 2)$ with no fake edges), both algorithms have a 100% success rate. The classical knight's tour problem is therefore fairly easy.

However, as we increase the number of fake edges from $f = 1$ to $f = 16*31 = 496$, Ham starts to fail with increasing frequency. Sometimes it gets only to stage 64, or even less (see Table 1). Still, the algorithm experiences difficulties only at the very last stages.

By contrast, SemiHam's success rate always stays at 100%. This already shows an advantage of SemiHam over Ham on certain graphs.

We proceeded to try other problems. Vandegriend provides a table (Table 5.16 in [3]) of "ultrahard" knight's tours that have solutions, but which his algorithm had difficulty in solving. We took a few entries from this table and tested them, each with a range of values for f. The results are shown in Table 2.

We see that SemiHam always outperforms Ham. Further, Ham's performance degrades as more fake edges are added. However, SemiHam's performance *improves* as more fake edges are added! This is certainly a surprising result. We comment on this in the Conclusion.

	Ham				SemiHam
Fake edges	65	64	63	62−	65
0	1000	0	0	0	1000
1	950	50	0	0	1000
10	770	225	5	0	1000
100	605	361	34	0	1000
200	552	407	40	1	1000
400	487	456	55	2	1000
496	460	470	69	1	1000

Table 1: Latest stage completed on 8×8–$(1,2)$ (1000 trials).

Board	**Fake edges**	**Ham**	**SemiHam**
3×34–$(1,2)$	0	38	88
	1	26	86
	5	27	95
	10	13	93
	100	7	100
	1275	6	100
9×12–$(1,4)$	0	0	34
	1	0	48
	5	0	66
	10	0	76
	20	0	83
	100	0	82
	1431	0	86
7×36–$(3,4)$	0	3	10
	10	0	37
	100	0	62
	1000	0	64
	7875	0	64
10×11–$(1,4)$	0	16	97
	1	19	98
	10	2	100
	1485	0	100

Table 2: Successes in 100 trials.

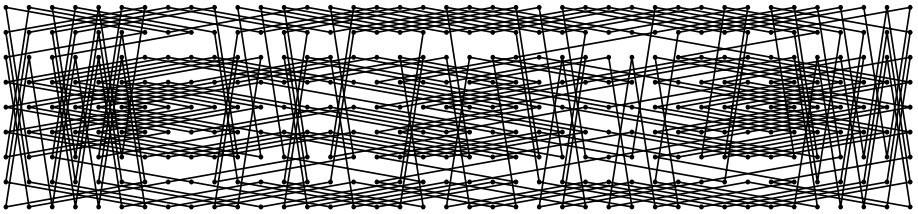

Figure 1: A solution to 9×40–$(1,6)$

Board	**Fake edges**	**Ham**	**SemiHam**
9×40–$(1,6)$	0	6	78
	16110	0	93
13×28–$(2,7)$	0	2	98
	16471	0	98

Table 3: Successes in 100 trials.

Comparing our results to Vandegriend's, we observe that there is little correlation between his success rates and ours. For example, in 7×36–$(3,4)$, Vandegriend had 4 successes out of 10, while in 10×11–$(1,4)$, he had only 1 success out of 10.

Vandegriend also provides another table (Table 5.12 in [3]) with problems for which his algorithm could not determine whether they have solution or not (Vandegriend placed a time limit on his exhaustive algorithm). We tried SemiHam on these instances, and we found that two of them have solution: 9×40–$(1,6)$ and 13×28–$(2,7)$. (See Figures 1 and 2.)

Table 3 compares Ham and SemiHam on these two instances, with no fake edges and all possible fake edges.

Finally, to give an idea of the running times involved, Table 4 gives the average running time of Ham and SemiHam on the various boards we tested, with $f = 0$. We used a 266–MHz Power Macintosh G3. We see that SemiHam, besides having a higher success rate, is also faster than Ham—especially in larger boards. (However, with many fake edges SemiHam can become as slow as Ham; this is not shown in the table.) In any case, the running times of both algorithms range from milliseconds to seconds, so they are very reasonable.

5 Conclusion

We experimented with two heuristic algorithms for finding Hamiltonian cycles, which we call Ham and SemiHam. The difference between them is that SemiHam does not allow paths with repeated pairs of endpoints when performing rotations.

We tested them on knight's tour problems, to which we added different

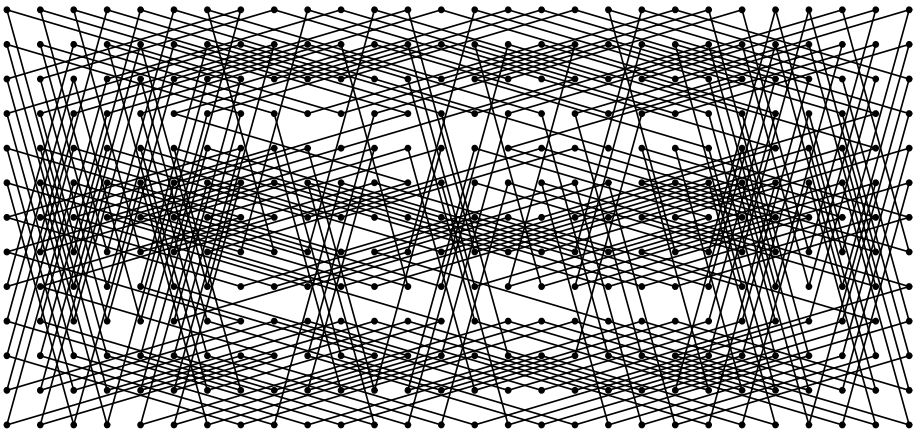

Figure 2: A solution to 13×28–$(2,7)$

Board	**Ham**	**SemiHam**
8×8–$(1,2)$	2.1	2.2
3×34–$(1,2)$	42	10
9×12–$(1,4)$	80	16
10×11–$(1,4)$	73	17
7×36–$(3,4)$	910	110
9×40–$(1,6)$	3080	520
13×28–$(2,7)$	3400	700

Table 4: Average running times (milliseconds).

numbers of *fake edges*—edges that provably cannot be part of the solution. We found that SemiHam always outperforms Ham, in terms of success rate. We also found that as more fake edges are added, Ham performs worse, but, surprisingly, SemiHam performs better.

Apparently, the fake edges allow SemiHam to perform more rotations at certain stages, giving it a greater chance of extending the path. Later on, SemiHam has little difficulty in getting rid of the fake edges and succeeding.

In fact, Vandegriend [3] (Section 3.4.3) speculated that this could happen. He suggested that, for heuristic algorithms, initially pruning unnecessary edges in the graph might prove counterproductive, since those edges might help the algorithm avoid reaching an early dead-end. This is indeed what happens with SemiHam.

Given its reasonable success rates and its low running times, our implementation of SemiHam is a good algorithm for solving knight's tour problems in practice. We speculate that it is also good for other classes of Hamiltonian graphs.

References

[1] B. Bollobas, T. I. Fenner, and A. M. Frieze. An algorithm for finding Hamilton paths and cycles in random graphs. *Combinatorica*, 7(4):327–341, 1987.

[2] Eran Keydar. *Finding Hamiltonian cycles in semi-random graphs*. Masters thesis, Weizmann Institute of Science, 2002.

[3] Basil Vandegriend. *Finding Hamiltonian cycles: Algorithms, graphs and performance*. Masters thesis, University of Alberta, February 1998.

A Source Code

A.1 Graph Initialization

```
const int BoardX = 9, BoardY = 40, KnightX = 1, KnightY = 6,
    N = BoardX*BoardY, //This better be even
    MaxFakeEdges = N/2 * (N/2 - 1) / 2,
    NFakeEdges = MaxFakeEdges;
int G[N][N], //The graph
    perm[N], pinv[N]; //will hold a random permutation and its inverse

//Generates a random permutation of n elements and its inverse:
void randomPerm(int n, int *a, int *b)
{
    int i, r, temp;
    for (i=0; i<n; i++)
        a[i]=b[i]=i;
    for (i=0; i<n; i++)
    {
```

```
        r = i + (rand() % (n-i)); //Random location between i and n-1
        temp=a[i]; a[i]=a[r]; a[r]=temp;
        temp=b[a[i]]; b[a[i]]=b[a[r]]; b[a[r]]=temp;
    }
} //randomPerm()

//Returns a random white square:
void randomWhite(int *x, int *y)
{
    *x = rand()%BoardX;
    if (*x%2==0)
    {
        *y = rand() % (BoardY/2);
        *y = 2*(*y)+1;
    }
    else
    {
        *y = rand() % ((BoardY+1)/2);
        *y = 2*(*y);
    }
} //randomWhite()

int Abs(int x) { return x>=0?x:-x; }

//Inits the graph to a knight's tour problem,
//with a certain number of fake edges to make it more difficult:
void initG()
{
    int x1, y1, x2, y2;
    randomPerm(N, perm, pinv); //Permute the cells randomly
    for (x1=0; x1<BoardX; x1++)
    for (y1=0; y1<BoardY; y1++)
    {
        int from=perm[x1*BoardY + y1];
        for (x2=0; x2<BoardX; x2++)
        for (y2=0; y2<BoardY; y2++)
        {
            int to=perm[x2*BoardY + y2];
            if ((Abs(x2-x1)==Abs(KnightX) && Abs(y2-y1)==Abs(KnightY))
                ||(Abs(x2-x1)==Abs(KnightY) && Abs(y2-y1)==Abs(KnightX)))
                G[from][to]=1;
            else
                G[from][to]=0;
        }
    }
    //Add the fake edges (somewhat inefficiently):
    for (int added=0; added<NFakeEdges; added++)
    {
        int r1, r2;
        do
```

```
        {
            randomWhite(&x1, &y1);
            r1=x1*BoardY + y1;
            do
            {
                randomWhite(&x2, &y2);
                r2=x2*BoardY + y2;
            }
            while (r1==r2);
        }
        while (G[perm[r1]][perm[r2]]);
        G[perm[r1]][perm[r2]]=G[perm[r2]][perm[r1]]=1;
    }
} //initG()
```

A.2 SemiHam

```
struct path {short length, v[N];};

//Inverts a chunk of an array:
void invert(short *a, int s, int t)
{
    short temp;
    while (s<t)
    { temp=a[s]; a[s++]=a[t]; a[t--]=temp; }
} //invert()

bool pathIsCycle(const path *p)
{ return G[p->v[0]][p->v[p->length-1]]==1; }

//Does the specifiend simple extension to the input path:
void simpleExtendPath(path *p, bool fromHead, int newV)
{
    int i, len=p->length;
    //Check validity:
    if (fromHead) assert(G[p->v[0]][newV]);
    else assert(G[p->v[len-1]][newV]);
    for (i=0; i<len; i++) assert(p->v[i]!=newV);

    if (fromHead)
    {
        for (i=len; i>0; i--)
            p->v[i] = p->v[i-1];
        p->v[0]=newV;
    }
    else //from tail
        p->v[len]=newV;
    p->length++;
} //simpleExtendPath()
```

```
//Does a cycle extension to the input path, which is a cycle,
//using the given vertex in the path and the given external vertex.
void cycleExtendPath(path *p, int vtxIndex, int externalVtx)
{
    int len=p->length;
    assert(pathIsCycle(p));
    assert(G[p->v[vtxIndex]][externalVtx]==1);
    //Re-arrange vertices:
    p->v[len]=externalVtx;
    invert(p->v, vtxIndex + 1, len);
    invert(p->v, 0, vtxIndex + 1);
    p->length++;
} //cycleExtendPath()

//Tries the simple and cycle extensions on the given path.
//p and vertexUsed are input/output parameters.
//externalNbr is an input array which specifies, for each vertex
//  in the path, its smallest external neighbor, if it has;
//  otherwise -1.
bool tryExtendPath(path *p, bool *vertexUsed, int *externalNbr)
{
    int newVtx, i, len = p->length;
    //Try head extension:
    newVtx = externalNbr[p->v[0]];
    if (newVtx != -1)
    {
        simpleExtendPath(p, true, newVtx);
        vertexUsed[newVtx]=true;
        return true;
    }
    //Try tail extension:
    newVtx = externalNbr[p->v[len-1]];
    if (newVtx != -1)
    {
        simpleExtendPath(p, false, newVtx);
        vertexUsed[newVtx]=true;
        return true;
    }
    //Try cycle extension:
    if (pathIsCycle(p))
    {
        for (i=0; i<len; i++)
        {
            newVtx = externalNbr[p->v[i]];
            if (newVtx != -1)
            {
                cycleExtendPath(p, i, newVtx);
                vertexUsed[newVtx]=true;
                return true;
            }
```

```
        }
        //Cycle with no external neighbors. This shouldn't happen.
        assert(0);
    }
    return false;
} //tryExtendPath()

//Performs a stage of SemiHam. Returns true if successful.
//The parameter justCloseCycle is true on the last stage.
bool SemiHamStage(path *p, bool *vertexUsed, bool justCloseCycle)
{
    int i, j, len = p->length;
    bool extended;
    //Get the smallest external neighbor for each vertex in the path:
    int externalNbr[N];
    for (i=0; i<len; i++)
    {
        int vtx = p->v[i];
        externalNbr[vtx]=-1;
        for (j=0; j<N; j++)
            if (vertexUsed[j]==0 && G[vtx][j])
            { externalNbr[vtx]=j; break; }
    }
    //Try to extend the input path; or check if it's a cycle:
    if (justCloseCycle)
    {
        if (pathIsCycle(p))
            return true;
    }
    else
    {
        extended=tryExtendPath(p, vertexUsed, externalNbr);
        if (extended)
            return true;
    }

    const int MaxPahts=N*N;
    static path pathList[MaxPaths];
    path *currP, *newP;
    int listCurr=0, listLen=1;
    pathList[0]=*p;
    //Keep only one path for each (head, tail) combination.
    static bool pathEnds[N][N];
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            pathEnds[i][j]=false;
    pathEnds[p->v[0]][p->v[len-1]]=true;
    //Perform rotations:
    while(1)
    {
```

```
if (listCurr >= listLen)
    return false; //No more paths to rotate
currP=pathList+listCurr;
int head=currP->v[0], tail=currP->v[len-1];
//Rotate from head:
for (i=2; i < len; i++)
    if (G[head][currP->v[i]]
        && pathEnds[currP->v[i-1]][tail]==false)
    {
        newP = pathList + listLen;
        //Add a copy of the current path to the end of the list:
        *newP=*currP;
        //Perform the rotation on the path copy:
        invert(newP->v, 0, i-1);
        //Mark the new (head, tail) combination in pathEnds:
        pathEnds[newP->v[0]][newP->v[len-1]]=true;
        //Try to extend the new path; or check if it's a cycle:
        if (justCloseCycle)
        {
            if (pathIsCycle(newP))
            {
                *p=*newP;
                return true;
            }
        }
        else
        {
            extended=tryExtendPath(newP, vertexUsed, externalNbr);
            if (extended)
            {
                *p=*newP;
                return true;
            }
        }
        //Increment listLen:
        listLen++;
        assert(listLen<MaxPaths); //Overflow shouldn't occur.
    }
//Rotate from tail, similarly:
for (i=0; i < len-2; i++)
    if (G[tail][currP->v[i]]
        && pathEnds[head][currP->v[i+1]]==false)
    {
        newP = pathList + listLen;
        *newP=*currP;
        invert(newP->v, i+1, len-1);
        pathEnds[newP->v[0]][newP->v[len-1]]=true;
        if (justCloseCycle)
        {
            if (pathIsCycle(newP))
```

```
                    { *p=*newP; return true; }
                }
                else
                {
                    extended=tryExtendPath(newP, vertexUsed, externalNbr);
                    if (extended)
                    { *p=*newP; return true; }
                }
                listLen++;
                assert(listLen<MaxPaths);
            }
        //Increment listCurr:
        listCurr++;
    } //while 1
} //SemiHamStage()

//The SemiHam algorithm.
//Tries to find a Hamiltonian cycle on the graph G.
//If successful, outputs true and returns the cycle in p.
bool SemiHam(path *p)
{
    int len, listLen;
    bool wasExtended;
    //Initialize path to just vertex 0:
    p->v[0]=0;
    p->length=1;
    //Initialize vertexUsed:
    bool vertexUsed[N];
    vertexUsed[0]=true;
    for (int i=1; i<N; i++)
        vertexUsed[i]=false;
    //Do stages until path has full length:
    for (len=1; len<N; len++)
    {
        assert(p->length==len);
        wasExtended=SemiHamStage(p, vertexUsed, false);
        if (!wasExtended)
            return false;
    }
    //Last stage: Try to close the path into a cycle:
    assert(p->length==N);
    wasExtended=SemiHamStage(p, vertexUsed, true);
    if (!wasExtended)
        return false;
    return true;
} //SemiHam()
```