Finding Hamiltonian Cycles in the Inner n-Cube

Second Paper

Brent M. Dingle

CPSC 626
Winter Interim 1998-1999
Texas A&M University

Abstract:
In this paper we will be examining general algorithms for finding Hamiltonian Cycles in the inner cube of a given n-cube. All of the algorithms presented will be based on a backtracking scheme with various heuristics (or absolute guessing) "thrown in." The purpose of this paper is to present several attempts at solving this problem and the various techniques used. The overall intent of the research was to find some sequential method that would be guaranteed to solve the problem which could be "sped up" via a parallel implementation.
Also in this paper will be found a discussion of the repetitive structure of the inner n-cube.

It is assumed the reader is familiar with the problem of finding a Hamiltonian cycle on the inner n-cube so no definitions nor detailed explanation of such will be given here.

For all of the implementations of the algorithms we are about to discuss we have opted to use a three function version of a backtracking based algorithm. So throughout this paper we will be using variations of these three general functions: *FindCycle()*, *BackTrack()*, and *HamCycle()*. In these functions, we assume we already have established the adjacency matrix and know the length of the cycle for which we are looking. The general idea is as follows, it is based from Fortran code presented in [10]:

In all of the methods presented, *FindCycle()* is changed only by changing the actual function names for *BackTrack()* and *HamCycle()*. Here is the general appearance of *FindCycle()*:

```
long FindCycle()
{
  long A[GRAPH_MAXVERT+1];      // the cycle, coded by vertex index
  long stack[GRAPH_STACK_MAX+1];
  long index, k, m, nstk;
  long count;

  count = 0;        // count keeps track of how many ham. cycles found
  nstk = GRAPH_STACK_MAX;   // maximum stack size
  index = 0;

  // num_verts = length of expected cycles
  // index = 0 should set k = 1, m = 0 on first call to BackTrack

  // BackTrack(...) sets
  // index = 1 for successfully done,
  // index = 3 means  no more vectors of type sought
  // index = 2 for call HamCycle

  while (index != 3)
  {
    BackTrack(NumVerts, A, &index, &k, &m, stack, nstk);
    if (index == 2)
    {
      HamCycle(NumVerts, A, k, &m, stack, nstk, AdjMat);
    }
    else if (index == 1)  // found a cycle
    {
      count++;          // count = total number of cycles found
                        // could also print A[] here to display the cycle
    }
  } // end while index !=3 (i.e. we are not done)

  printf("\nDone.\n\n");
  return 1;
} // end FindCycle
```

The general appearance of *BackTrack()* is:

```
long BackTrack(long stop_len, long A[GRAPH_MAXVERT+1],
                        long *state, long *A_index, long *stack_index,
                        long Stack[GRAPH_STACK_MAX+1], long nstk)
// stop_len is the desired length of the a complete output vector
// A is the output vector
// On entry:
// state = 0 for just starting
// state = 2 for keep going     (shouldn't ever be 1 or 3 on entry)
//    BackTrack(...) sets
//    state = 1 for successfully done, (size of A equals stop_len)
//    state = 3 means  no more vectors of type sought
//    state = 2 for call HamCycle
// A_index is the length of a partially constructed vector:
//         a call to HamCycle() is a request for position A[A_index];
//         A_index is set by BackTrack()
// stack_index is the location of the last item on the stack;
//         it is changed by both BackTrack() and HamCycle()
// stack is a linear array, of maximal length nstk, whose appearance
//    at a typical step is:
// z z z z z z z n1 z z z z z n2 z z z z n3 z z...
// \_________/    \_______/
//     \/                \/
//  n1 candidates      n2 candidates
//  for position       for position          … etc
//   A(1)               A(2)
//
// i.e. let the lists of candidates for A(1),...,A(A_index-1), A(A_index)
// be stored in stack[], each list followed by its length
//
////////////////////////////////////////////////////////////////////////////
{
  long num_cand, done;
  long out_vert_index, cur_nhbr;
  long i;

  if (*state == 0)
  {
    *A_index = 1;
    *stack_index = 0;
    *state = 2;

    return 2;
  }

  done = 0;
  while (!done)
  {
    num_cand = Stack[*stack_index];
    *stack_index = *stack_index - 1;

    if (num_cand != 0)
    {
      done = 1;
    }
```

```
    else  // if (num_cand == 0)
    {
      // So we are going to pop off an element from A by decrementing A_index.

      *A_index = *A_index -1;
      if (*A_index == 0)
      {
        *state = 3;   // ran out of options, so fail
        return 3;
      }
    }
  } // end while !done
  // Now we pop off one of the elements and run with it.
  A[*A_index] = Stack[*stack_index];
  Stack[*stack_index] = num_cand - 1;
  if (*A_index == stop_len)                  // Check if we are done
  {
    *state = 1;
    return 1;
  }
  *A_index = *A_index + 1;
  *state = 2;
  return 2;
} // end BackTrack
```

The general appearance of *HamCycle()* is:

```
long HamCycle(long num_verts, long A[GRAPH_MAXVERT+1],
              long A_index, long *stack_index,
              long Stack[GRAPH_STACK_MAX+1], long nstk)
//
// num_verts = number of vertices in graph G
// A => A(i) is the i-th vertex in the current circuit
// A_index = index of the next step on the current partial circuit
// stack_index = current length of stack i
// stack = candidates for steps 1 to k-1
// nstk = maximum length of stack
//
// m_AdjMat[][] => if (m_AdjMat[i][j] = 1) then edge (i, j) is in G
//
// Purpose: Find the candidates for k-th vertex in Ham. cycle
// store them in stack and update m accordingly
////////////////////////////////////////////////////////////////////////
{
  long vert[GRAPH_MAXVERT+1];  // temp storage of what verts adj. to A[k-1]
  long i;                                  // standard looping index
  long k1, a1, m1;
  long min_num_nhbrs, nhbr_vert;

  if (A_index == 1)
  {
    Stack[1] = 1;              // put vert 1 first
    Stack[2] = 1;              // then there is 1 element in that section of the stack
    *stack_index = 2;
    return 2;
  }
```

```
  // Find ALL the verts adjacent to A[A_index-1]
  k1 = A_index - 1;
  a1 = A[k1];
  for (i=1; i <= num_verts; i++)
  {
    vert[i] = m_AdjMat[a1][i];
  }

  // Make sure we remove all verts we have previously visited.
  for (i=1; i<= k1; i++)
  {
    m1 = A[i];
    vert[m1] = 0;       // 0 = false, 1 = true
  }

  m1 = *stack_index;
  if (A_index != num_verts)
  {
    for (i=1; i<= num_verts; i++)
    {
      if (vert[i] == 1)
      {
        m1 = m1 + 1;
        Stack[m1] = i;
      }
    } // end for
    Stack[m1+1] = m1 - (*stack_index);
    *stack_index = m1 + 1;
    return (m1 + 1);
  }  // end if k != num_verts

  for (i=1; i <= num_verts; i++)
  {
    if (vert[i])
    {
      if (!m_AdjMat[i][1])
      {
        Stack[m1+1] = m1 - (*stack_index);
        *stack_index = m1 + 1;
        return (m1+1);
      }
      *stack_index = *stack_index +2;
      Stack[*stack_index - 1] = i;
      Stack[*stack_index] = 1;
      return (*stack_index);
    }
  } // end for

  Stack[m1+1] = m1 - (*stack_index);
  *stack_index = m1 + 1;

  return (m1+1);

} // end HamCycle
```

ASIDE:
Since all of the following methods are based on backtracking it may be important to note that the backtracking method is almost directly related to a depth first graph traversal. This may be significant in porting any backtracking algorithm over to a parallel machine.

For our comparison of methods we choose to examine the n = 5 and n = 7 cases.
The solution time on a Pentium 266 Windows 95 machine for the 5 case using the above "do nothing special" backtracking algorithms takes about two seconds to solve completely (arriving at 48 "different" cycles). The 7 case takes about 2.5 – 3.0 hours to arrive at the first finding of a cycle.

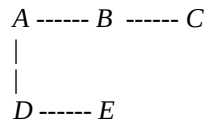
METHOD 1:
Our first method of attack was to always choose the vertex of minimal degree. To illustrate this, assume we are at vertex *A*, which has neighbors *B*, *C*, and *D*. Further assume *B* and *C* each has 3 neighbors not already in our presumed cycle while *D* has only 2 such neighbors. In this scenario we would choose to go to vertex *D* and would never consider going to vertices *B* and *C*.

This method failed in the seven case.

METHOD 1 (modified):
While implementing method 1 we made a small typing error which resulted in us only throwing out ONE of the non-minimal neighbors from our choice list (as opposed to throwing them all out). This resulted in the five case arriving at 13 solutions while the 7 case's first solution was arrived at in approximately 9 minutes. While this was nice we ran this algorithm on the 9 case for about 60 hours and never arrived at a solution. We further could not demonstrate that finding a solution was guaranteed so we abandoned this approach.

METHOD 2:
In this method we tried to implement a heuristic argument which goes as follows:
If we come to a vertex which has TWO or more unvisited neighbors each with only 2 neighbors remaining (counting the one we are at) then we back up a step by "dead-ending" the vertex we are at (i.e. in our stack we say that it has no neighbors).

This is based on the scenario:



Say we are at *A*, if we go to *B* then eventually we must visit *D* and there is no way to leave *D*, so we would be done, so we abort early to save time. Of course neither *B* nor *D* are the start-end vertex as that has already been visited. The downside is we must check that neither *B* nor *D* is a neighbor of the start-end vertex. If one is the neighbor of the start-end vertex, then we go to the one that is not by removing the option of going to the one that is.

This seemed like a good idea but when implemented and run on the 7 case, the program finished in 1.5 minutes with the conclusion there were no Hamiltonian cycles. This may have been an error in the way we were testing the neighbors.

METHOD 2 (modified):

As oddities would have it when we were implementing method 2 we again made a typing error. This error went to the effect that:

IF we are on vertex *A* with neighbors *B* and *C* and "others" not already in our cycle
and *B* has only 1 neighbor remaining
and *C* has only 1 OR 2 neighbors remaining
and "others" all have more than 2 neighbors.
THEN we dead-end vertex *A*.
ELSE IF we are on vertex *A* with neighbors *B* and *C* and "others" not already in our cycle
and *B* has only 1 neighbor remaining
and *C* has more than 2 neighbors
and "others" all have more than 2 neighbors
THEN we force our choice to be *B* (we remove the option of going to any other neighbor).

This method for n = 3 found both cycles, for n = 5 it found 47 cycles (it missed 1) and for n = 7 it took only 9 SECONDS to arrive at the first cycle.
It would be hoped that such a dramatic improvement in time would carry over to the n = 9 case. Unfortunately this is not likely simply based on the number of possible combinations that still must be examined.
Consider the very worst numbers (the real ones are slightly better):

For the n = 3 case the number of possibilities are $2 * 1^5$,
for the n = 5 case the number of possibilities are $3 * 2^{19}$,
for the n = 7 case the number of possibilities are $4 * 3^{69}$,
for the n = 9 case the number of possibilities are $5 * 4^{251}$.

In the n = 9 case, even if we reduce the number of possibilities at each step from 4 to 2 we still are left with $5 * 2^{251}$ possibilities to examine. Only by the density of the Hamiltonian cycles could we hope to find one in any short time period.

There is still one method we have yet to try:

METHOD 3:

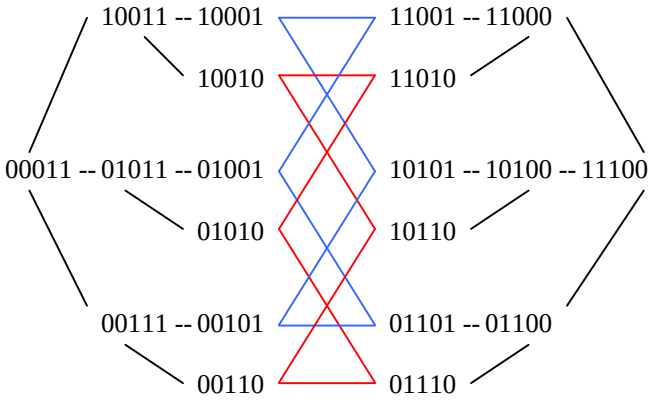
From the above two accidental errors leading to a significant time improvement it seems obvious that there is a way to selectively discard some choices at each step. It is highly likely that by studying the structure of the inner n-cube these choices could be readily determined. The unfortunate downside is that we still must face the fact that unless we are able to reduce the number of choices at each step down to exactly one we will still have a very large number of possibilities to examine. With this in mind we shall now list some of the properties of the inner n-cube.

1. The graph is clearly bipartite.
2. The graph is NOT chordal (i.e. there is not a chord across every cycle of length 6 or more).
3. The graph is NOT quasi-transitive directed, QTD, where a digraph D is QTD if for any triple *x*, *y*, *z*, of distinct vertices of D such that (*x*, *y*) and (*y*, *z*) are arcs of D there is at least one arc from *x* to *z* or from *z* to *x*).
4. The graph is NOT semi-complete, where a semi-complete digraph is a digraph with no pair of nonadjacent vertices.
5. In the inner n-cube, to get from a vertex to its inverse will take at least n steps.
6. The graph is almost self-similar repeating (like a fractal). The complication is that the repetitive structures are interlocked with one another in a way that is not one or two dimensional.

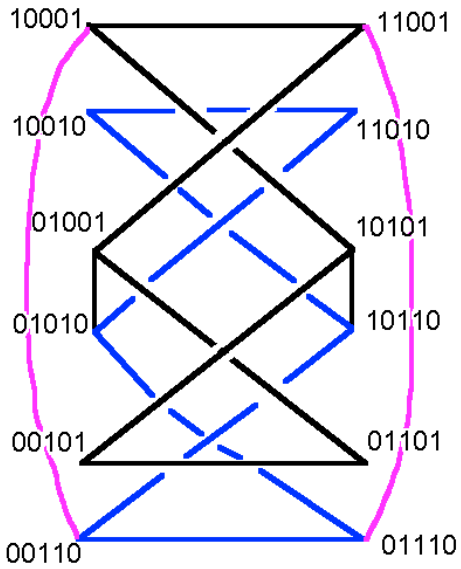
Property 1 has an enormous amount of literature written about it. Property 2, 3 and 4 were encountered in searching for a method to determine Hamiltonian cycles. It would seem that if the graph were chordal, semi-complete or quasi-transitive we could apply some other people's research. Unfortunately this is not the case. Property 5 is just a nice observation, for it is property 6 that seems likely to be the real key. To illustrate this repeating structure consider the following graphs:

For the n=3 case the inner n-cube is simply a cycle of 6 vertices:

For the n=5 case we have the structure:

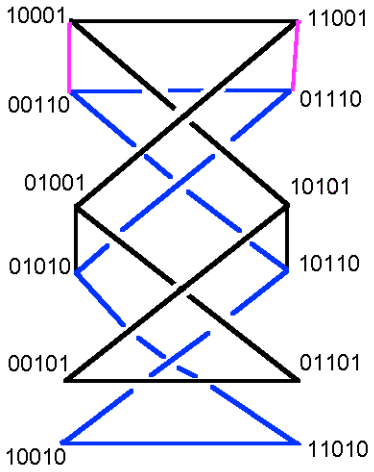


It might be worth noting there are at least twenty cycles of length 6 inside this structure. However it is much more interesting if we "contract" some items into a single edge and then redraw to obtain the following:

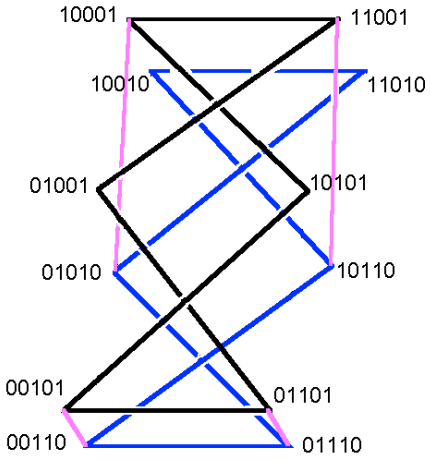


So how did we arrive at the new graph? It is rather simple.
First we make the path 01001 – 01011 – 01010 into a single edge between 01001 and 01010.Thus we also discard the edge from 01011 to 00011. We do a similar contraction on the path 10101 – 10100 – 10110. Having done this we then contract the path 10001 – 10011 – 00011 – 00111 – 00110 into a single edge between 10001 – 00110. We then do a similar contraction on the path 11001 – 11000 – 11100 – 01100 – 01110.

By doing this we have removed eight vertices from the graph and have abstracted a path between two vertices into a "complex" edge between two vertices. We now redraw the graph, flipping the bottom double-eight figure, to make it a little prettier:

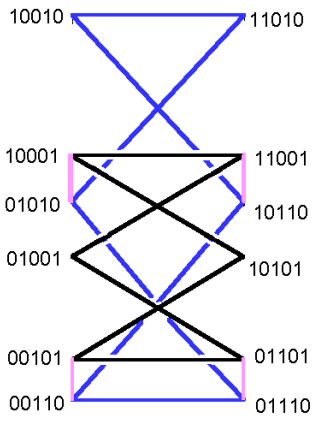


Notice that in this graph it is VERY easy to locate a Hamiltonian cycle – however, as the four "vertical" edges now represent edges containing points (i.e. complex edges) any valid Hamiltonian cycle we want to find must contain all four of these edges. In this particular contraction you cannot find such a Hamiltonian cycle. This does not, however, mean the method fails. In fact if we simply do a different contraction, such as that below, the method will succeed:



Note that:
1. edge 00101 – 00110 = path 00101 – 00111 – 00110
2. edge 01110 – 01101 = path 01110 – 01100 – 01101
3. edge 11001 – 10110 = path 11001 – 11000 – 11100 – 10100 – 10110
4. edge 01010 – 10001 = path 01010 – 01011 – 00011 – 10011 – 10001

Now if we make the graph a little prettier we have:



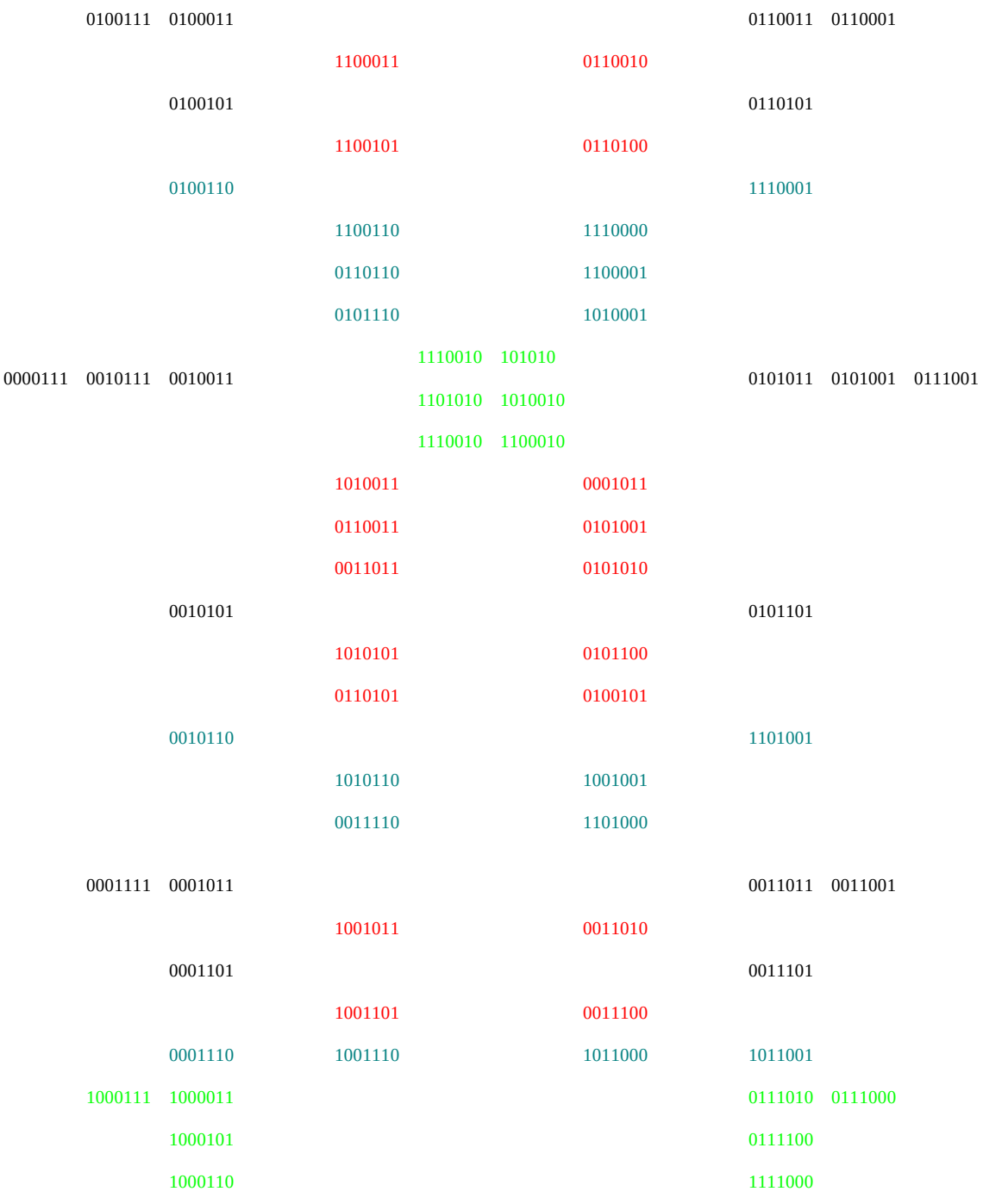
And we see that the cycle:
00101 – 00110 –
01110 – 01101 –
01001 –
11001 – 10110 –
10010 – 11010 –
01010 – 10001 –
10101

does indeed hit all the needed edges and expands to the cycle:

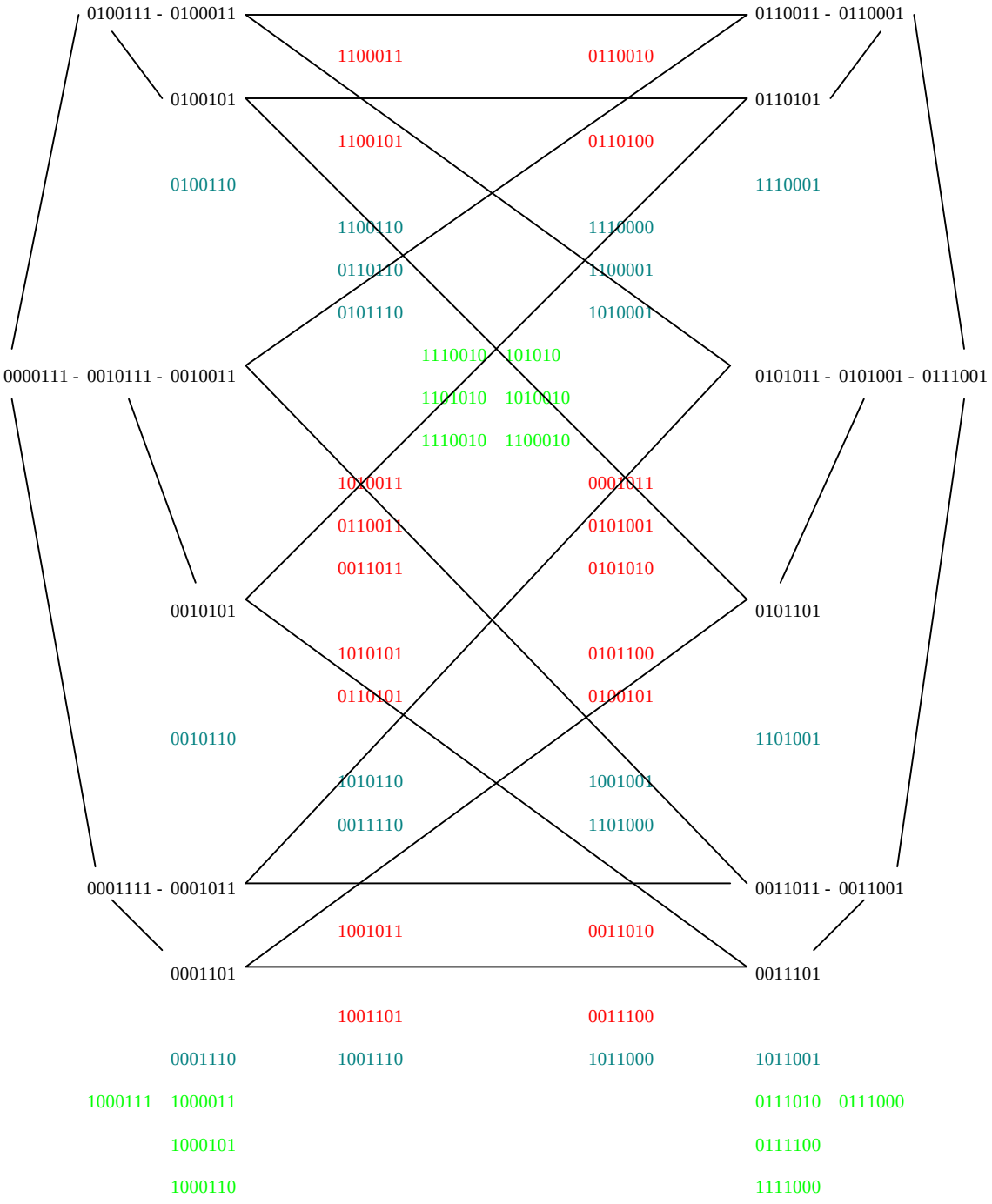
00101 – 00111 – 00110 –
01110 – 01100 – 01101 –
01001 –
11001 – 11000 – 11100 – 10100 – 10110 –
10010 – 11010 –
01010 – 01011 – 00011 – 10011 – 10001 –
10101

Something you might notice is that we now have a shape repeated that is “half” the size of the original shape. This may just be a coincidence, or it may be significant. Regardless the real problem to face is how to get the computer to perform contractions that will be successful as opposed to those contractions which fail.

The structure for n = 7 (lines omitted for clarity):



Notice that the black vertices form the same structure as the inner 5-cube. This effect is repeated throughout the structure. Also notice that the green vertices in the center of the structure only have neighbors in the “first shell” around them.



Note that while the above picture demonstrates the structure recurrence it does not lend itself immediately to illustrating a way to contract the graph to something “better.”

So what does this repetitive structure mean? Perhaps nothing, but maybe everything. It is suspected that the inner 7-cube can be compressed in a fashion similar to how the inner 5-cube was. Likewise it is suspected that the inner 9-cube can be compressed. If these suspicions are correct then we have solved the problem. Unfortunately we do not, yet, know how to get the computer to do the proper compressions necessary and drawing out the inner 9-cube to do it manually would be a rather tedious chore.

So in the mean time, it seems obvious that all the paths need not be searched. For example if the cycle went through 0000111 to 0100111 to 0100101 and you did an exhaustive search and found no cycles then when you backtrack and arrive at the option 0000111 to 0001111 to 0001101 you should not have to look down the path that next goes to 0101101 as that was an option you explored via symmetry because 0100101 also would have gone to 0101101. (Honest this makes sense if you look at the graph on the previous page). The key point being that the symmetry reduces the need to check this option. The problem is getting the computer to recognize this symmetry.

This structural symmetry technique might be expanded to simply exclude options of vertices which share a neighbor and are on a cycle with that neighbor. For example in the n = 5 case: is it really necessary if we start at 00011 to consider both 10011 and 00111?

These techniques may be exploited by careful construction of the adjacency matrix or by some clever coding trick (given the success of our errors it might just be best to randomly discard one of the choices at each step). Regardless, after studying the structure for over a month now, we have come to no definite way to simplify the problem down to just one choice at each step. Nor have we even created a method which would reduce the number of choices to one for a majority of the steps. This is unfortunate, but not unexpected.

For the time being we must step back from the problem and work on more “solvable” problems.

Should any further work be attempted on this problem it would be advised that before attempting to create a parallel algorithm to aid in the solution, that a sequential one be developed that is guaranteed to find a cycle. Once that algorithm exists it would seem that if it is based on a backtracking method then porting it into a parallel algorithm would require some derivative of a depth first graph searching technique.

As for future work that may be taken, perhaps exploring the algebraic representation of the graph would be worth performing. It seems that it could represent some form of discrete (abelian?) group. It might also be worth noting some of the effects the known cycles have. For example in the n=5 case it is interesting to see that all the cycles have the pattern that inverse elements’ distances from one another follow this pattern:
15, 13, 7, 5, 15, 13, 7, 5, …
Notice that 15 = 5 in the opposite direction of the cycle as does 13 = 7. Unfortunately the n = 7 case does not seem to follow the same concept, or if it does it is much more convoluted, and how this pattern would be determined for the n = 9 case would appear to be an equally difficult problem to solve.

It might also be useful to implement an algorithm which does not start at 000001111 nor at 111110000 as these just might be bad starting vertices for exhaustive measures. There might also be a way to create an algorithm which starts its search by using a given “input” path. Or perhaps it is possible to create a program to find all paths that are half the length of the cycle and try to splice them together.

Then again, perhaps backtracking is not the solution at all. Perhaps it would be better to spend time on finding a way to have the computer contract the larger graphs into the smaller ones.

Whatever the solution may be, we have not been able to determine it in the last two months.

- BMD

Bibliography

1. Bang-Jensen, J., M El Haddad, Y. Manoussakis and T. M. Przytycka, *Parallel Algorithms for the Hamiltonian Cycle and Hamiltonian Path Problems in Semicomplete Bipartite DiGraphs* from Algorithmica, 17: 67-87, Springer-Verlag New York Inc., 1997.
2. Bang-Jensen, Jorgen, and Gregory Gutin, *Vertex heaviest paths and cycles in quasi-transitive digraphs*, (unpublished ?) paper from the Department of Mathematics and Computer Science of Odense University, Denmark.
3. Bellman, Richard. Algorithms Graphs and Computers, Academic Press, New York, 1970. This book is Volume 62 in *Mathematics in Science and Engineering*.
4. Chambers, Lance. Practical Handbook of Genetic Algorithms: Applications, Volume I, CRC Press, New York, 1995.
5. Evans, James R. Optimization algorithms for networks and graphs – 2nd ed., rev. and expanded, Mercel Dekker, Inc., New York, 1992.
6. Hu, T. C. Combinatorial Algorithms, Addison Wesley Publishing Company, 1982.
7. Laufer, Henry B. Discrete mathematics and Applied Modern Algebra, Prindle, Weber and Schmidt (PWS), Boston, 1984.
8. Mackenzie, Philip D. and Quentin F. Stout, *Optimal parallel Construction of Hamiltonian Cycles and Spanning Tree in Random Graphs (Preliminary Version)*, from Proc. 5th ACM Symp. on Parallel Algorithms and Architectures, pp. 224-229, 1993.
9. Mott, Joe L. Discrete Mathematics for Computer Scientists and Mathematicians, Prentice-Hall, New Jersey, 1986.
10. Nijenhuis, Albert and Herbert S. Wilf. Combinatorial Algorithms, Academic Press, New York, 1975.
11. Reinelt, Gerard. Lecture Notes in Computer Science 840: The Traveling Salesman, Computation Solutions for TSP Applications, Springer-Verlag Berlin Heidelberg 1994.
12. Robinson, R.W. and N.C. Wormald, *Existence of long cycles in random cubic graphs*, Enumeration and Design, Academic Press, Toronto, pp. 251-270, 1981.
13. Rulan, Kevin Scott. *Polyhedral Solution to the Pickup and Delivery Problem*, Dissertation presented to the Sever Institute of Washington University at Saint Louis for partial fulfillment for the degree of Doctor of Science, August, 1995.
14. Vandegriend, Basil. *Finding Hamiltonian Cycles: Algorithms, Graphs and Performance*, Thesis submitted to the University of Alberta, Spring 1998.

Code Attachments:
Below is the code used to implement method 1 (modified)

```
////////////////////////////////////////////////////////////////////////
// FindCycleMinWay
//
// Identical to FindCycle but calls *MinOnly functions
////////////////////////////////////////////////////////////////////////
long CGraph::FindCycleMinWay()
{
  FILE *out_file;
  long A[GRAPH_MAXVERT+1];      // the cycle, coded by vertex index
  long stack[GRAPH_STACK_MAX+1];
  long index, k, m, nstk;
  long i2, count;
  long vert_index;
  long adj_ok;

  // Init the A and stack
  printf("Initializing variables...\n");
  for (index=0; index < GRAPH_MAXVERT+1; index++)
  {
    A[index] = 0;
  }
  for (index=0; index < GRAPH_STACK_MAX+1; index++)
  {
    stack[index] = 0;
  }

  // Guarantee our adjacency matrix is up to date
  printf("Creating Adjacency Matrix...\n");

  adj_ok = SetAdjMat();
  if (!adj_ok)
  {
    printf("ERROR (FindCycle):\n");
    printf("Unable to create adjacency matrix.\n");
    return 0;
  }

  out_file = fopen("ham.txt", "w");
  printf("Writing to ham.txt\n");

  printf("Beginning to look for Ham. Cycles...\n\n");
  printf("Using MinOnly routines.\n\n");
  fprintf(out_file, "\nHamHer, number of vertices = %ld\n", m_NumVerts);
  fprintf(out_file, "Using MinOnly routines.\n\n");
  PrintTime(out_file);
  printf("\n");
  fprintf(out_file, "\n");
  fprintf(out_file, "=======================================================\n");
```

```
  count = 0;        // count keeps track of how many ham. cycles found
  nstk = GRAPH_STACK_MAX;
  index = 0;
  // num_verts = length of expected cycles
  // index = 0 should set k = 1, m = 0 on first call to BackTrack
  // BackTrack(...) sets
  // index = 1 for successfully done,
  // index = 3 means  no more vectors of type sought
  // index = 2 for call HamCycle

while (index != 3)
  {
    BackTrackMinOnly(m_NumVerts, A, &index, &k, &m, stack, nstk);
    if (index == 2)
    {
      HamCycleMinOnly(m_NumVerts, A, k, &m, stack, nstk);
    }
    else if (index == 1)  // found a cycle, print it
    {
      count++;          // count = total number of cycles found
      printf("C%ld: ", count);
      fprintf(out_file, "C%ld: ", count);

      // Print out the vertex indices
      for(i2=1; i2<= m_NumVerts; i2++)
      {
        printf("%ld\t",A[i2]);
        fprintf(out_file, "%ld\t",A[i2]);
      }
      printf("\n");
      fprintf(out_file, "\n");

      // Print out the actual vertex identifying strings
      for(i2=1; i2<= m_NumVerts; i2++)
      {
        vert_index = A[i2];
        printf("%s\t", m_Vert[vert_index].m_AssocStr);
        fprintf(out_file, "%s\t", m_Vert[vert_index].m_AssocStr);
      }
      printf("\n");
      fprintf(out_file, "\n");
      printf("\n");
      fprintf(out_file, "\n");
      PrintTime(out_file);
      fprintf(out_file, "------------------------------------------------------------\n");
    }
  } // end while index !=3 (i.e. we are not done)

  printf("\nDone.\n\n");
  fprintf(out_file, "\nDone.\n\n");
  PrintTime(out_file);
  fprintf(out_file, "============================================================\n");
  printf("\n");
  printf("Total number of Hamiltonian cycles found: %ld\n", count);
  printf("Note: depending on how the graph was set up there might\n");
  printf("be only half that number (or fewer) which are 'distinct.'\n");
```

```
  printf("\n");

  fclose(out_file);
  return 1;
} // end FindCycleMinWay

////////////////////////////////////////////////////////////////////////
// BackTrackMinOnly
//
// Just like backtrack but we only choose the vertices with the
// MINIMUM number of nhbrs remaining.
// This may not find ALL cycles but it has a good chance of finding ONE.
//
// stop_len is the desired length of the a complete output vector
// A is the output vector
// On entry:
// state = 0 for just starting
// state = 2 for keep going     (shouldn't ever be 1 or 3 on entry)
//    BackTrack(...) sets
//    state = 1 for successfully done, (size of A equals stop_len)
//    state = 3 means  no more vectors of type sought
//    state = 2 for call HamCycle
// A_index is the length of a partially constructed vector:
//         a call to HamCycle() is a request for position A[A_index];
//         A_index is set by BackTrack()
// stack_index is the location of the last item on the stack;
//         it is changed by both BackTrack() and HamCycle()
// stack is a linear array, of maximal length nstk, whose appearance
//    at a typical step is:
// z z z z z z z n1 z z z z z n2 z z z z n3 z z...
// \_____________/    \________/
//     \/              \/
//  n1 candidates      n2 candidates
//  for position       for position      etc
//     A(1)             A(2)
//
// i.e. let the lists of candidates for A(1),...,A(A_index-1), A(A_index)
// be stored in stack[], each list followed by its length
//
// Return the value should = state's new value.
// (this was translated from Fortran code, so indices start at 1)
// (and code is UGLY!)
//
////////////////////////////////////////////////////////////////////////
long CGraph::BackTrackMinOnly(long stop_len, long A[GRAPH_MAXVERT+1],
                        long *state, long *A_index, long *stack_index,
                        long Stack[GRAPH_STACK_MAX+1], long nstk)
{
  long num_cand, done;
  long out_vert_index, cur_nhbr;
  long i;

  if (*state == 0)
  {
    *A_index = 1;
    *stack_index = 0;
```

```
  *state = 2;

  return 2;
}

done = 0;
while (!done)
{
  num_cand = Stack[*stack_index];
  *stack_index = *stack_index - 1;

  if (num_cand != 0)
  {
    done = 1;
  }
  else  // if (num_cand == 0)
  {
    // So we are going to pop off an element from A
    // by decrementing A_index.
    // In the process we need to update our edge counts
    out_vert_index = A[*A_index];
    for (i=1; i<= m_Vert[out_vert_index].m_Degree; i++)
    {
      cur_nhbr = m_Vert[out_vert_index].m_Nhbr[i];
      m_Vert[cur_nhbr].m_NumNhbrsLeft++;
    }

    *A_index = *A_index -1;
    if (*A_index == 0)
    {
      *state = 3;   // ran out of options, so fail
      return 3;
    }
  }
} // end while !done

// Now we pop off one of the elements and run with it.
// It is assumed that HamCycleMinOnly put only those
// vertices with the fewest remaining edges.
A[*A_index] = Stack[*stack_index];
Stack[*stack_index] = num_cand - 1;

// And we must update our edge counts according
// to the new element we just added into or A path-cycle.
out_vert_index = A[*A_index];
for (i=1; i<= m_Vert[out_vert_index].m_Degree; i++)
{
  cur_nhbr = m_Vert[out_vert_index].m_Nhbr[i];
  m_Vert[cur_nhbr].m_NumNhbrsLeft--;
}

if (*A_index == stop_len)   // Check if we are done
{
  *state = 1;
  return 1;
}
```

```
  *A_index = *A_index + 1;
  *state = 2;
  return 2;

} // end BackTrackMinOnly

//////////////////////////////////////////////////////////////////////////
// HamCycleMinOnly   (more code converted from Fortran)
//
// This function is similar to HamCycle, except we screen out some
// of the vertices we put on the candidate stack based on
// how many nhbrs they have remaining, the fewer, the better.
//
// Note: if we ever get to a point where all the (unvisited) nhbrs have
// exactly 2 nhbrs remaining, we know we are screwed. There is no further
// reason to follow this path. (Unless one of them is the start/end pt)
// We do NOT currenly check this.
//
// num_verts = number of vertices in graph G
// A => A(i) is the i-th vertex in the current circuit
// A_index = index of the next step on the current partial circuit
// stack_index = current length of stack i
// stack = candidates for steps 1 to k-1
// nstk = maximum length of stack
//
// m_AdjMat[][] => if (m_AdjMat[i][j] = 1) then edge (i, j) is in G
//
// Purpose: Find the candidates for k-th vertex in Ham. cycle
// store them in stack and update m accordingly
//
// I try to return what m is set to.
//
//////////////////////////////////////////////////////////////////////////
long CGraph::HamCycleMinOnly(long num_verts, long A[GRAPH_MAXVERT+1],
                        long A_index, long *stack_index,
                        long Stack[GRAPH_STACK_MAX+1], long nstk)
{
  long vert[GRAPH_MAXVERT+1];  // temp storage of what verts adj. to A[k-1]
  long i;                  // standard looping index
  long k1, a1, m1;
  long min_num_nhbrs, nhbr_vert;

  if (A_index == 1)
  {
    Stack[1] = 1;  // put vert 1 first
    Stack[2] = 1;  // then there is 1 element in that section of the stack
    *stack_index = 2;
    return 2;
  }

  // Find ALL the verts adjacent to A[A_index-1]
  k1 = A_index - 1;
  a1 = A[k1];
  for (i=1; i <= num_verts; i++)
  {
```

```
  vert[i] = m_AdjMat[a1][i];
}

// Make sure we remove all verts we have previously visited.
for (i=1; i<= k1; i++)
{
  m1 = A[i];
  vert[m1] = 0;   // 0 = false, 1 = true
}

// Now find those with the least number of nhbrs remaining
// a1 = A[A_index-1] = current vertex index
min_num_nhbrs = 99999;
for (i=1; i< m_Vert[a1].m_Degree; i++)
{
  nhbr_vert = m_Vert[a1].m_Nhbr[i];
  // if the nhbr is still in vert we see if its nhbrs are minimal
  if (vert[nhbr_vert])
  {
    if (m_Vert[nhbr_vert].m_NumNhbrsLeft < min_num_nhbrs)
    {
      min_num_nhbrs = m_Vert[nhbr_vert].m_NumNhbrsLeft;
    }
  }
  // if the nhbr is NOT minimal, throw it out of the viable candidate stack
  // Doing this check here would leave some "large" candidates
  // if (m_Vert[nhbr_vert].m_NumNhbrsLeft > min_num_nhbrs)
  //    vert[nhbr_vert] = 0;
} // end for
for (i=1; i< m_Vert[a1].m_Degree; i++)
{
  //nhbr_vert = m_Vert[a1].m_Nhbr[i];   // this line appeared to be missing
                                        // with addition of this line things fail
  if (m_Vert[nhbr_vert].m_NumNhbrsLeft > min_num_nhbrs)
  {
    vert[nhbr_vert] = 0;
  }
} // end for

m1 = *stack_index;
if (A_index != num_verts)
{
  for (i=1; i<= num_verts; i++)
  {
    if (vert[i] == 1)
    {
      m1 = m1 + 1;
      Stack[m1] = i;
    }
  } // end for
  Stack[m1+1] = m1 - (*stack_index);
  *stack_index = m1 + 1;
  return (m1 + 1);
}  // end if k != num_verts

for (i=1; i <= num_verts; i++)
```

```
  {
    if (vert[i])
    {
      if (!m_AdjMat[i][1])
      {
        Stack[m1+1] = m1 - (*stack_index);
        *stack_index = m1 + 1;
        return (m1+1);
      }
      *stack_index = *stack_index +2;
      Stack[*stack_index - 1] = i;
      Stack[*stack_index] = 1;
      return (*stack_index);
    }
  } // end for

  Stack[m1+1] = m1 - (*stack_index);
  *stack_index = m1 + 1;

  return (m1+1);

} // end HamCycleMinOnly
```

Below is the code used to implement method 2 (modified)

```
////////////////////////////////////////////////////////////////////////
// FindQuickShoot
//
// Similar to FindCycle()
// however we also implement:
// 1. we also should now have a check that if we come to a vertex who
//    has TWO or more unvisted nhbrs each with only 2 nhbrs remaining
//    (counting the one we are at) then we back up a step.
//    This is based on the scenario:
//
//    a ------ b  ------ c
//    |
//    |
//    d ------ e
//
//    Say we are at a, if we go to b then eventually we must
//    visit d and there is no way to leave d, so we would
//    be done, so we abort early to save time.
//    (of course neither b nor d = start/end vertex as that has
//     already been visited)
//    The downside is we must check that neither b nor d is a nhbr
//    of the start vertex. If one is the nhbr of start, then we
//    go to the one that is not (remove the option of going to the
//    one that is)
//
// It is in the quick routines we first introduce member variable
// m_StartCycle in function HamCycleQuick(...)
//
// A clever, well maintained, data structure can give you
// glimpses of the future.
//
////////////////////////////////////////////////////////////////////////
long CGraph::FindQuickShoot()
{
  FILE *out_file;
  long A[GRAPH_MAXVERT+1];      // the cycle, coded by vertex index
  long stack[GRAPH_STACK_MAX+1];
  long index, k, m, nstk;
  long i2, count;
  long vert_index;
  long adj_ok;

  // Init the A and stack
  printf("Initializing variables...\n");
  for (index=0; index < GRAPH_MAXVERT+1; index++)
  {
    A[index] = 0;
  }
  for (index=0; index < GRAPH_STACK_MAX+1; index++)
  {
    stack[index] = 0;
  }
```

```
// Guarantee our adjacency matrix is up to date
printf("Creating Adjacency Matrix...\n");

adj_ok = SetAdjMat();
if (!adj_ok)
{
  printf("ERROR (FindCycle):\n");
  printf("Unable to create adjacency matrix.\n");
  return 0;
}

out_file = fopen("ham.txt", "w");
printf("Writing to ham.txt\n");

printf("Beginning to look for Ham. Cycles...\n\n");
printf("Using QuickShoot routines.\n\n");
fprintf(out_file, "\nHamHer, number of vertices = %ld\n", m_NumVerts);
fprintf(out_file, "Using QuickShoot routines.\n\n");
PrintTime(out_file);
printf("\n");
fprintf(out_file, "\n");
fprintf(out_file, "=======================================================\n");

count = 0;        // count keeps track of how many ham. cycles found

nstk = GRAPH_STACK_MAX;
index = 0;

// num_verts = length of expected cycles
// index = 0 should set k = 1, m = 0 on first call to BackTrack

// BackTrack(...) sets
// index = 1 for successfully done,
// index = 3 means  no more vectors of type sought
// index = 2 for call HamCycle

// Choose our first starting vertex:
m_StartCycle = 1;

while (index != 3)
{
  BackTrackQuick(m_NumVerts, A, &index, &k, &m, stack, nstk);
  if (index == 2)
  {
    HamCycleQuick(m_NumVerts, A, k, &m, stack, nstk);
  }
  else if (index == 1)  // found a cycle, print it
  {
    count++;          // count = total number of cycles found
    printf("C%ld: ", count);
    fprintf(out_file, "C%ld: ", count);

    // Print out the vertex indices
    for(i2=1; i2<= m_NumVerts; i2++)
    {
      printf("%ld\t",A[i2]);
```

```
          fprintf(out_file, "%ld\t",A[i2]);
        }
        printf("\n");
        fprintf(out_file, "\n");

        // Print out the actual vertex identifying strings
        for(i2=1; i2<= m_NumVerts; i2++)
        {
          vert_index = A[i2];
          printf("%s\t", m_Vert[vert_index].m_AssocStr);
          fprintf(out_file, "%s\t", m_Vert[vert_index].m_AssocStr);
        }
        printf("\n");
        fprintf(out_file, "\n");
        printf("\n");
        fprintf(out_file, "\n");
        PrintTime(out_file);
        fprintf(out_file, "------------------------------------------------------------\n");
      }
    } // end while index !=3 (i.e. we are not done)

    printf("\nDone.\n\n");
    fprintf(out_file, "\nDone.\n\n");
    PrintTime(out_file);
    fprintf(out_file, "============================================================\n");
    printf("\n");
    printf("Total number of Hamiltonian cycles found: %ld\n", count);
    printf("Note: depending on how the graph was set up there might\n");
    printf("be only half that number (or fewer) which are 'distinct.'\n");
    printf("\n");

    fclose(out_file);
    return 1;
  } // end FindCycleQuick

  ////////////////////////////////////////////////////////////////////////
  // BackTrackQuick
  //
  // Just like backtrack but we only choose the vertices with the
  // MINIMUM number of nhbrs remaining.
  // This may not find ALL cycles but it has a good chance of finding ONE.
  //
  // stop_len is the desired length of the a complete output vector
  // A is the output vector
  // On entry:
  // state = 0 for just starting
  // state = 2 for keep going     (shouldn't ever be 1 or 3 on entry)
  //    BackTrack(...) sets
  //    state = 1 for successfully done, (size of A equals stop_len)
  //    state = 3 means  no more vectors of type sought
  //    state = 2 for call HamCycle
  // A_index is the length of a partially constructed vector:
  //         a call to HamCycle() is a request for position A[A_index];
  //         A_index is set by BackTrack()
  // stack_index is the location of the last item on the stack;
  //         it is changed by both BackTrack() and HamCycle()
```

```
// stack is a linear array, of maximal length nstk, whose appearance
//    at a typical step is:
// z z z z z z z n1 z z z z z n2 z z z z n3 z z...
// \_____________/    \________/
//     \/              \/
//  n1 candidates      n2 candidates
//  for position       for position      etc
//     A(1)             A(2)
//
// i.e. let the lists of candidates for A(1),...,A(A_index-1), A(A_index)
// be stored in stack[], each list followed by its length
//
// Return the value should = state's new value.
// (this was translated from Fortran code, so indices start at 1)
// (and code is UGLY!)
//
////////////////////////////////////////////////////////////////////////
long CGraph::BackTrackQuick(long stop_len, long A[GRAPH_MAXVERT+1],
                            long *state, long *A_index, long *stack_index,
                            long Stack[GRAPH_STACK_MAX+1], long nstk)
{
  long num_cand, done;
  long out_vert_index, cur_nhbr;
  long i;

  if (*state == 0)
  {
    *A_index = 1;
    *stack_index = 0;
    *state = 2;

    return 2;
  }

  done = 0;
  while (!done)
  {
    num_cand = Stack[*stack_index];
    *stack_index = *stack_index - 1;

    if (num_cand != 0)
    {
      done = 1;
    }
    else  // if (num_cand == 0)
    {
      // So we are going to pop off an element from A
      // by decrementing A_index.
      // In the process we need to update our edge counts
      out_vert_index = A[*A_index];
      for (i=1; i<= m_Vert[out_vert_index].m_Degree; i++)
      {
        cur_nhbr = m_Vert[out_vert_index].m_Nhbr[i];
        m_Vert[cur_nhbr].m_NumNhbrsLeft++;
      }
```

```
        *A_index = *A_index -1;
        if (*A_index == 0)
        {
          *state = 3;   // ran out of options, so fail
          return 3;
        }
      }
    } // end while !done

    // Now we pop off one of the elements and run with it.
    // It is assumed that HamCycleMinOnly put only those
    // vertices with the fewest remaining edges.
    A[*A_index] = Stack[*stack_index];
    Stack[*stack_index] = num_cand - 1;

    // And we must update our edge counts according
    // to the new element we just added into or A path-cycle.
    out_vert_index = A[*A_index];
    for (i=1; i<= m_Vert[out_vert_index].m_Degree; i++)
    {
      cur_nhbr = m_Vert[out_vert_index].m_Nhbr[i];
      m_Vert[cur_nhbr].m_NumNhbrsLeft--;
    }

    if (*A_index == stop_len)   // Check if we are done
    {
      *state = 1;
      return 1;
    }

    *A_index = *A_index + 1;
    *state = 2;
    return 2;

} // end BackTrackQuick

////////////////////////////////////////////////////////////////////////
// HamCycleQuick   (more code converted from Fortran)
//
// This function is similar to HamCycle, except we screen out some
// of the vertices we put on the candidate stack based on
// how many nhbrs they have remaining, the fewer, the better.
//
// Note: if we ever get to a point where all the (unvisited) nhbrs have
// exactly 2 nhbrs remaining, we know we are screwed. There is no further
// reason to follow this path. (Unless one of them is the start/end pt)
// We do NOT currenly check this.
//
// num_verts = number of vertices in graph G
// A => A(i) is the i-th vertex in the current circuit
// A_index = index of the next step on the current partial circuit
// stack_index = current length of stack i
// stack = candidates for steps 1 to k-1
// nstk = maximum length of stack
//
// m_AdjMat[][] => if (m_AdjMat[i][j] = 1) then edge (i, j) is in G
```

```
//
// Purpose: Find the candidates for k-th vertex in Ham. cycle
// store them in stack and update m accordingly
//
// I try to return what m is set to.
//
////////////////////////////////////////////////////////////////////////
long CGraph::HamCycleQuick(long num_verts, long A[GRAPH_MAXVERT+1],
                           long A_index, long *stack_index,
                           long Stack[GRAPH_STACK_MAX+1], long nstk)
{
  long vert[GRAPH_MAXVERT+1];  // temp storage of what verts adj. to A[k-1]
  long i;                      // standard looping index
  long k1, a1, m1;
  long min_num_nhbrs, nhbr_vert;
  long number_with_2;

  if (A_index == 1)
  {
    Stack[1] = m_StartCycle;  // put vert m_StartCycle first
    Stack[2] = 1;  // then there is 1 element in that section of the stack
    *stack_index = 2;
    return 2;
  }

  // Find ALL the verts adjacent to A[A_index-1]
  k1 = A_index - 1;
  a1 = A[k1];
  for (i=1; i <= num_verts; i++)
  {
    vert[i] = m_AdjMat[a1][i];
  }

  // Make sure we remove all verts we have previously visited.
  for (i=1; i<= k1; i++)
  {
    m1 = A[i];
    vert[m1] = 0;   // 0 = false, 1 = true
  }

  // Now find those with the least number of nhbrs remaining
  // a1 = A[A_index-1] = current vertex index
  min_num_nhbrs = 99999;
  number_with_2 = 0;    // we shall count how many nhbrs are viable candidates
for (i=1; i< m_Vert[a1].m_Degree; i++)
  {
    nhbr_vert = m_Vert[a1].m_Nhbr[i];
    // if the nhbr is still in vert we see if its nhbrs are minimal
    if (vert[nhbr_vert])
    {
      if (m_Vert[nhbr_vert].m_NumNhbrsLeft < min_num_nhbrs)
      {
        min_num_nhbrs = m_Vert[nhbr_vert].m_NumNhbrsLeft;
        // The below if was also some bizarre coding error
        // which allowed the first solution to the 7 case
        // to be found in about 7 seconds
```

```
        // With it we did miss 1 of the 48 solutions to n = 5
        // Probably because we don't check that the nhbr with
        // 1 is also nhbr of start vertex
        // This if should be technically correct, though
        // it makes the variable names misleading.
        if (min_num_nhbrs == 2)
        {
          number_with_2++;
        }
      }
      if (m_Vert[nhbr_vert].m_NumNhbrsLeft == 1)
      {
        if (!m_AdjMat[nhbr_vert][m_StartCycle]) // if nhbr is not also a nhbr of start vert
        {
          number_with_2++;                  // we count it
        }
      }
    }  // end if vert. was originally still viable
  } // end for
  if ((min_num_nhbrs == 1) && (number_with_2 > 1))   // we dead end this vertex now
  {
    for (i=1; i <= num_verts; i++)
    {
      vert[i] = 0;
    }
  }
  else if (min_num_nhbrs == 1)  // we MUST go to that nhbr, remove all other possibles
  {
    for (i=1; i< m_Vert[a1].m_Degree; i++)
    {
      nhbr_vert = m_Vert[a1].m_Nhbr[i];
      if (m_Vert[nhbr_vert].m_NumNhbrsLeft > min_num_nhbrs) // > 2
      {
        vert[nhbr_vert] = 0;
      }
    } // end for
  }

  m1 = *stack_index;
  if (A_index != num_verts)
  {
    for (i=1; i<= num_verts; i++)
    {
      if (vert[i] == 1)
      {
        m1 = m1 + 1;
        Stack[m1] = i;
      }
    } // end for
    Stack[m1+1] = m1 - (*stack_index);
    *stack_index = m1 + 1;
    return (m1 + 1);
  }  // end if k != num_verts

  for (i=1; i <= num_verts; i++)
  {
```

```
    if (vert[i])
    {
      if (!m_AdjMat[i][1])
      {
        Stack[m1+1] = m1 - (*stack_index);
        *stack_index = m1 + 1;
        return (m1+1);
      }
      *stack_index = *stack_index +2;
      Stack[*stack_index - 1] = i;
      Stack[*stack_index] = 1;
      return (*stack_index);
    }
  } // end for

  Stack[m1+1] = m1 - (*stack_index);
  *stack_index = m1 + 1;

  return (m1+1);

} // end HamCycleQuick
```

c’est fini.

- BMD