An Efficient Algorithm for Hamilton Cycle Based on the Enlarged Rotation-Extension Technique

Lizhi Du

Abstract—Algorithm studies on the Hamilton cycle are mainly based on the Rotation-Extension method developed by Posa. However, due to the deficiency of Posa’s method, all these products are only efficient for much denser graphs or sparse but regular graphs. By many years’ study, we developed the “Enlarged Rotation-Extension” technique which utterly changed and expanded the Posa’s original one and can surmount its deficiency. Based on this technique, our algorithm can quickly calculate randomly produced un-directed graphs with up to ten thousand vertices on personal computer, no matter dense or sparse, the correctness is one hundred percent. We also calculated the data of Hamilton cycles on a famous web site and we still get one hundred percent correctness.

Index Terms—Computer Algorithm, Computational Complexity, Hamilton Cycle, Hamilton Path, Polynomial Time

I. INTRODUCTION

A Hamilton path is a path between two vertices of a graph that visits each vertex exactly once. A Hamilton path that is also a cycle is called a Hamilton cycle.

Finding Hamilton cycles(paths) in simple undirected graphs is a classical NP Complete problem, known to be difficult both theoretically and computationally, so we can not expect to find polynomial time algorithms that always succeed, unless P=NP. 

For this algorithm, the challenging job is to develop an efficient random algorithm for all general graphs(in this paper, we only concern undirected graphs), i.e., this random algorithm can work for all kinds of graphs successfully with high probability. For this purpose, the main problem is: can this be possible?

For finding Hamilton cycles(paths), we would mention the famous rotation-extension technique, developed by Posa[8]. In fact, most of random algorithms are based on the rotation-extension technique. Due to this technique’s immanent deficiency, all these random algorithms can only work for dense graphs. So if we can overcome the rotation-extension technique’s immanent deficiency, it is possible for us to get an efficient random algorithm for all general graphs.

We develop a method which we call the “enlarged rotation-extension” technique. This technique can overcome Posa’s deficiency. Our method contains all advantages of the rotation-extension technique but utterly enlarges its functions. By a lot of test, we confirm that our method is useful for both dense graphs and sparse graphs.

Based on this technique, we get an efficient algorithm for finding a Hamilton cycle(path) in an undirected graph. This algorithm works very well for all kinds of undirected graphs. A program on this algorithm has been tested over a hundred million times for graphs whose vertex number is between 100 to 10000, no fails.

II. WHAT IS THE ENLARGED ROTATION-EXTENSION TECHNIQUE?

Suppose we have a path \( P = x_0x_1\ldots x_k \) in a graph \( G \) and we wish to find a path of length \( k+1 \). If \( x_0 \) or \( x_k \) has a neighbor not in \( P \), then we can extend \( P \) by adding the neighbor. If not, suppose \( x_k \) has a neighbor \( x_i \), where \( 0 <= i <= k-2 \). If \( i=0 \) and \( G \) is connected, then there is an edge \( e = (x_i, w) \) joining the cycle \( x_0x_1\ldots x_kx_0 \) to the rest of the graph, and so the path \( wx_0x_1\ldots x_kx_0\ldots x_{i-1} \) has length \( k+1 \). This is called a cycle extension. If \( i=0 \), then we construct the path \( x_0x_1\ldots x_kx_{i-1}x_{i+1} \) of length \( k \) with a different endpoint \( x_{i+1} \) and look for further extensions. This is called a rotation, or a simple transform. This is Posa’s Rotation-Extension technique.

This method’s main deficiency is: it does the rotation or extension at the fixed place, in order to always fulfill the rotation or extension condition, the graph must have dense edges. So this technique is not useful for sparse graphs. We change it as following:

First, let the \( n \) vertices sit side by side to form a cyclic sequence, we call this “broad cycle”. In this cycle, some two consecutive vertices may not be adjacent, we call this point “break point”. Apparently, a break point is constituted by two vertices.

If a broad cycle has only one break point, we call it “one break point cycle”; If a broad cycle has two break points, we call it “two break points cycle”. And so on.

Each time, cut a segment we call a subsequence of a broad cycle a segment) from a break point, insert the segment in some place of the broad cycle. How to cut and insert? The rule is: make the number of new break points the least. Also we must design some way to prevent circulation or repeating job, and to limit the calculating times.

We can see that our technique contains all advantages of the rotation-extension technique and the rotation-extension is only one special case of ours. So, we call ours the “enlarged rotation-extension” technique. We will describe our algorithms later which are mainly based on this technique.

Manuscript received November 18, 2013; revised December 23, 2013.

Lizhi Du, Author is with the College of Computer Science and Technology, Wuhan University of Science and Technology, Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System, Wuhan 430065, China (phone: 86-13554171855; e-mail: edw95@yahoo.com).
III. ALGORITHM

Definition 1

For an undirected graph G with n vertices, let \( x_1, x_2, \ldots, x_n \) denote the n vertices. A broad cycle is defined as a cyclic sequence \( x_1, x_2, \ldots, x_n, x_1 \) of the vertices of G where every pair \( x_i, x_{i+1} \) may or may not be adjacent in G. We call a pair \( (x_i, x_{i+1}) \) (including \( (x_n, x_1) \)) of non-adjacent vertices a break point. So the number of break points is between 0 and n for a broad cycle. Apparently, a break point is constituted by two vertices. A general point(a break point or a connecting point) is constituted by two vertices, we say that the general point contains these two vertices.

A segment: in a broad cycle, we call any subsequence of a broad cycle a “segment”. So, for a broad cycle, any part between two general points is a segment, and there are \( n(n-1)/2 \) possible different break points and connecting points. For an undirected graph with n vertices, the number of all possible different break points and connecting points is \( n(n-1)/2 \). A connecting point or a break point is also called “a general point”. A general point (a break point or a connecting point) is constituted by two vertices, we say that the general point contains these two vertices.

For a broad cycle, cut a segment at some place of this broad cycle a “segment”. So, for a broad cycle, any part between two general points is a segment, and there are \( n(n-1)/2 \) different segments for this broad cycle.

For a broad cycle, cut a segment at some place of this broad cycle, insert the segment between two consecutive vertices in some other place of the broad cycle. Before inserting, we may do the rotation and extension on the segment and then insert it. Note: for this rotation or extension, it is not necessary that the two end vertices are adjacent. We call this operation a “cut and insert”. The main operation in our algorithm is the “cut and insert”, now we explain it: Let \( x_1, \ldots, x_n, x_1 \) denote a broad cycle, let \( s = x_1, x_{i+1}, \ldots, x_{i+r} \) be a subsequence of this broad cycle (i.e., a segment), and let \( j \) be an index such that \( x_j \) and \( x_{j+1} \) are not in \( s \). A broad cycle \( C \) is obtained by “cutting \( s \) and inserting it into \( x_j \) and \( x_{j+1} \)” if either \( C = x_j, x_{i+1}, x_{i+2}, \ldots, x_{i+r}, x_{j+2}, \ldots, x_n, x_1, \ldots, x_{j-1}, x_j \) or \( C = x_j, x_{i+1}, x_{i+2}, \ldots, x_{i+r}, \ldots, x_j, x_{j+1}, x_{j+2}, \ldots, x_n, x_1, \ldots, x_{j-1}, x_j \) (addition is meant modulo n). Also, before inserting, we may do the rotation and extension on the segment and then insert it.

At each step of the algorithm, we need to choose a break point as the “main break point” as explained later in the algorithm.

Algorithm 1 FindHTcycle

Input: An adjacency matrix \( A \) to denote an undirected graph. A matrix \( B \) to record all main break points. An array \( C \) to record the broad cycle. An integer \( n \), the number of vertices.

Output: A Hamilton cycle(path)(if exists), or “No Hamilton cycle(path)” message.

```
void FindHTCycle(matrix A, matrix B, int[] C, int n)
// if an edge between vertex i and j, then A[i,j]=1,
//else A[i,j]=0
1. int[] C = new int[n];
2. for(int i=0; i<n; i++)
3. C[i]=i; //0~n-1 denote the n vertices, the C[0] //and C[n-1] are consecutive, so, C[1] denotes the
4. broad cycle.
5. For each break point in C, add an edge between the two vertices(say k, l) of the break point, set
7. Check if there are some edges in C whose value in matrix A is 2, if no, output a message:
   “success get the hamilton cycle” and store C which is the Hamilton cycle then end the
   procedure. If yes, delete one of these edges to produce a break point, set its value in A to 0.
   we call this “main break point”, initialize all values of matrix B to 0.
8. Set the value of the two vertices of the main break point in matrix B to 1. From the main
   break point in cycle C, cut a segment, insert the segment in some place of the cycle C. How to
   cut and insert? The rule is: make the number of new break points the least, and one new break
   point must be different from all former main break points(this new one as the new main break
   point, using matrix B to record all main break points, this guarantees our algorithm’s
do not repeat the algorithm still is polynomial(i.e., we only need to repeat
   our algorithm’s polynomial). Note: “make the number of new break points the least” means:
   only for all the possible “cut and inserts” to choose the best one, the “real least”. Also, at least
   one of the two end vertices of the cut segment must be adjacent to one vertex of the inserting point.
   Notes: when calculating the number of new break points for getting the least,
   if more than one case have the same least number, choice one any of them.
   Notes: in the algorithm, if we always keep any two vertices as neighbors, we can get a Hamilton path
   between these two vertices.
   Now, we describe our Algorithm 2. As stated above, if a broad cycle has only one break
   point, we call it “one break point cycle”; If a broad cycle has two break points, we call it “two break points cycle”. And so on. Our algorithm 2 only handles these two kinds of broad cycles.
   At first, a broad cycle may have k break points(0<=k<n), we add k edges so that no break point in
   the path, so this cycle is a Hamilton cycle. We remember all the added k edges, each time, we delete one such edge, to get an
   “one break point cycle”, our algorithm only leads this one break point cycle to a Hamilton cycle(if exists). If our
   algorithm is polynomial, after deleting all the added k edges, the algorithm still is polynomial i.e., we only need to repeat
   our algorithm at most k times). So, now, our algorithm’s job only is to transform an one break point cycle to a Hamilton
   cycle(if exists).
```
Algorithm 2 FindHCycle2
Input: An adjacency matrix A to denote an undirected graph. A matrix B to record all main break points. An one break point cycle. An broad cycle array C to record all the new broad cycles at current step. An integer n, the number of vertices.
Output: A Hamilton cycle(if exists), or “No Hamilton cycle” message.

Our algorithm’s main job is: cut and insert. Now we use an example to discuss it again.

0…. x y…..a b….c*d….n-1 (1)

(1) is a broad cycle from vertex 0 to n-1, also vertex 0 is adjacent to n-1. Vertex x and vertex y are consecutive, so are vertex a and b, vertex c and d. “….…” denotes many other vertices. Vertex c is not adjacent to d, let c*d is the main break point, we cut “b….c” from the broad cycle, insert it between x and y, then we can get a new broad cycle. This is the “cut and insert”.

Our algorithm includes three functions: Do0(), Do1() and Do2().

When we cut “b….c”, insert it between x and y, if vertex a adjacent to d, also, vertex b adjacent to x and vertex c adjacent to y, or, vertex b adjacent to y and vertex c adjacent to x, the result is

0….x b….c y…..a d….n-1 or 0….x c….b y…..a d….n-1.

Function Do0() does the above job.

When we cut “b….c”, insert it between x and y, if vertex a adjacent to d, also, vertex b adjacent to x and vertex c is not adjacent to y and c*y was not as the main break point before(then c*y as the new main break point now), or, vertex c adjacent to y and vertex b is not adjacent to x and b*x was not as the main break point before(then b*x as the new main break point), or, vertex b adjacent to y and vertex c is not adjacent to x and c*x was not as the main break point before(then c*x as the new main break point), or, vertex b adjacent to y and vertex c is not adjacent to x and b*y was not as the main break point before(then b*y as the new main break point).

Or, if vertex a is not adjacent to d and a*d was not as the main break point before(then a*d as the new main break point), also, vertex b adjacent to x and vertex c adjacent to y, or, vertex c adjacent to x and vertex b adjacent to y.

Function Do1() does the above job.

When we cut “b….c”, insert it between x and y, if vertex a is not adjacent to d and a*d was not as the main break point before(then a*d as the new main break point), also, vertex b adjacent to x and vertex c adjacent to y, or, vertex c adjacent to x and vertex b adjacent to y.

Function Do2() does the above job.

Because the number of main break points is polynomial and it cannot repeat, the algorithm is polynomial.

By the way, we also can use B[i][j] to remember the main break points in one break point cycles, use B[i][j] to do so for two break point cycles, sometimes this way is much quicker.

IV. EXPERIMENT DATA

We have three kinds of undirected graphs to test our algorithms. Programs on these algorithms have been designed in VC++.

First, we use the random graphs. To discuss random graphs, we must first introduce the probability spaces(or models) of random graphs. All graphs are undirected. The model we focus on is G(n,p). The model G(n,p) (sometimes called the independently model) consists of all graphs with vertex set [n]={1,2…n} in which the edges are chosen independently with probability p, where 0<p<1. We consider a random graph G composed of a Hamilton path on n labeled vertices and some random edges that “hide” the path, the random edges are produced as above[7][8]. We carefully choose the probability p to make the graph is hard to calculate.

Without loss of generality, for an undirected graph with N nodes, node number is 0,1,2…N-1, the algorithm calculates Hamilton path from node 0 to node N-1. The input data is randomly produced undirected graphs. In order to test the program, each graph includes a randomly produced Hamilton path which the program does not know. We have tested the program of Algorithm 1 over one hundred million inputs, no one fails. The data is as Table I (computer: HP PC, CPU: Intel 1G, Memory:1G):
When randomly producing the un-directed graphs, we try to make the graphs as hard as possible to calculate. A lot of tests show that when its average vertex degree is about between 2.5 to 4, the graph is hardest to calculate (even its biggest vertex degree is 3, this problem still is NP-Complete [21]). With the vertex number much greater, the hardest average vertex degree may increase very slowly. So, our random graphs are mainly with 2.5 to 4 average degree.

Secondly, we get the test data from the famous web site, the famous standard test bed on http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. On this web, there are 9 files for Hamilton cycles. Our program of Algorithm 1 can calculate all the 9 files, very easy, very fast. The calculating time for each is just like the time on the above table 1. For each file, we can quickly get a hamilton cycle which is different from the web owner’s, because each one has more than one hamilton cycle.

Now, we discuss what kinds of graphs are hard to calculate. By a lot of test, we find that when the graph’s average vertex degree is between 2.5 to 4, it is hard to get the Hamilton cycle. When the average vertex degree is over 4, our Algorithm 1 can always get the result quickly. But, we only need to make a little change to the Algorithm 1, it still can always get the result quickly even when the graph’s average vertex degree is between 2.5 to 4. The change is: each time when we cut a segment, before we insert the segment in some point, do the cycle extension for the segment, then try to insert it.

By a long time experiment and study, we find another kind of graphs which are very hard to calculate. Even using our changed Algorithm 1, we still can not calculate them quickly with high probability. This kind of graphs is: first we carefully design a hard 3SAT, then transform the 3SAT to Hamilton cycle problem (an undirected graph). For 3SAT, when its clauses is about 4 to 4.5 times of its variables, it is the hardest to calculate. So we design the 3SAT according to this rule. In this way, we get the graphs which are very hard to calculate for Hamilton cycle. Note: if only for hamilton path (not cycle, also not path between two vertices, only any hamilton path), our changed Algorithm 1 still can calculate this kind of graphs with high probability. We first explain how to transform 3SAT to Hamilton cycle of an undirected graph. We use two vertices to denote a variable, and use 13 vertices to denote a clause. We got this way after a long time research and we think this is the best way to transform 3SAT to Hamilton cycle directly. See M.R. GAREY [21] for another way to do this job. We have the same principle and logic with GAREY, but our way is the cheapest one (because his way only for a special kind of graphs).

This is the third kind of graphs. Our Algorithm 2 can calculate these graphs very well. In order to guarantee the high correctness, we also make a little change to the

Algorithm 2: each time when we cut a segment, before we insert the segment in some point, try to do the cycle extension for the segment, then try to insert it. Note, all graphs which the Algorithm 1 can calculate, the Algorithm 2 still can calculate. The experiment data is as Table II (computer: HP PC, CPU: Intel 2G, Memory:2G):

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Calculation number of different inputs</th>
<th>Success number of times</th>
<th>Average run time</th>
<th>Longest run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>1000000000</td>
<td>100000000</td>
<td>0.07 seconds</td>
<td>0.1 second</td>
</tr>
<tr>
<td>10000</td>
<td>10000000</td>
<td>100000000</td>
<td>0.014 seconds</td>
<td>0.01 second</td>
</tr>
<tr>
<td>100</td>
<td>10000000</td>
<td>100000000</td>
<td>48 seconds</td>
<td>192 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Calculation number of different inputs</th>
<th>Success number of times</th>
<th>Average run time</th>
<th>Longest run time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1250</td>
<td>100000000</td>
<td>10000000</td>
<td>9 minutes</td>
<td>21 minutes</td>
</tr>
<tr>
<td>2000</td>
<td>100000000</td>
<td>10000000</td>
<td>58 minutes</td>
<td>1 hour 52 minutes</td>
</tr>
</tbody>
</table>

REFERENCES

[22] M.R. GAREY etc. The planar Hamiltonian Circuit problem is NP-Complete. SIAM J. COMPUT. Vol5, No. 4, 1976