TCS-TR-A-14-77

TCS Technical Report

An Efficient Algorithm for Enumerating Eulerian Paths

by

Muhammad Kholilurrohman and Shin-ichi Minato

Division of Computer Science

Report Series A

October 3, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682
Fax: +81-011-706-7682

An Efficient Algorithm for Enumerating Eulerian Paths

Muhammad Kholilurrohman[1] and Shin-ichi Minato[1,2]

[1] Graduate School of Information Science and Technology,
Hokkaido University, Sapporo, Japan
[2] ERATO MINATO Discrete Structure Manipulation System Project,
Japan Science and Technology Agency, Sapporo, Japan

Abstract. Although a mathematical formula for counting the number of Eulerian circles in a directed graph is already known [1, 2], no formula is yet known for enumerating such circles if the graph is an undirected one. In computer science, the latter problem is known to be in #P-complete [3], enumerating the solutions of such problem is known to be hard. In this paper, an efficient algorithm to enumerate all the Eulerian paths in an undirected graph, both simple graph and multigraph, is proposed.

Keywords: enumeration, Eulerian circle, Eulerian path, graph theory, undirected graph

1 Introduction

In graph theory, Eulerian path (or Eulerian trail) is a path in a graph which visits every edge exactly once. If the path starts and ends on the same vertex, then it is called an Eulerian circle. Eulerian path problem dates back to 1736, when for the first time Leonhard Euler discuss in his paper [4] the famous *Seven Bridges of Königsberg problem*. He proved the necessary condition for the existence of Eulerian circle that all the vertices of the connected graph must have an even degree, while the necessity condition was proved later by Carl Hierholzer [5] in 1873. Hereafter, if we mention about graph, then we always refer to a connected graph.

There are some algorithms for constructing Eulerian path, one of them is called Hierholzer's algorithm, which takes linear time [6]. Thus, finding one Eulerian path is considerably easy, but enumerating all of the Eulerian paths of a given graph remains hard.

If the graph is directed, then we can count the Eulerian circles in it using a mathematical formula discovered by de Bruijn, van Aardenne-Ehrenfest, Smith and Tutte. The formula is called BEST theorem [1] [2], named after the people who discovered it.

Theorem 1 (BEST theorem). *Let D be a directed graph with vertices $V = \{v_1, v_2, \cdots, v_m\}$, and suppose that the in-degree (which is the same as out-degree) of a vertex v_i is d_i. Let $t(D)$ be the number of directed spanning trees rooted at*

any fixed vertex v in D. The BEST theorem states that the number of Eulerian circles $Eul(D)$ in D can be stated mathematically as:

$$Eul(D) = t(D) \prod_{i=1}^{m} (d_i - 1)! \tag{1}$$

Despite the number of Eulerian cycles in a directed graph can be expressed in a simple mathematical equation, there is no any mathematical expression if the graph is undirected. In this paper, we introduce an efficient algorithm for this problem, which is much faster than naive brute force algorithm in many cases. The idea of our algorithm is based on the algorithm introduced by Knuth, which is called *simpath* [7] (exercise 225 in 7.1.4). Generally speaking, *simpath* is an algorithm that constructs a *directed acyclic graph (DAG)* which later can be transformed efficiently into a *Zero-suppressed Binary Decision Diagram (ZDD)* [8], a kind of data structure that can be used to represent a set of all *simple paths*[3] between two given vertices of a given graph. *Simpath* is so powerful that a ZDD representing 64528039343270018963357185158482118 simple paths connecting two opposite corners of a 13×13 grid graph is constructed within just a few seconds. Like *simpath*, our algorithm makes good use of DAG to represent all Eulerian paths. In this paper we show that counting paths on such DAG can be efficiently done without having to count the path one by one.

This paper is organized as follows. Algorithm to enumerate simple paths (*simpath*) and the data structure (ZDD) used in the algorithm is introduced in Section 2. The details of our algorithm are discussed in Section 3. Next in Section 4, we present the experimental results of this algorithm on some graphs. Lastly, we conclude the paper in Section 5.

2 Enumerating Simple Paths

In this section we start our discussion from the naive backtrack algorithm for enumerating all simple paths between given vertices s and t, then the explanation about ZDD, and at last about *simpath* algorithm.

2.1 Naive Method

Let $G = (V, E)$ be an undirected graph having vertices $V = \{v_1, v_2, \cdots, v_m\}$ and edges $E = \{e_1, e_2, \cdots, e_n\}$. Let the source and the destination of the paths as vertices s and t, where $s, t \in V$. We want to enumerate all simple paths between s and t, which is called s-t path.

One of the simplest ways to enumerate such paths is by using a naive backtrack algorithm. But, using this method is infeasible in term of time, because the time needed for backtracking is proportional to the number of the s-t paths. Even for a relatively small graph, backtrack algorithm is considered to be impractical.

[3] Simple path is a path which does not pass through a vertex more than once.

2.2 Zero-suppressed Binary Decision Diagram (ZDD)

Zero-suppressed Binary Decision Diagram (ZDD) [8] is a labeled directed acyclic graph obtained by reducing a binary decision tree graph. Just like binary decision tree, each node (drawn as circle) in ZDD has two edges, namely *0-edge* and *1-edge*, which connect a node to its two *child nodes*, *0-child* and *1-child*, respectively. *0-edge* of a node e_i is denoted as $e_i = 0$, and *1-edge* is denoted as $e_i = 1$. In this paper, *0-edge* is drawn using a dashed-line, while *1-edge* is drawn using a solid line. By traversing the nodes of a ZDD from its root node[4], eventually we will reach either *0-terminal* or *1-terminal*, which is respectively expressed as a squared box labeled by 0 or 1. Here we always refer to an ordered ZDD, hence for the input variables $e_1, e_2, \cdots, e_n$, they will appear ordered regardless of the path we traverse from the root to the terminal node (although maybe there will be some missing variables due to the ZDD reduction rules).

ZDD can be obtained from a binary decision tree graph by recursively applying these two rules until no further reduction is possible.

1. Eliminate all the nodes whose *1-edge* points to the *0-terminal* node. Then connect the edge from its parent node to the other sub-graph directly.
2. Share all equivalent sub-graphs.

ZDD is proved to be ideal for compactly expressing family of sets [8]. For example, Figure 1 (d) shows how family of sets $F = \{\{e_1, e_3\}, \{e_2\}\}$ is expressed compactly using ZDD compared to its corresponding binary decision tree graph.

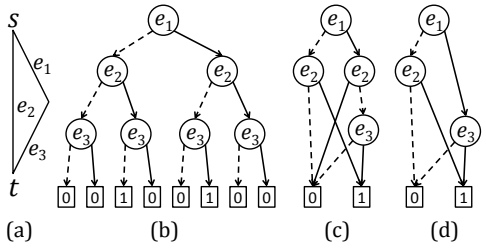

Fig. 1. A triangle graph, a binary decision tree, a DAG, and a ZDD

This family of sets $F = \{\{e_1, e_3\}, \{e_2\}\}$ is representing all s-t paths of the triangle graph in (a). In binary decision tree (b), the set $\{e_1, e_3\}$ is represented by the path with $e_1 = 1, e_2 = 0, e_3 = 1$, which is called as *1-path* because it is connected to *1-terminal*. The path representing $\{e_2\}$ is also connected to the *1-terminal*, and the others are connected to *0-terminal*. Thus, the number of s-t paths correspond to the number of *1-paths* in the binary decision tree, which is also equivalent with the number of *1-paths* in the ZDD.

[4] Root node is the node at the top of a ZDD.

The number of *1-paths* in a ZDD can be enumerated as follows. First, we assign 0 to *0-terminal* node and 1 to *1-terminal* node. Next, for the others, the value of a node is the same as the sum of values of its *child nodes*. Finally, the value of the ZDD's root node is the same as the number of s-t paths of the given graph. Thus, the time needed for enumerating s-t paths is proportional to the number of nodes in ZDD, not to the number of s-t paths. If the compression rate of ZDD is high, the enumeration time can be very fast.

2.3 *Simpath*

A ZDD can be obtained by reducing a binary decision tree using ZDD reduction rules. But, creating ZDD by this way is time consuming, because for n input variables, the size of binary decision tree is already $O(2^n)$.

To speed up the process, *simpath* does not create a binary decision tree, instead it creates a DAG which then can be effectively reduced into a ZDD. The construction of the DAG is done in *breadth-first search* manner, pruning out the branch that will not produce any s-t path as early as possible. For example in Figure 1 (a), if both edges e_1 and e_2 are not selected, no s-t path is able to be constructed, so the branch when $e_1 = 0$ (shown as *0-edge* of the node e_1) and $e_2 = 0$ can be directly appointed to *0-terminal* node without having to concern about the assignment of e_3. *Simpath* also merges two nodes with the same index (the same level in the DAG) when it is known that the two branches will produce the same output value for any combinations of the rest of the input variables. Thanks to branch pruning and node merging, the constructed DAG is usually much more compact than the corresponding binary decision tree and can be efficiently reduced into a ZDD.

Branch pruning and node merging is carried out based on the information of the vertices. Each node in the DAG represents current selected edges. These selected edges form path fragments, and each vertex will have one of the following three states which need to be remembered by the node:

1. not included in any path fragments, or
2. intermediate point of a path fragment, or
3. endpoint of a path fragment.

In a computer memory, we can use an array to express the state of each vertex. In his book, Knuth calls this array as *mate* array, which is a mapping from V to $V \cup \{0\}$.

$$mate[v] = \begin{cases} v & \text{if vertex } v \text{ is untouched so far} \\ 0 & \text{if vertex } v \text{ is touched by exactly two edges} \\ u & \text{if vertex } u \text{ and } v \text{ are endpoints of a path fragment} \end{cases}$$

At first, each vertex is assigned to have a *mate* value equal to itself ($mate[v] = v$).

When an edge is selected, we need to update the *mate* value of some vertices[5]. Suppose $mate[m] = \hat{m}$ and $mate[j] = \hat{j}$. If edge j-m is selected, we need to assign

[5] At most the *mates* of 4 vertices need to be updated when an edge is selected.

$mate[m] \leftarrow 0$, $mate[j] \leftarrow 0$, $mate[\hat{m}] \leftarrow \hat{j}$, $mate[\hat{j}] \leftarrow \hat{m}$, in exactly this order. Figure 2 explains what this assignment is doing.

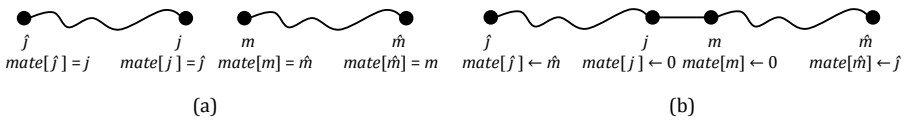

Fig. 2. Updating Mate

A set of selected edges is accepted when the vertices s and t are connected and no other path fragment exists, and is rejected when:

1. an edge is added to an intermediate point of a path fragment, or
2. it is known that s and t cannot be connected by current edges selection, or
3. a path connecting s and t is formed and some other path fragments remain.

Actually, each node does not have to remember the state of all vertices. All we need is the information of the vertices which shall be called *frontier*, the set of vertices contained in both processed and unprocessed edge. For example in Figure 3, if we let the thin solid line be the unprocessed edges, thick solid line be the selected edges, and dashed line be the edges not selected, then the *frontier* is $\{v_5, v_6, v_7\}$.

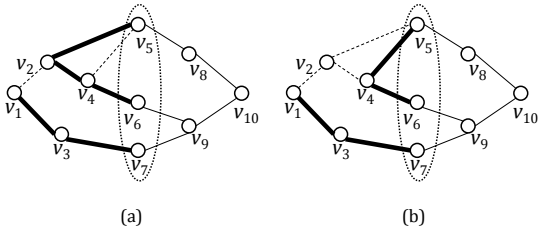

Fig. 3. Two path fragments which can be merged

At this stage, further selection of the unprocessed edges that makes Figure 3 (a) has an s-t path will also makes (b) has an s-t path. If we look closer, we know that both (a) and (b) have the same *mate* values for each vertex in the *frontier*. So, we do not need to process (b) further, instead we can merge the node representing (b) to the node representing (a).

The algorithm 1 bellow is the pseudocode for *simpath*. The *CheckTerminal* function is the part where the decision is made, whether we need to proceed to the next edge selection, or we have to appoint the current DAG branch to *0-terminal*, or we can appoint the DAG branch to *1-terminal*. Line 10 of the code is executed when there is already an equivalent node (node having the same *mate*) created in the current level of DAG, thus the node merging happens.

Algorithm 1: Simpath [7]

```
input  : An undirected graph, s, and t
output: ZDD representing all simple paths
1  N_1 ← {node_root};
2  N_i ← ∅ for i = 2, ··· , n + 1; // N_i holds nodes in level i
3  for i ← 1 to n do // process nodes in level i
4      foreach node ∈ N_i do
5          foreach x ∈ {0, 1} do // processing 0-edge and 1-edge
6              node' ← CheckTerminal(node, i, x);
               // CheckTerminal returns 0-terminal, 1-terminal, or nil
7              if node' = nil then
8                  Create a new node and set it to node';
9                  if there exists node'' ∈ N_i s.t. node'' is equivalent to node'
                   then
10                     node' ← node''
11                 else
12                     N_{i+1} ← N_{i+1} ∪ {node'}
13             Create x-edge to connect node and node'.
14 Using ZDD reduction rule, reduce the DAG into ZDD;
```

3 Enumerating Eulerian Paths

In this section, we discuss about the basic idea of the proposed algorithm, the use of *mate*, and the idea of node merging.

We distinguish two similar Eulerian paths having different direction. Hence, the Eulerian cycle $v_1 \to v_2 \to v_3 \to v_1$ is counted separately form $v_1 \to v_3 \to v_2 \to v_1$. Notice that the problem of enumerating Eulerian cycles can be transformed into enumerating Eulerian paths by adding an additional edge. For example, adding an edge $v_m - v_{m+1}$ to the graph having vertices $V = \{v_1, v_2, \cdots, v_m\}$, then the problem of enumerating Eulerian cycle starting from v_m back to v_m is now become the problem of enumerating Eulerian paths from v_m to v_{m+1}.

3.1 Basic Idea

Assume there are x_i edges connected to vertex v_i. We want to make pairing of edges, so that every edge in this vertex has a pair. The number of combination of such edge pairing[6] is:

$$f(x_i) = \binom{x_i}{2} \cdot \binom{x_i - 2}{2} \cdots \binom{2}{2}. \tag{2}$$

In Subsection 2.2, we used a binary decision tree for expressing all possible combination of edge selection, and connected the set of edge selection that forms

[6] Assign $x_i \leftarrow x_i + 1$ for vertex s and t, because their degree are odd.

an *s-t* path to *1-terminal* node. Using similar idea, suppose we replace the binary decision DAG with a "multi-decision" DAG whose the number of child nodes in level i is $f(x_i)$, each expressing one of the combination of edge pairing in vertex v_i as described above. Because we are considering all combinations of any possible edge matching in all vertices, we can be sure that the entire Eulerian paths are represented in the DAG. The node labelled $v_1, v_2, \cdots, v_m$ will appear during our traversal from the root node of the DAG, each node represents one possible edge pairing in the corresponding vertex. If the overall edges pairing in these nodes are not forming an Eulerian path, we connect the traversed path of the DAG to *0-terminal* and otherwise to *1-terminal*.

Actually, the way we construct the DAG is slightly different from what has been explained above. Instead of the vertices, we process the edges one by one. This enables us to use the idea of *mate* array to hold the information of each vertex. In *simpath*, each vertex has exactly one *mate* value, because each vertex can be passed through by at most one path fragment. The Eulerian path is different; one vertex can be used by multiple path fragments. To accommodate this, we need to modify the *mate* array so that each vertex will be able to keep track on the path fragments currently using it. Thus, we will need to use array of *"lists of vertices"*[7] to express the state of each vertex. If we denote $deg(v_i)$ to be the degree of vertex v_i, then the maximal size of the $mate[v_i]$ for each vertex v_i will be[8]:

$$\frac{deg(v_i)}{2}. \tag{3}$$

At first, when there is still no any path fragment connected to vertex v_i, the size of $mate[v_i]$ is 0. Then, there are two choices if an edge is added to v_i:

1. connect to the endpoint of a path fragment that is already in there, or
2. create a new path fragment with endpoint at this vertex. In this case, increment the size of $mate[v_i]$, for accommodating the new path fragment. Choice 2 is only available if the number in Equation 3 is not reached yet.

In Figure 4 (a), vertex v has four edges e_1, e_2, e_3 and e_4. Suppose we process the edges in that order. For the root node, there is still nothing connected to vertex v, so we just put edge e_1 as the node 1 in (b) as a new path fragment. Next, there are two possibilities when we are adding edge v_2, whether to connect this edge to the already existing edge e_1, or we do not connect to any of the path fragment already in the vertex. When we are expanding node 3, we cannot make a new path fragment because the maximal number of Equation 3 is already reached, so we just connect the edge e_3 to the existing path fragments. The time when we finish processing all edges connected to a vertex, then we get the all possible edge matching in that vertex is already expressed in the DAG. In our algorithm, we do not necessarily process all the edges connected to a vertex at once. Sometimes we leave some edges unprocessed, and instead process another edge not directly connected to this vertex.

[7] More practically, we are using an array of a resizable array.
[8] Again, we need to make $deg(v_i) \leftarrow deg(v_i) + 1$ for vertices s and t.

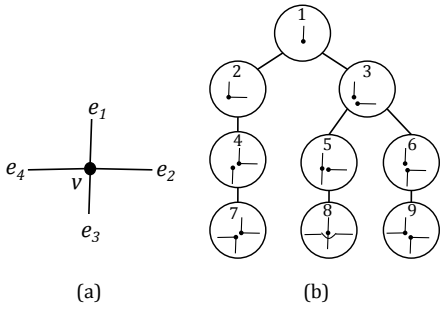

Fig. 4. Edge matching on a vertex

3.2 *Mate* Update and Node Merging

As in *simpath*, our algorithm works by processing the edges one by one. Each edge has two vertices, at each step we consider edge matching in these two vertices. If there are k path fragments connected to p and l path fragments connected to q, then $k \times l$ combinations of edge matching have to be considered. In addition, if there is still a chance to add a new path fragment to the vertex p and q, we have to consider in overall $(k+1) \times (l+1)$ cases.

But there is good news here; we can reduce this number into smaller cases by removing the duplicates, treating $mate[v]$ as a *weighted set* instead of a *list of vertices*. Let us write $mate[p]$ as $\{\alpha_1, \alpha_2, \cdots, \alpha_g\}$ and $mate[q]$ as $\{\beta_1, \beta_2, \cdots, \beta_h\}$, $g \leq k$ and $h \leq l$. Now, the cases can be reduced into at most $(g+1) \times (h+1)$. To implement $mate[p]$ we need three operators, namely $inc(\alpha_i)$ to increment, $dec(\alpha_i)$ to decrement, and $w(\alpha_i)$ to know the weight of α_i. For the sake of simplicity, we will write $p.operator(\alpha_i)$ instead of $mate[p].operator(\alpha_i)$.

Updating *mate* when α_i and β_j is connected[9] is done in the similar way as in *simpath*, except that now we are dealing with *weighted set*, which makes us have to see case by case according to which category do α_i and β_j belong. These categories are:

1. $\alpha_i = q$ or $\beta_j = p$ (i.e. p and q are connected by a path fragment).
2. $\alpha_i = p$ or $\beta_j = q$ (i.e. there is a path fragment/self-loop connecting p back to p or q back to q).
3. $\alpha_i = x$ or $\beta_j = y$, where $x, y \neq \{p, q\}, x, y \in V$ (i.e. there is a path fragment connecting p to x or q to y).
4. a newly added *mate* (in *simpath* we initialize $mate[v] = v$, but here initialization is not needed).

For example if both α_i and β_j belong to category 1, *mate* update is done as follows. Suppose $mate[p] = \{\alpha_1\} = \{q\}, (p.w(\alpha_1) = 3)$ and $mate[q] = \{\beta_1\} = \{p\}, (q.w(\beta_1) = 3)$, which is illustrated in Figure 5. Because the weight of each *mate* is 3, there are three path fragments connecting p to q. The dashed line in

[9] Simply ignore if α_i or β_j equals 0, which represents body of a path fragment.

the left figure shows the $3 \times 3 = 9$ possibilities of connecting vertex p to q using edge $p-q$. But, 3 cases will end up with a cycle being formed. This leaves us with the remaining $3 \times 3 - 3 = 6$ cases as shown in the right figure, which correspond to the number of *child nodes* need to be created. Joining any two different path fragments here will result the same, $mate[p] = \{0, q\}, (p.w(0) = 1, p.w(q) = 2)$ and $mate[q] = \{0, p\}, (q.w(0) = 1, q.w(p) = 2)$. Hence, the *mate* of these 6 *child nodes* will be the same. In conclussion, if α_i and β_j from category 1 are going to be connected, we need to create $p.w(\alpha_i)^2 - p.w(\alpha_i)$ child nodes each having *mate* as follow. Copy the *mate* of the parent's node, then assign $p.dec(\alpha_i)$, $q.dec(\beta_j)$, $p.inc(0)$, $q.inc(0)$.

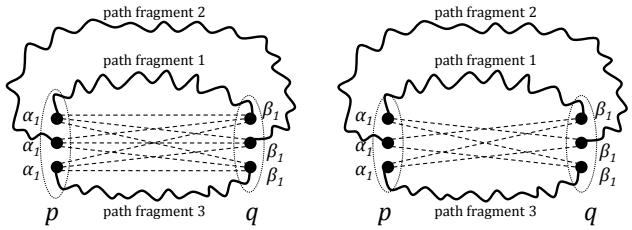

Fig. 5. Connecting two path fragments

Table 1 bellow summarizes all cases. At first copy the *mate* of the parent's node. Then, update this *mate* according to the assignment in the "Update rule" column. "Number of Child Nodes" column shows how many child nodes having identical *mate* need to be created, the weight in this column is based on *mate* of the parent's node. Shortly we will know that these child nodes can be merged, so that actually we can just create one child node.

Node merging in this algorithm is done based on the same idea used in *simpath*. Two nodes from the same level in the DAG can be merged if the *mates* of their vertices in *frontier* are the same. The only difference is that this time the *mate* of a vertex is considered as a *weighted set*. Therefore, we need that the *mate* of each vertex is sorted before comparison. The algorithm for enumerating Eulerian paths is also similar to Algorithm 1 except at line 5 and 13, because instead of having only two child nodes, now each node can have multiple child nodes. Also, we do not reduce the DAG, thus line 14 is not needed.

4 Experimental Results

In this section, we present the results of our experiment. The algorithm was implemented using C++11 and run on a machine with 4 GB memory Intel® Core™i3-2330M CPU @ 2.20GHz × 4.

At first, we present the result for graphs having multiple edges and compare the results with the corresponding simple graph. Then, for the Aztec diamond graph [9], we present the results up to $n = 8$.

Table 1. The Rules in Updating the Mate

No.	Category		Update Rule	Number of Child Nodes
	α	β		
1	1	1	$p.dec(q), p.inc(0), q.dec(p), q.inc(0)$	$p.w(q)^2 - p.w(q)$
2	1	2	$p.dec(q), p.inc(0), q.dec(p), q.inc(0)$	$p.w(q) \times q.w(q)$
3	1	3	$p.dec(q), p.inc(0), q.dec(p), q.inc(0)$	$p.w(q) \times q.w(y)$
4	1	4	$p.dec(q), p.inc(0), q.dec(p), 2 \times q.inc(q)$	$p.w(q)$
5	2	2	$2 \times p.dec(p), p.inc(0), p.inc(q),$ $2 \times q.dec(q), q.inc(0), q.inc(p)$	$p.w(p) \times q.w(q)$
6	2	3	$2 \times p.dec(p), p.inc(0), p.inc(y),$ $q.dec(y), q.inc(0), y.dec(q), y.inc(p)$	$p.w(p) \times q.w(y)$
7	2	4	$2 \times p.dec(p), p.inc(0), p.inc(q), q.inc(p)$	$p.w(p)$
8	3	3	$x.dec(p), x.inc(y), p.dec(x), p.inc(0),$ $q.dec(y), q.inc(0), y.dec(q), y.inc(x)$	$p.w(x) \times q.w(y)$
9	3	4	$x.dec(p), x.inc(q), p.dec(x), p.inc(0), q.inc(x)$	$p.w(x)$
10	4	4	$p.inc(q), q.inc(p)$	1

4.1 Multigraph

The proposed algorithm in this paper is relatively fast for processing multigraph[10] compared to its "corresponding" simple graph. For example, let the left graph in Figure 6 be $a(n)$ and the right one be $b(n)$. Let n be an odd number. We want to enumerate all Eulerian paths starting from vertex 1. The results for some n are summarized in Table 2.

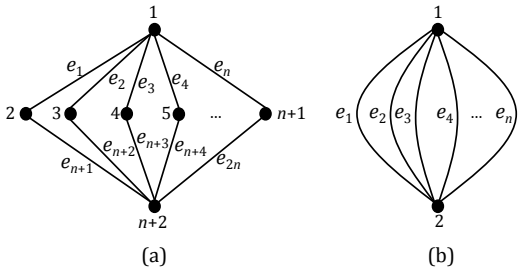

Fig. 6. Simple Graph vs Multigraph

It turns out that the nodes created for counting Eulerian paths in $a(n)$ and $b(n)$ are greatly different. For $b(13)$, the 6227020800 Eulerian paths are stored in the DAG having merely 131 nodes compared to $a(13)$ which needs 529397 nodes. This shows that node merging when processing $b(n)$ is done effectively, which also makes the processing time of $b(n)$ much faster. We also compare our algorithm with a backtrack based algorithm. Notice that the number of Eulerian

[10] Here, a multigraph is a graph with multiple edge, but without *self-loop*.

Table 2. Simple Graph vs Multigraph

n	$a(n)$			$b(n)$			solutions
	proposed algorithm		backtrack	proposed algorithm		backtrack	
	nodes	time	time	nodes	time	time	
3	16	0.006962	0.000973	8	0.003770	0.000385	6
5	72	0.023053	0.015367	16	0.008759	0.005399	120
7	463	0.033030	0.388146	30	0.018849	0.096984	5040
9	3940	0.131531	30.007690	52	0.025878	5.796462	362880
11	41774	1.370419	3612.151051	85	0.023074	638.443941	39916800
13	529397	20.117619	time out	131	0.025434	time out	6227020800
15	-	time out	time out	194	0.030318	time out	1307674368000
31	-	time out	time out	1786	0.160600	time out	8.22284×10^{33}
51	-	time out	time out	9776	1.222678	time out	1.55112×10^{66}
101	-	time out	time out	118014	29.145220	time out	9.42595×10^{159}

paths of these kind of graphs are actually $n!$, same as the number of possible permutation of $e_1, e_2, \cdots, e_n$.

4.2 Aztec Diamond

The Aztec diamond in this paper is a bit different from the usual[11] Aztec diamond. The Aztec diamond here as shown in Figure 7 is the same as the graph described by Audibert in his book [9]. Starting from the lower vertex of the leftmost edge, i.e. vertex number 1, we want to count the number of Eulerian cycles coming back to this vertex. Using backtrack algorithm, we could only count the Eulerian cycles up to $n = 3$, which took 150.041790 seconds. For $n = 4$, the running time is already more than 1 hour, so we stopped the program. On the contrary, using our proposed algorithm we could count the Eulerian paths up to $n = 8$. The detailed results are presented in Table 3.

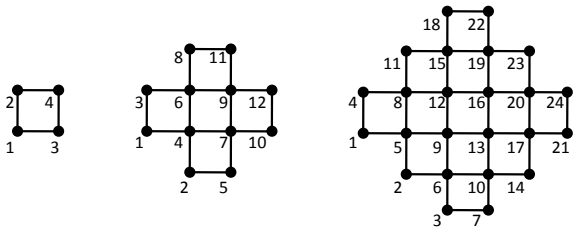

Fig. 7. Aztec Diamond for $n = 1, 2$, and 3

[11] The general term of Aztec diamond refers to a similar graph but with the leftmost, rightmost, uppermost, and lowermost edges are 2.

Table 3. The Result for Aztec Diamond Using Proposed Algorithm

n	nodes	time	solutions
1	8	0.000228	2
2	40	0.015503	80
3	286	0.032317	264320
4	2164	0.071024	67131225600
5	17271	0.500763	1282298454848135168
6	148224	6.153548	182395883547404421922439168 0
7	1382302	73.527219	192178269775153104174170778660103782400
8	14083862	942.330957	1495157006436041186484738405257449073460914460033024

5 Conclusion

We have explained the algorithm for enumerating Eulerian paths in a undirected graph, which can be used for both simple graph and multigraph. The algorithm is much faster compared to a naive backtrack algorithm in many cases. Also, treating *mate* of a vertex as a *weighted set* makes the computation in the algorithm efficient, removing the need to repeat the same operation over and over. Unfortunately, there is a side effect of treating *mate* of a vertex as a *weighted set*; the indexing of the Eulerian paths becomes difficult, which will we leave as future work.

References

1. van Aardenne-Ehrenfest, T., de Bruijn, N. G.: Circuits and trees in oriented linear graphs. Simon Stevin 28, 203–217 (1951)
2. Tutte, W. T., Smith, C. A. B.: On unicursal pahts in a network of degree 4. American Mathematical Monthly 48, 233–237 (1941)
3. Brightwell, G. R., Winkler, Peter: Note on Counting Eulerian Circuits. CDAM Research Report LSE-CDAM-2004-12 (2004)
4. Euler, Leonhard: Solutio Problematis ad Geometriam Situs Pertinentis. Comment. Academiae Sci. I. Petropolitanae 8, 128–140 (1736)
5. Hierholzer, Carl: Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren. Mathematische Annalen 6 (1), 30–32 (1873)
6. Fleischner, Herbert: X.1 Algorithms for Eulerian Trails. Eulerian Graphs and Related Topics: Part 1, Volume 2, Annals of Discrete Mathematics 50, Elsevier pp. X.1-13 (1991)
7. Knuth, D. E.: The Art of Computer Programming, Volume 4A, Combinatorial Algorithm, Part 1. 1st.Addison-Weasley Professional (2011)
8. Minato, Shin-ichi: Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems. In Proc. of 30th ACM/IEEE Design Automation Conference (DAC'93), pp. 272-277 (1993)
9. Audibert, Pierre: Mathematics for Informatics and Computer Science. ISTE Ltd. pp. 832 (2010)