𝕋ℂ𝕊-𝕋ℝ-𝔹-15-10

𝕋ℂ𝕊 Technical Report

Master's Thesis: A Fast Algorithm for Enumerating Eulerian Paths

by

Muhammad Kholilurrohman

Division of Computer Science

Report Series B

February 26, 2015

Hokkaido University
Graduate School of
Information Science and Technology

Email: kholil@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

Master’s Thesis
平成26年度　修士学位論文

A Fast Algorithm for Enumerating Eulerian Paths

オイラー路を高速に数え上げるアルゴリズム

Muhammad Kholilurrohman

ムハマド　ホリルロハマン

Laboratory for Algorithmics, Division of Computer Science
Graduate School of Information Science and Technology
Hokkaido University

北海道大学　大学院情報科学研究科
コンピュータサイエンス専攻　アルゴリズム研究室

February 26, 2015

Contents

Summary

Although a mathematical formula for counting the number of Eulerian cycles in a directed graph is already known [1, 2], no formula is yet known for enumerating such cycles 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 thesis, an efficient algorithm to enumerate all the Eulerian cycles (paths) of an undirected graph, both simple graph and multigraph, is proposed.

Chapter 1

Introduction

In graph theory, Eulerian path (or Eulerian trail) is a path in a graph that visits every edge exactly once. If the path starts and ends on the same vertex, then it is called an Eulerian cycle. Graph possessing an Eulerian path is called a *Semi-Eulerian* graph, while the graph possessing an Eulerian cycle is called an *Eulerian* graph.

Eulerian cycle 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 an Eulerian cycle that all vertices of the connected graph must have an even degree, while the sufficient condition was proved later by Carl Hierholzer [5] in 1873. Their proofs are extendable to show that the necessary and the sufficient conditions for a graph having an Eulerian path is that there are exactly two vertices with odd degree. As a result, removing one edge from an Eulerian graph will make the graph being a Semi-Eulerian, and connecting the two vertices with odd degree in a Semi-Eulerian graph will then form an Eulerian graph.

There are some algorithms for constructing an 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 and proved to be in #P-complete [3], a complexity class in counting problems that is as hard as NP-complete class in decision problems[1].

In this thesis, we introduce our approach to solve the problem of Eulerian paths enumeration. We show that our proposed algorithm is efficient for practical use, and is able to enumerate all Eulerian paths of many graphs which are otherwise will be impossible to do using a naïve backtrack based algorithm. In our method we use a *Multi-decision Directed Acyclic Graph* (Multi-decision DAG) to store all of the Eulerian paths. We also show how to retrieve an Eulerian path from it which can be done in polynomial time.

[1]In #P-complete class we ask how many solutions for a given problem, while in NP-complete class we ask whether there is a solution for a given problem.

1.1 Background

The most notable research in the past concerning Eulerian paths enumeration is probably due to de Bruijn, van Aardenne-Ehrenfest, Smith and Tutte. They showed that for a directed graph, the number of Eulerian cycles in it can be expressed using a simple mathematical formula. The formula is called BEST theorem [1] [2], named after their names (**B**ruijn, **E**hrenfest, **S**mith, **T**utte).

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*[2] *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 cycles $Eul(D)$ in D can be stated mathematically as:*

$$Eul(D) = t(D) \prod_{i=1}^{m} (d_i - 1)! \tag{1.1}$$

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. Some researchers have been able to find the approximate number of the Eulerian paths in a specific kind of undirected graph. For example in [7], McKay and Robinson show that the number of Eulerian cycles in a Complete graph[3] is asymptotic.

Almost at the same time as the author proposed the method in this thesis, Hanada et al. [8] tried to solve the same problem by converting the original graph into its *line graph*[4]. Their method is based on the known fact that every Eulerian path of the original graph is corresponding to a Hamiltonian path in its line graph. But since the converse is not true, that some of the Hamiltonian paths of the line graph actually do not correspond to any of the Eulerian path in the original graph, they have to label these Hamiltonian paths and exclude them from the enumeration.

To count the Hamiltonian paths, Hanada et al. use an algorithm introduced by Knuth which is called *simpath* [9] (exercise 225 in 7.1.4). Generally speaking, *simpath* is an algorithm that constructs a binary decision graph which later can be transformed efficiently into a *Zero-suppressed Binary Decision Diagram* (ZDD) [10], a kind of data structure that can be used to compactly represent a set of all *simple paths*[5] between two given vertices of a given graph. *Simpath* is easily extendable to count the Hamiltonian paths instead of simple paths by adding one restriction that only the simple paths those visit all vertices of the graph are accepted.

Our approach to solve Eulerian paths enumeration is also based on *simpath*. But unlike their method, we modify *simpath* further so that it can be used to count the

[2]In a directed Eulerian graph, the in-degree of each vertex must be equal to its out-degree.
[3]A Complete graph is a graph in which each pair of its vertices are connected by an edge.
[4]Line graph is a graph that represents adjacencies between edges of the original graph.
[5]Simple path is a path which does not pass through a vertex more than once.

Eulerian paths instead of the simple paths. Our algorithm is different from *simpath*, since we store the Eulerian paths in a multi-decision DAG instead of a binary decision graph and we also do not use a ZDD. In *simpath*, each vertex is only able to be visited by a path at most once, but in our method we eliminate this restriction and allow each vertex to be visited multiple times.

1.2 Contributions

Our approach and algorithm has allowed us to confirm and extend the results of some previous researches, including the number of Eulerian cycles in a Complete graph C(n) [7] up to $n = 9$, the results from Audibert in his book Mathematics for Informatics and Computer Science [11], and the results from Hanada et al. [8] for various types of graphs.

Although in [7] McKay and Robinson are able to enumerate C(n) for $n = 21$, their technique is only limited to a Complete graph. On the contrary, our method aims for a more general graph, both simple graph and multigraph.

1.3 Thesis Structure

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

Chapter 2

Preliminaries

This chapter is about *simpath* algorithm and the data structure used in it, which is called *Zero-suppressed Binary Decision Diagram* (ZDD).

2.1 Zero-suppressed Binary Decision Diagram (ZDD)

Zero-suppressed Binary Decision Diagram (ZDD) [10] is a labeled directed acyclic graph obtained by reducing a binary decision tree graph. Each node (drawn as circle) in a ZDD is labeled by an input variable, and has two outgoing edges, namely *0-edge* and *1-edge*, which connect a node to its two *child nodes*, *0-child* and *1-child*, respectively. *0-edge* (*1-edge*) of a node e_i is denoted as $e_i = 0$ ($e_i = 1$), and represent a state after the decision that 0 (1) is assigned to that variable. By traversing the nodes of a ZDD from its root node (the node at the top of a ZDD), eventually we will reach either *0-terminal* or *1-terminal*, which shows the output binary value of variables assignment in that path.

If a ZDD consists of n input variables, then it will have $n+1$ levels (height). Each level consists of several nodes which have the same label, and it is common to regard the two terminal nodes as being together in level $n+1$. In this thesis, we always refer to an ordered ZDD, thus 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 (some variables may be missing due to the ZDD reduction rules below).

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

1. **Zero-suppression rule:** eliminate all nodes whose *1-edge* points to the *0-terminal* node. Then connect the edge coming from its parent node to the sub-graph previously pointed by *0-edge* of the erased node. See the left figure of Fig. 2.1.

2. **Merging rule:** share all equivalent nodes which have the same label, *0-child*, and *1-child*. See the right figure of Fig. 2.1.

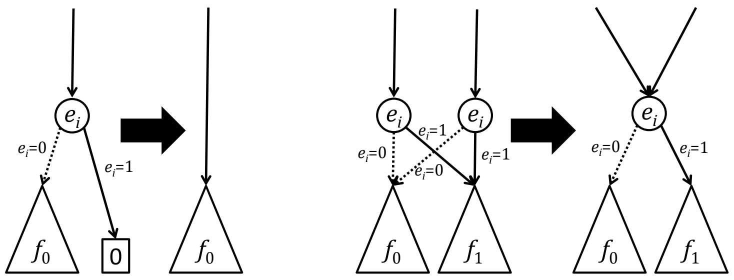

Figure 2.1. ZDD reduction rules

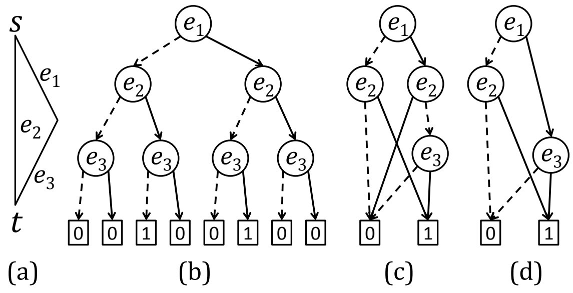

Figure 2.2. (a) A triangle graph, (b) a binary decision tree, (c) a binary decision graph (unreduced ZDD), and (d) a ZDD. Here *0-edges* are drawn as dashed-line.

ZDD is proved to be ideal for compactly expressing family of sets [10]. For example, Figure 2.2 (d) shows how the family of sets $F = \{\{e_1, e_3\}, \{e_2\}\}$ is expressed compactly using a ZDD compared to its corresponding binary decision tree graph (b).

This family of sets $F = \{\{e_1, e_3\}, \{e_2\}\}$ is also representing all simple paths between the source vertex s and the target vertex t of the triangle graph in (a). In the 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.* Furthermore, the path with $e_1 = 0, e_2 = 1, e_3 = 0$ which is representing $\{e_2\}$ is also connected to the *1-terminal.* Strictly speaking, $e_i = 1$ means that edge e_i is selected, and if the set of selected edges is forming a simple path, then the corresponding branch in the binary decision tree will be connected to *1-terminal.* Thus, the number of simple paths of a given graph corresponds to the number of *1-paths* in the binary decision tree, which is also equivalent with the number of *1-paths* in the ZDD.

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

2.2 *Simpath*

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 or the starting vertex of the paths be s and the target of the paths be t, where $s,t \in V$. The goal of *simpath* is to enumerate all simple paths between s and t and to store them in a ZDD.

A simple way to construct a ZDD is by recursively applying ZDD reduction rules into a binary decision tree. The problem is, creating ZDD by this way is not only time consuming, but also need a lot of memory 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 binary decision graph[1] which then can be effectively reduced into a ZDD. The construction of the binary decision graph is done in a *breadth-first search* manner, pruning out the branch that will not produce any simple path in the early stage as possible. For example in Figure 2.2 (a), if both edges e_1 and e_2 are not selected, then no simple path is able to be constructed, so the branch when $e_1 = 0$ and $e_2 = 0$ can be directly appointed to *0-terminal* without having to concern about the assignment of e_3. *Simpath* also merges two nodes with the same label when it is known that the two branches will produce the same output value for any combinations of the rest of the input variables.

A path from the root node to a certain node in a ZDD represents the selected edges so far. These selected edges will form some path fragments, and each vertex will have one of the following three states which need to be remembered by that particular node:

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

Basically, *simpath* decides if a branch of the binary decision graph needs to be pruned out and two nodes can be merged based on the state of the vertices.

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 $V \cup \{0\}$.

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

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

[1]This binary decision graph is different from a binary decision tree, since two different nodes in it can have the same child node. A binary decision graph here can also be regarded as an unreduced ZDD, which generally has the size between a binary decision tree and a ZDD.

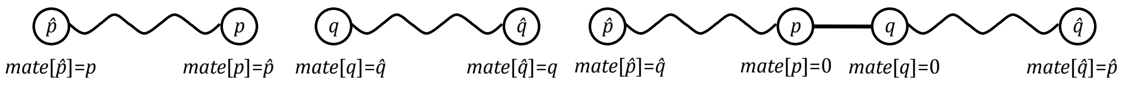

Figure 2.3. Changes in the state of the vertices when an edge *p*–*q* is added

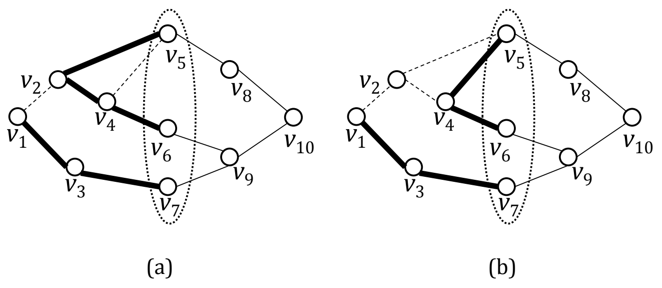

Figure 2.4. Two nodes which are considered to be identical, since both nodes have $mate[v_5] = v_6$, $mate[v_6] = v_5$, and $mate[v_7] = v_1$

If an edge is selected, we need to update the *mate* value of some vertices[2]. Suppose $mate[p] = \hat{p}$ and $mate[q] = \hat{q}$. If edge *p*–*q* is selected, we need to assign $mate[p] \leftarrow 0$, $mate[q] \leftarrow 0$, $mate[\hat{p}] \leftarrow \hat{q}$, $mate[\hat{q}] \leftarrow \hat{p}$, in exactly this order. Figure 2.3 graphically explains what this assignment is actually doing.

A set of selected edges is accepted when the source vertex *s* and the target vertex *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, the node does not have to remember the state of all vertices. All it needs is the information of the vertices which shall be called *frontier*, the set of vertices contained in both processed and unprocessed edge. At the beginning when no edge is processed, the *frontier* is empty. Next when the first edge, let say $e_1 = u$–v, is being processed, then the current *frontier* is $\{u, v\}$. Eventually when all edges connected to a vertex in *frontier* are already processed, then that vertex will be removed from *frontier*.

In Figure 2.4, if we let the thin solid line be the unprocessed edges and the other lines be the processed edges, then the current *frontier* is $\{v_5, v_6, v_7\}$ since they are contained in both the processed edges and the unprocessed edges. Furthermore if we let the thick solid line be the selected edges those will be included in the simple path, and dashed line be the edges not selected, then further selection of the remaining unprocessed edges

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

that makes Figure 2.4 (a) has a simple path will also make (b) has a simple path. If we look closer, then we will know that both (a) and (b) have the same *mate* values for each vertex in the *frontier*. In this case, we do not need to process (b) further, instead we can merge the node representing (b) to the node representing (a).

The Algorithm 1 below is the pseudocode for *simpath*. The *CheckTerminal* function in line 6 is the part where the decision is made, whether we can proceed to the next edge selection, or we have to appoint the current branch of the binary decision graph to *0-terminal*, or we can appoint the branch to *1-terminal*. The pseudocode for *CheckTerminal* function is shown in Algorithm 2. It uses *GetDegree* function which is shown in Algorithm 3 to get the current degree of a vertex v. Line 10 of the code in Algorithm 1 is executed if there is already an equivalent node (node having the same *mate* configurations) created in the binary decision graph, thus the node needs to be merged with this already existing identical node.

Algorithm 1: Simpath [9]

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

Algorithm 2: CheckTerminal($node, e, x, s, t$)

```
input  : $node, e = p–q, x, s, t$
output: 0-terminal, 1-terminal, or nil

if $x = 1$ then
    if GetDegree($p$) $= 2$ or GetDegree($q$) $= 2$ then
        return 0-terminal;
    if ($p = s$ or $p = t$) and GetDegree($p$) $= 1$ then
        return 0-terminal;
    if ($q = s$ or $q = t$) and GetDegree($q$) $= 1$ then
        return 0-terminal;
    if mate[$p$] $= q$ and mate[$q$] $= p$ then
        return 0-terminal;
    if (mate[$p$] $= s$ and mate[$q$] $= t$) or (mate[$p$] $= t$ and mate[$q$] $= s$) then
        foreach vertex $v \in frontier$ do
            if $v \neq s$ and $v \neq t$ and $v \neq p$ and $v \neq q$ then
                if GetDegree($v$) $= 1$ then
                    return 0-terminal;
        return 1-terminal;
foreach $v$ leaving $frontier$ do
    if $v = s$ or $v = t$ then
        if GetDegree($v$) $= 0$ then
            return 0-terminal;
    else
        if GetDegree($v$) $= 1$ then
            return 0-terminal;
```

Algorithm 3: GetDegree(v)

```
input  : Vertex $v$
output: Current degree of $v$, which can be 0, 1 ,or 2
1 if $mate[v] = v$ then // no path fragment is connected to $v$
2     return 0;
3 else if $mate[v] = 0$ then // vertex $v$ is an intermediate point
4     return 2;
5 else // $v$ is an endpoint of a path fragment
6     return 1;
```

Chapter 3

Enumerating Eulerian Paths

In this section, two algorithms for enumerating Eulerian paths are introduced. The first algorithm is the basic version, which in many ways is similar to *simpath*, thus hopefully can be understood easily. In contrast to this, the second algorithm, which is the improved version, uses many modifications to *simpath*. The first algorithm is intended as the introduction to make it easier to understand the improved algorithm. All experiments in this thesis are done using the improved algorithm. In both algorithms, we distinguish two similar Eulerian cycles having different direction. For example, the Eulerian cycle $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_1$ is counted separately from $v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_1$.

3.1 Basic Idea

Suppose there are x_i edges connected to a vertex $v_i \in V = \{v_1, v_2, \cdots, v_m\}$. We want to make pairing of edges in vertex v_i, by at first disconnecting these x_i edges and then reconnecting them in a way such that every edge has a pair. The number of combinations of such edge pairing[1] is

$$f(x_i) = \binom{x_i}{2} \cdot \binom{x_i - 2}{2} \cdots \binom{2}{2} \cdot \frac{1}{\frac{x_i}{2}!} = \frac{x_i!}{2^{\frac{x_i}{2}} \cdot \frac{x_i}{2}!} = \prod_{i=1}^{n/2} (2i - 1). \tag{3.1}$$

Using the counting rules in combinatorics, at first we choose 2 from the x_i edges, then choose 2 from the remaining $x_i - 2$ edges, and so on until no edge remains. But since a pair of edges chosen in the first selection or second or third etc. are not differentiated, the overall number must be divided by $\frac{x_i}{2}!$, which gives us the Equation 3.1.

Now, if we disconnect all edges of the input graph and then reconstruct it by combining all of the possible edge pairing in the entire vertices of the graph, then the total

[1]We assign $x_i \leftarrow x_i + 1$ for vertex s and t if they have odd degree.

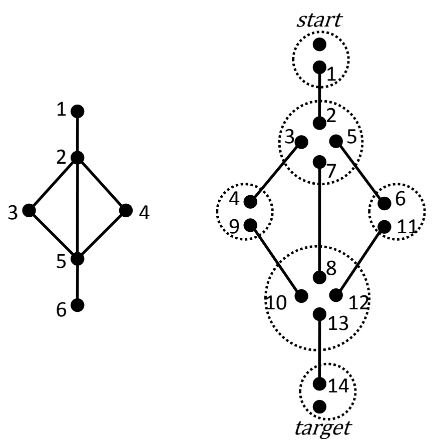

Figure 3.1. Disconnecting the edges of a graph and relabelling its vertices

number of such constructions is

$$\prod_{i=1}^{m} f(x_i). \tag{3.2}$$

Clearly, this number consists of all Eulerian paths of the input graph plus additional graph structures those have a cycle fragment. Our aim is to remove the latter kind of the graphs.

3.2 The Basic Algorithm

In this algorithm, we start from disconnecting all of the edges and relabel the vertices of the input graph such that no two edges have the same vertex. More precisely, let $G = (V,E)$ be the input graph with vertices $V = \{v_1, v_2, \cdots, v_m\} = \{1,2,\cdots,m\}$ and edges $E = \{e_1, e_2, \cdots, e_n\}$. Let the starting vertex of the Eulerian paths be s and the target be t, where $s,t \in V$. Then, the algorithm disconnects the edges of the input graph such that no two edges is connected and also relabels the graph vertices such that vertices $V' = \{1,2,\cdots,2n, start, target\}$ and edges $E' = \{1\text{–}2, 3\text{–}4, \cdots, (2n-1)\text{–}2n\}$, where $start \leftarrow 2n+1, target \leftarrow 2n+2$, and they are being isolated vertices.

Let us define $\gamma_i (1 \leq i \leq m)$ as a set of vertices in the new graph that correspond to the vertex v_i in the input graph[2]. For example, if the left part of Fig. 3.1 shows an input graph with $s = 1$ and $t = 6$, and the right part shows the disconnected and relabeled graph, then we have $\gamma_1 = \{1, start\}$, $\gamma_2 = \{2,3,5,7\}$, $\gamma_3 = \{4,9\}$, $\gamma_4 = \{6,11\}$, $\gamma_5 = \{8,10,12,13\}$, and $\gamma_6 = \{14, target\}$.

The *mate* initialization in the algorithm is done by considering the disconnected graph as path fragments, thus $mate[start] \leftarrow start$, $mate[target] \leftarrow target$, $mate[2i-1] \leftarrow 2i$ and $mate[2i] \leftarrow 2i-1$ for $(1 \leq i \leq n)$. Next if two vertices are going to be connected

[2]The vertex *start* is in the set that corresponds to the vertex s in the original graph, and the vertex *target* is in the set that corresponds to the vertex t in the original graph.

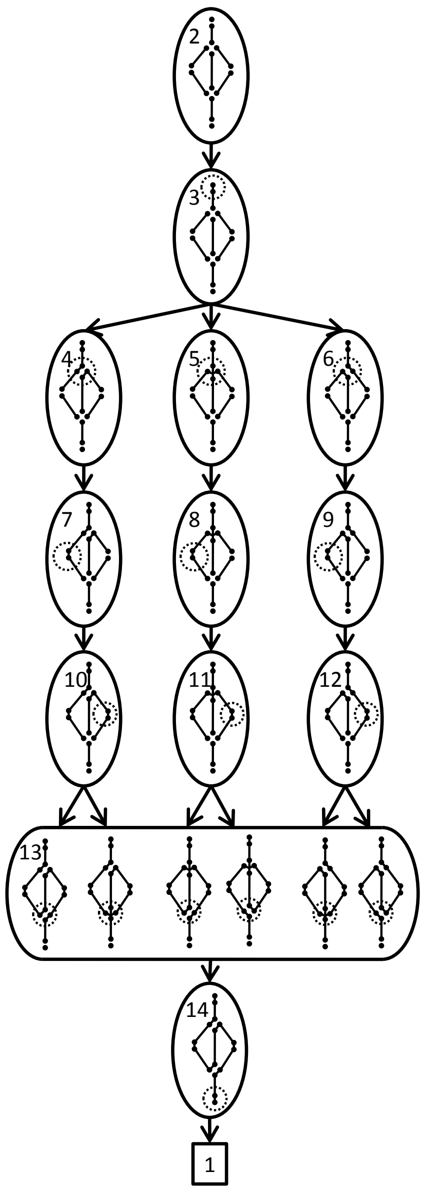

Figure 3.2. The DAG that stores all of the Eulerian paths of Fig. 3.1

(during the edge pairing), we need to update the *mates* of the corresponding vertices using the exactly same rules as in *simpath*.

The algorithm makes a breadth-first search to find all of the Eulerian paths by trying all possible edge pairings in $\gamma_i (1 \leq i \leq m)$[3], while creating a multi-decision DAG that stores all of the Eulerian paths. The DAG pruning and the node merging are carried out with the similar manner as in *simpath*, that is by observing the state of the vertices in *frontier*. For example, the process when the algorithm is used for finding the Eulerian paths in the left figure of Fig. 3.1 is shown graphically in Fig. 3.2.

The Algorithm 4 below shows the pseudocode of the basic algorithm for enumerating

[3]For Eulerian cycles enumeration, vertex *start* and vertex *target* are not allowed to be paired.

Eulerian paths. Actually, this algorithm is time consuming, because the number of the possible edge pairing in γ_i as shown in line 7 of the algorithm, which is also equals to the Equation 3.1, is growing fast as the size of γ_i grows bigger. Moreover, the algorithm is also memory consuming since the node merging is expected to be done in an inefficient way. For example, we cannot merge the node having $mate[a] = x, mate[b] = y (a \neq b, x \neq y)$ with the node having $mate[a] = y, mate[b] = x$ even though vertex a and vertex b are both representing the same vertex c in the original graph. The improved version of this algorithm uses a different approach, it considers path fragments connected to a vertex of the original graph as a set (more precisely, a weighted set). For example in this scenario, the two nodes will have $mate[c] = \{x, y\}$, thus increasing the chance of the two nodes being merged.

Algorithm 4: The basic algorithm for enumerating Eulerian paths

```
input  : An undirected graph, $s$, and $t$
output: Multi-decision DAG that stores all Eulerian paths
1  Disconnect the graph edges and relabel its vertices;
2  Initialize the $mate$;
3  $N_1 \leftarrow \{node_{root}\}$;
4  $N_i \leftarrow \emptyset$ for $i = 2, \cdots, n+1$; // $N_i$ holds nodes in level $i$
5  for $i \leftarrow 1$ to $m$ do // processing $\gamma_i$
6      foreach $node \in N_i$ do // processing nodes in level $i$
7          foreach $posib \in$ possible edge pairing in $\gamma_i$ do
8              $childnode \leftarrow$ CheckTerminal($node, posib$);
               // CheckTerminal returns 0-terminal if a
               cycle fragment is formed, otherwise nil
9              if $childnode$=nil then
10                 Create a new node and set it to $childnode$;
11                 if there exists $childnode' \in N_{i+1}$ s.t. $childnode'$ is equivalent to
                   $childnode$ then
12                     $childnode \leftarrow childnode'$
13                 else
14                     $N_{i+1} \leftarrow N_{i+1} \cup \{childnode\}$
15             Create an arc to connect $node$ and $childnode$ in the DAG.
16 Connect the last node to the 1-terminal;
```

3.3 The Improved Algorithm

In this algorithm, we disconnect the input graph and reconstruct it by putting the edges one by one while considering the edge pairing in the two vertices of each edge. Every time we put an edge, generally we have two choices, whether to connect it to one of the already existing unpaired edges or not. For example, let the edge p–q is being processed and is going to be added to the reconstructed graph so far. If we see at the vertex p, then the possibilities are whether the edge p–q is connected to one of the already existing unpaired edges in p, or it is not connected to any of them. The same condition also applies for vertex q, whether we connect the edge p–q to one of the already existing unpaired edges in q or not. In the other words, at each step we have to consider edge pairing in the two vertices of the currently being processed edge.

As in *simpath*, we keep track on the state of the vertices in *frontier*[4] by using a *mate* array. The difference is, since each vertex may appear in a simple path at most once, each vertex in *simpath* has exactly one *mate* slot. In contrast to this, a vertex in an Eulerian path can appear multiple times. Therefore, during the graph reconstruction process we have to allow a vertex to hold multiple path fragments. This is done by associating each vertex with a *weighted set*, and the *mate* array is now being an array of weighted set instead of an array of integers which is used in *simpath*.

These are some reasons why we use a weighted set instead of the other data containers such as a *set*, a *linked list* or an *array of integers*:

1. There will be some different path fragments those have the same two endpoints, therefore using an ordinary set as *mate* is not possible since it cannot have multiple identical elements.

2. The elements of a weighted set are sorted, which is very important because it increases the probability of two nodes being merged. If we use a linked list or an array of integers, then we have to re-sort its elements each time a change happens.

3. The weight of an element in the weighted set shows how many different path fragments leading to the same other endpoint. We do not need to examine these path fragments one by one when an edge is connected to them, since eventually they will result in the same *mate* configurations. As the result, this will speed up the algorithm.

The weighted set associated with a vertex v_i needs to be a resizable container. Initially when the edges connected to this vertex are not yet processed, the size of the weighted set is zero. If an edge is processed and is connected to a path fragment already in

[4]The definition of *frontier* in this algorithm is exactly the same as in *simpath*, that is the set of vertices contained in both processed and unprocessed edges.

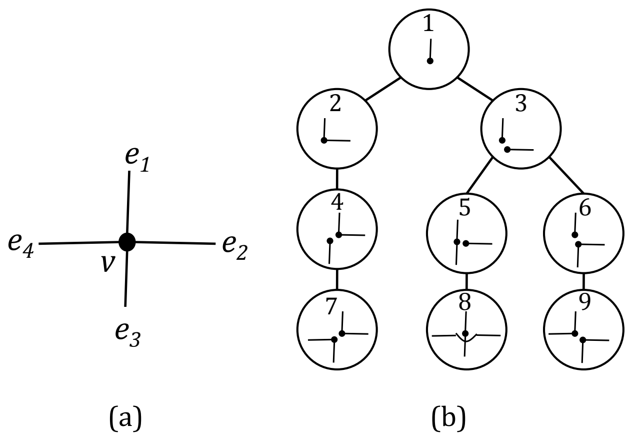

Figure 3.3. Edge matching on a vertex

the vertex, then the size of weighted set is not changed. If the processed edge is not connected to any of the path fragments in the vertex, then the size of the weighted set is incremented, for accommodating this newly added edge. Finally when all of the edges connected to this vertex are processed, the size of the weighted set needs to be exactly[5]:

$$\frac{deg(v_i)}{2}. \tag{3.3}$$

which is equal to the number of appearance of v_i in the Eulerian path.

Summarizing the discussion, we have these two options when an edge is added to a vertex v_i:

1. connect to an unpaired edge in v_i, or
2. do not connect to anything, thus creating a new unpaired edge. This choice is only available if the number in Equation 3.3 is not reached yet.

For example, vertex v in Figure 3.3 (a) has four edges e_1, e_2, e_3 and e_4. Suppose we process the edges in that order. In our algorithm, when adding an edge to the being reconstructed graph we need to consider the edge pairing in its two vertices. But for the sake of explanation, let us take attention only on vertex v. At first when e_1 is being processed, there is still nothing connected to vertex v, so we just put edge e_1 as a new unpaired edge as shown in node 1 of Fig. 3.3 (b). Next, when we are adding edge v_2 we have two possibilities, whether to connect this edge to the already existing edge e_1, or we do not connect to any of the unpaired edges already in the vertex. In node 3, we cannot make a new unpaired edge because the maximal number of Equation 3.3 is already reached, so we just connect the edge e_3 to the existing unpaired edges there.

In our algorithm, we do not necessarily process all of the edges connected to a vertex at once. Sometimes we leave some edges unprocessed, and instead process another edge

[5]Again, we need to make $deg(v_i) \leftarrow deg(v_i) + 1$ for vertices s and t if they have odd degree.

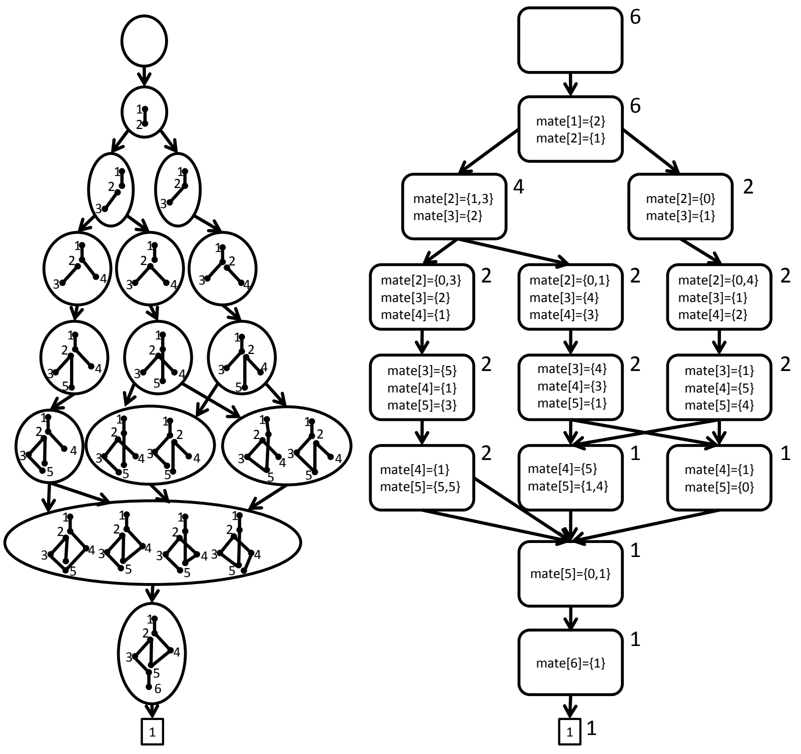

Figure 3.4. Enumerating Eulerian paths of the graph in Fig. 3.1 using the improved algorithm

not directly connected to this vertex. The actual example of edge pairing is shown in Fig. 3.4, which shows what happen when the left figure of Fig. 3.1 is processed using this algorithm.

3.4 *Mate* in the Improved Algorithm

The definition of *mate* in the improved algorithm is different from the *mate* in *simpath* since we are using a weighted set. For a vertex $p \in frontier$, we write $mate[p]$ as $\{\alpha_1, \alpha_2, \cdots, \alpha_g\}$, where $\alpha_1 < \alpha_2 < \cdots < \alpha_g$ (sorted) and each element $\alpha_i (1 \leq i \leq g)$ has its own weight. Since a *frontier* consists of several vertices, the actual *mate* of a node in a computer memory can be described graphically using Fig. 3.5. In this figure, the current *frontier* is $\{f_1, f_2, f_3, f_4, f_5\}$, and each $f \in frontier$ has its own *mate* table which shows the path fragments connected to f. In addition, since the elements of the *mate* table need to be always sorted, in our experiment we use a *multiset* class provided by the *Standard Template Library* (STL) in C++ instead of an array. If we use an array, then we need to re-sort the *mate* table each time a change happens.

Similar to *simpath*, $\alpha_i = 0$ means an intermediate point of a path fragment, and $\alpha_i = v$ for an $v \in V$ refers to a path fragment that has one endpoint at p and the other at v (in the other words, p and v are connected by a path fragment). Different from *simpath*, in this

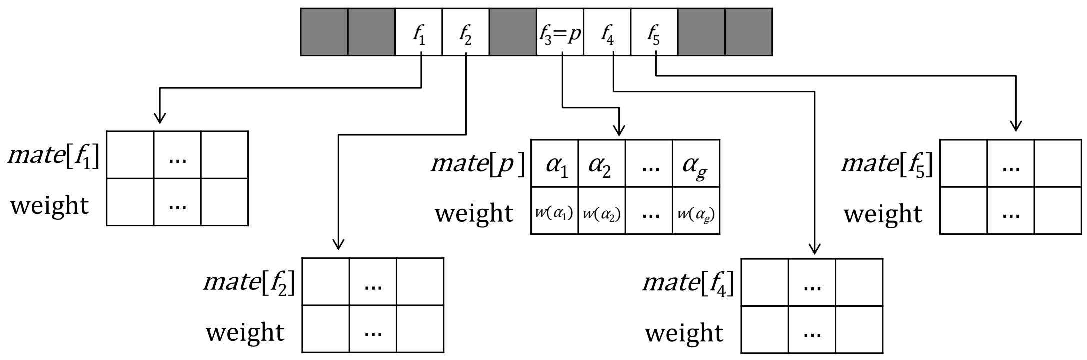

Figure 3.5. The *mate* of a node having $frontier = \{f_1, f_2, f_3 = p, f_4, f_5\}$. The $mate[f]$ for each $f \in frontier$ is assumed to be sorted

algorithm $\alpha_i = p$ does not mean that p is an untouched vertex, but it means that there is a path fragment which has two endpoints at p. An untouched vertex in the algorithm is known by its *mate* size which is equal to zero.

In the implementation, $mate[p]$ is wrapped in a class which has several operators.

- The most basic operator is called $get(\alpha_i)$, which return the actual address of the element α_i inside the computer memory (which is needed since the *mate* table of a vertex is not necessarily an array, thus we assume that there is no such $mate[p][i]$ to call α_i).
- $new(\alpha)$ will create and add a new element α with weight 1 into $mate[p]$. This operator is guaranteed to be called only if no element in $mate[p]$ equals α.
- $delete(\alpha_i)$, to delete α_i from $mate[p]$.
- $inc(\alpha_i)$, to increment the weight of α_i if α_i is already in $mate[p]$, or to call $new(\alpha_i)$ otherwise.
- $dec(\alpha_i)$, to decrement the weight of α_i if the weight of α_i is more than 1, or to call $delete(\alpha_i)$ otherwise.
- $w(\alpha_i)$, that return the weight of α_i.
- $size()$, that will return the current size of $mate[p]$ (which is equal to the total weight of $\alpha_i (1 \leq i \leq g)$).
- The last operator is called $replace(\alpha_i, \alpha_i')$ which will call $inc(\alpha_i')$ and then call $dec(\alpha_i)$, in exactly this order.

Basically, all operators except $new(\alpha)$ will use operator $get(\alpha_i)$ in order to get the actual address of α_i inside the computer memory. Also, we assume that when a change happens to $mate[p]$, then its elements will be automatically re-sorted.

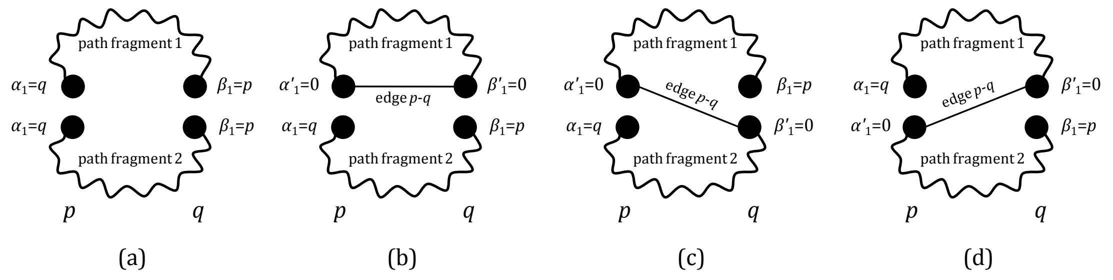

Figure 3.6. Connecting $\alpha_1 = q$ and $\beta_1 = p$

3.5 Updating a *Mate* in the Improved Algorithm

Suppose there are k unpaired edges (path fragments) connected to vertex p and l unpaired edges (path fragments) connected to vertex q. More formally, let we assume $mate[p] = \{\alpha_1, \alpha_2, \cdots, \alpha_g\}$ and $mate[q] = \{\beta_1, \beta_2, \cdots, \beta_h\}$ do not have any element equals zero, and $mate[p].size() = k$ and $mate[q].size() = l$. Then the $k \times l$ combinations of possible edge pairing have to be considered if we add an edge p–q to the reconstructed graph so far. In addition, if there is still a slot to add a new path fragment[6] to the vertex p and q, we have to consider in overall $(k+1) \times (l+1)$ possible edge pairings.

Similar to *simpath*, when an edge is added to the current constructed graph, we need to update *mates* of some vertices. The difference is, since the *mate* of a vertex in our algorithm is a weighted set, the updating rule now is a little different from *simpath*. Furthermore, identifying a cycle fragment being formed is also a bit tricky. In *simpath*, we could not connect vertex p and vertex q if $mate[p] = q$ and $mate[q] = p$ since they will form a cycle fragment. But in our method, $\alpha_i = q (1 \le i \le g)$ and $\beta_j = p (1 \le j \le h)$ does not simply mean that connecting them will form a cycle fragment, since there is a possibility that they are actually belong to two different path fragments. For example in Fig. 3.6 (c) and (d), we can still safely connect α_i to β_j despite $\alpha_i = q$ and $\beta_j = p$. In contrast, connecting $\alpha_i = q$ and $\beta_j = p$ which come from the same path fragment as shown in Fig. 3.6 (b) will cause a cycle fragment being formed, thus need to be avoided.

Since a *mate* only remembers the endpoints of a path fragment, the algorithm could not differentiate which of $\alpha_i = q$ and $\beta_j = p$ those belong to different path fragments from the ones those belong the same path fragment. But fortunately, we know that any possible edge pairing that does not form a cycle fragment will result in the same *mate* configurations. For example in Fig. 3.6, connecting α_1 which comes from path fragment 1 to β_1 which comes from path fragment 2 (as shown in Fig. 3.6 (c)) will have the same identical result with connecting α_1 which comes from path fragment 2 to β_1

[6]When the size of $mate[p]$ or $mate[q]$ is still less than Equation 3.3.

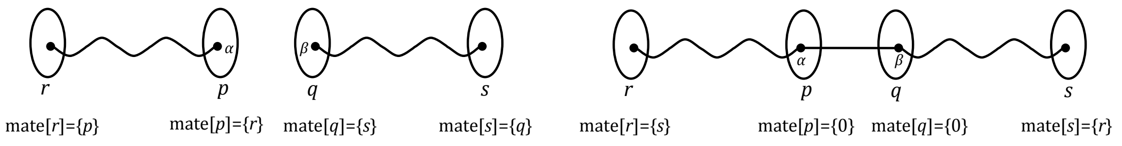

Figure 3.7. Connecting two path fragments in p and q with edge p–q

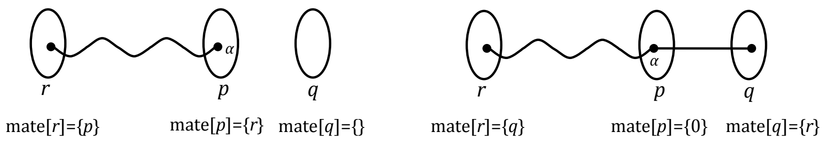

Figure 3.8. Connecting a path fragment in p to vertex q with edge p–q

which comes from path fragment 1 (as shown in Fig. 3.6 (d)), that is the weight of α_1 and β_1 are decreased by one, and the weight of 0s in the both vertices are increased by one. Therefore, our problem can be solved by not examining the $(k+1)\times(l+1)$ cases one by one, instead we need to examine just $(g+1)\times(h+1)$ cases, in which each case has several identical results depending of the weight of α_i and β_j.

Generally, the number of identical results when α_i and β_j are connected by edge p–q is equal to $mate[p].w(\alpha_i)\times mate[q].w(\beta_j)$. The only exception is when a cycle fragment is formed, which happens if $\alpha_i = q$ and $\beta_j = p$. Fortunately, we know that the number of results having a cycle fragment is equal to the number of path fragments connecting p and q, which is equal to the weight of α_i, and which is also equal to the weight of β_j. Therefore in the case $\alpha_i = q$ and $\beta_j = p$, we need to subtract $mate[p].w(\alpha_i)$ from the general number of identical results.

If $\alpha_i = r$ and $\beta_j = s$, then the result when a path fragment having endpoints at r and p is connected to a path fragment having endpoints at q and s by edge p–q is obtained by calling $mate[r].replace(p,s)$, $mate[s].replace(q,r)$, $mate[p].replace(r,0)$, $mate[q].replace(s,0)$. This procedure is very similar with *mate* update in *simpath*, and can be interpreted graphically using Fig. 3.7.

Next, if edge p–q is added to connect a path fragment in p which has another endpoint at $\alpha_i = r$ to vertex q (hence we do not connect edge p–q to any of the path fragments in q, instead we create a new unpaired edge in q), then the result is obtained by calling $mate[r].replace(p,q)$, $mate[p].replace(r,0)$, $mate[q].inc(r)$, which is described graphically in Fig. 3.8.

The opposite condition, if edge p–q is added to connect vertex p with a path fragment in q which has another endpoint at $\beta_j = s$ (hence we do not connect edge p–q with any

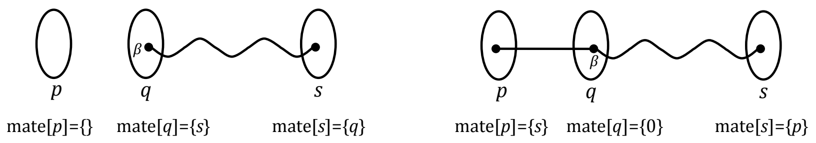

Figure 3.9. Connecting vertex p to a path fragment in q with edge p–q

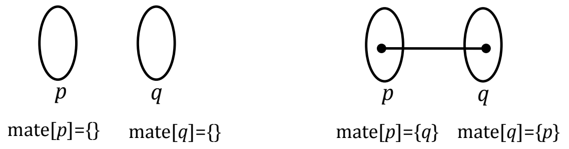

Figure 3.10. Adding edge p–q without connecting it to any of the existing path fragments

of the path fragments in p, instead we create a new unpaired edge in p), then the result is obtained by calling $mate[p].inc(s)$, $mate[q].replace(s,0)$, $mate[s].replace(q,p)$. These procedures are described in Fig. 3.9.

Last, if edge p–q is added without neither connecting it to any of the path fragments in p nor q, then the result is obtained by calling $mate[p].inc(q)$, $mate[q].inc(p)$, which can be described using Fig. 3.10.

Node merging in this algorithm is done in the same way as in *simpath*, that is two nodes having the same *mate* configurations can be merged. In our method, we use a weighted set to express *mate* of a vertex to improve the possibility of nodes merging, since the elements of a weighted set is always sorted. If the elements of a *mate* is not sorted, for example a node having $mate[v] = \{2,1,3\}$ will not be merged with a node having $mate[v] = \{3,1,2\}$. The condition may be different if they are sorted, since both nodes will have the same $mate[v] = \{1,2,3\}$, thus increasing the chance of the two nodes being merged.

In our method, we do not use a *0-terminal* to express a dead end during the graph construction. A branch in the multi-decision DAG that needs to be pruned out will simply be ignored without expanding it further, while a branch that corresponds to an Eulerian path will be connected to *1-terminal*. When the multi-decision DAG is completed, then the number of Eulerian paths of the input graph is equal to the number of paths leading from root node to *1-terminal* in this DAG.

3.6 The Pseudocode for the Improved Algorithm

The Algorithm 5 below is the pseudocode for the improved algorithm. In this algorithm, N_i is a set of nodes in level $i(1 \leq i \leq n+1)$ of the DAG. As shown in line 1 and line 2, N_1 is initialized to contain the root node of the DAG, while for $i > 1$, N_i is initialized to be an empty set, which later will be filled with nodes during the program execution. Next in line 5, the *MateForChildNodes* function tries all possible constructible graphs (the uncompleted Eulerian path) when the edge $e_i(1 \leq i \leq n)$ is added to the reconstructed graph so far. The details of this function are then described as Algorithm 6.

Algorithm 5: The improved algorithm for enumerating Eulerian paths

```
input  : An undirected graph, $s$, and $t$
output: A multi-decision DAG storing all Eulerian paths
1 $N_1 \leftarrow \{node_{root}\}$;
2 $N_i \leftarrow \emptyset$ for $i = 2, \cdots, n+1$; // $N_i$ holds nodes in level $i$
3 for $i \leftarrow 1$ to $n$ do // processing edge $e_i$
4     foreach $node \in N_i$ do // processing nodes in level $i$
5         MateForChildNodes($node$, $e_i$, $N_{i+1}$);
```

Algorithm 6 shows how to connect vertex p with vertex q using an edge p–q. More precisely, line 1 up to line 9 shows how to connect one of the existing path fragments connected to p to one of the existing path fragments connected to q. Next, line 10 is to confirm that the *mate* size of vertex q is still below the number in Equation 3.3, thus we can add an edge to vertex q without connecting it to any of path fragment there. The next 5 lines shows how to connect one of path fragments connected to p with vertex q (which is already explained using Fig. 3.8). Line 16 to line 21 shows the opposite, that is connecting a path fragment in q to vertex p, which is also already explained using Fig. 3.9. The last part, line 22 to line 26 shows how to connect vertex p to vertex q using edge p–q without connecting it to any of the existing path fragments. Since connecting an edge to the intermediate point of a path fragment is not allowed, we assume that α_i and β_j in the entire algorithm is not equal to 0.

Algorithm 7 is used for connecting the newly created child node to its parent node in the multi-decision DAG using $numOfChilds$ many arcs. Line 1 and line 2 are executed when there is only one path fragment connecting vertex p with vertex q (for example in Fig. 3.6 (a), there are two path fragments connecting p with q). Line 3 and line 4 are the parts used for node merging, while line 5 and line 6 are used to add the newly created node into the multi-decision DAG if there is no identical node in there.

Algorithm 6: MateForChildNodes(*node*, *e*, N_{next})

```
input  : node, $e = p–q$, and $N_{next}$
output: –
// Suppose node has mate configurations parentmate, and
$\{\alpha_1, \alpha_2, \cdots \alpha_g\}$ equals $parentmate[p]$ with 0 element being
removed and $\{\beta_1, \beta_2, \cdots, \beta_h\}$ equals $parentmate[q]$ with 0
element being removed
1  for $i \leftarrow 1$ to $g$ do // processing $\alpha_i$
2      for $j \leftarrow 1$ to $h$ do // processing $\beta_j$
3          $childnode \leftarrow node$; // for the child node.  Let the
           mate configurations of childnode being named mate
4          $numOfChilds \leftarrow mate[p].w(\alpha_i) \times mate[q].w(\beta_j)$;
5          if $\alpha_i = q$ and $\beta_j = p$ then
6              $numOfChilds = numOfChilds - mate[p].w(\alpha_i)$;
7          $mate[\alpha_i].replace(p, \beta_j)$; $mate[\beta_j].replace(q, \alpha_i)$;
8          $mate[p].replace(\alpha_i, 0)$; $mate[q].replace(\beta_j, 0)$;
9          CreateChildNodes(node, childnode, numOfChilds, $N_{next}$);
10 if $parentmate[q].size() < \frac{deg(q)}{2}$ then
11     for $i \leftarrow 1$ to $g$ do // processing $\alpha_i$
12         $childnode \leftarrow node$;
13         $numOfChilds \leftarrow mate[p].w(\alpha_i) \times 1$;
14         $mate[\alpha_i].replace(p, q)$; $mate[p].replace(\alpha_i, 0)$; $mate[q].inc(\alpha_i)$;
15         CreateChildNodes(node, childnode, numOfChilds, $N_{next}$);
16 if $parentmate[p].size() < \frac{deg(p)}{2}$ then
17     for $j \leftarrow 1$ to $h$ do // processing $\beta_j$
18         $childnode \leftarrow node$;
19         $numOfChilds \leftarrow 1 \times mate[q].w(\beta_j)$;
20         $mate[\beta_j].replace(q, p)$; $mate[p].inc(\beta_j)$; $mate[q].replace(\beta_j, 0)$;
21         CreateChildNodes(node, childnode, numOfChilds, $N_{next}$);
22 if $parentmate[p].size() < \frac{deg(p)}{2}$ and $parentmate[q].size() < \frac{deg(q)}{2}$ then
23     $childnode \leftarrow node$;
24     $numOfChilds \leftarrow 1$;
25     $mate[p].inc(q)$; $mate[q].inc(p)$;
26     CreateChildNodes(node, childnode, numOfChilds, $N_{next}$);
```

Algorithm 7: CreateChildNodes($node$, $childnode$, $numOfChilds$, N_{next})

```
input  : $node$, $childnode$, $numOfChilds$, $N_{next}$
output: –
1 if $numOfChilds = 0$ then
2     return
3 if there exists $childnode' \in N_{next}$ s.t. $childnode'$ is equivalent to $childnode$ then
4     $childnode \leftarrow childnode'$
5 else
6     $N_{next} \leftarrow N_{next} \cup \{childnode\}$
7 Connect $node$ and $childnode$ using $numOfChilds$ many arcs.
```

3.7 Retrieving an Eulerian Path from the Constructed Multi-decision DAG

To retrieve the number *n*’th Eulerian path from the constructed multi-decision DAG, at first we need to give the nodes in the multi-decision DAG their values. The value of a *1-terminal* node is equal to 1, and the value of the other nodes are equal to the sum of values of their children nodes. For example, the right figure of Fig. 3.4 shows a multi-decision DAG with the value of each node being printed out next to the node. Once this is done, we can determine the path in the multi-decision DAG that correspond to the number *n*’th Eulerian path by comparing *n* with the value of the nodes while traversing the multi-decision DAG from its root node to *1-terminal*. Later, this path can be used as guidance to reconstruct the Eulerian path. The complexity of reconstructing an Eulerian path is $O(|E| \cdot max(\frac{deg(v)}{2})^2)$, since we are adding edges one by one, and each time an edge is added we need to search the right path fragments need to be connected by this edge.

However, there is a limitation in our method since we could not extract a set of Eulerian paths that satisfies a given condition from the constructed multi-decision DAG. For example, we could not get a set of Eulerian paths that has the property that e_1 must be followed by e_2 from the multi-decision DAG.

Chapter 4

Experimental Results

In this section, we present the results of our experiments only for the improved algorithm. In all experiments, the computational time is measured in second unit. The algorithm was implemented using C++11 and run on a machine with 4 GB memory Intel® Core™i3-2330M CPU @ 2.20GHz $\times$ 4.

In this thesis we present the results of our experiments with five different types of graphs. At first, we show the results for graphs having multiple edges and compare the results with the corresponding simple graph. Then, we also present the results for Complete graph. For the Aztec diamond graph [11], we present the results up to $n = 8$. Next, we also show the results for Congolese drawings. At last, we present the results for Ring graph and confirm these results by comparing them with the number of Eulerian paths derived from a mathematical equation.

4.1 Multigraph

The proposed algorithm in this thesis is relatively fast for processing both multigraph[1] and its "corresponding" simple graph. Although processing a multigraph is slightly faster, we could not distinguish some edges once an Eulerian path is retrieved from the DAG, while in a simple graph we can easily distinguish the edges since there is no any multiple edge. For example, let the left graph in Figure 4.1 be $a(n)$ and the right one be $b(n)$. Starting from vertex 1, we want to enumerate all Eulerian paths if n is odd, and all Eulerian cycles if n is even. The results for some n are summarized in Table 4.1. For the same n, we can see in this table that the number of DAG nodes used to store the Eulerian paths of a simple graph is more than the number of DAG nodes used for a multigraph, which makes the processing time for a multigraph becomes faster.

Using a simple backtrack based algorithm to search the number of Eulerian paths in both $a(n)$ and $b(n)$, we already running out of time for $n = 12$. In contrast, using

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

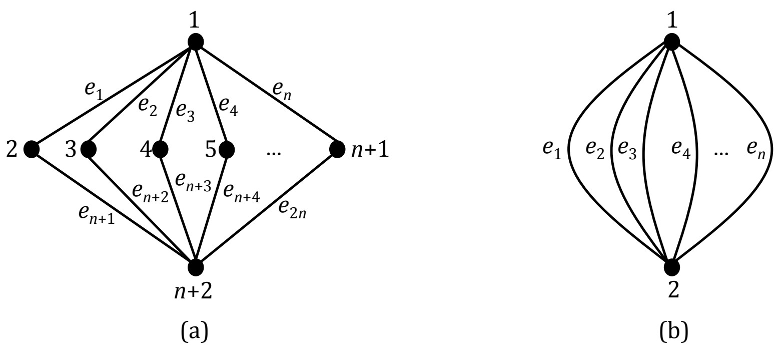

Figure 4.1. Simple Graph vs Multigraph

Table 4.1. Simple Graph vs Multigraph

n	$a(n)$			$b(n)$			solutions
	proposed algorithm		backtrack	proposed algorithm		backtrack	
	nodes	time	time	nodes	time	time	
1	5	0.000331	0.000138	4	0.000258	0.000159	1
2	8	0.000484	0.000392	6	0.000339	0.000195	2
3	14	0.005143	0.000822	8	0.003770	0.000385	6
4	19	0.008264	0.002060	11	0.004230	0.001214	24
5	33	0.014939	0.013594	16	0.008759	0.005399	120
6	42	0.015787	0.065717	21	0.012045	0.021279	720
7	67	0.025232	0.382385	30	0.018849	0.087972	5040
8	82	0.026160	3.187142	38	0.016174	0.650604	40320
9	112	0.028066	30.145249	52	0.025878	5.796462	362880
10	146	0.031964	320.352194	64	0.028012	58.130634	3628800
11	205	0.029756	3612.151051	85	0.023074	638.443941	39916800
12	241	0.036714	time out	102	0.034845	time out	479001600
13	324	0.036364	time out	131	0.025434	time out	6227020800
30	4236	0.191787	time out	1584	0.133952	time out	2.652529×10^{32}
40	11434	0.599417	time out	4169	0.412097	time out	8.159153×10^{47}
50	25276	1.554008	time out	9068	1.097576	time out	3.041409×10^{64}
60	48949	4.022040	time out	17364	2.497769	time out	8.320987×10^{81}
70	86267	8.368751	time out	30349	5.105830	time out	1.197857×10^{100}
80	141674	16.148767	time out	49523	9.789022	time out	7.156946×10^{118}
90	220258	27.067279	time out	76602	16.798351	time out	1.485716×10^{138}
100	327603	44.423667	time out	113518	28.106356	time out	9.332622×10^{157}

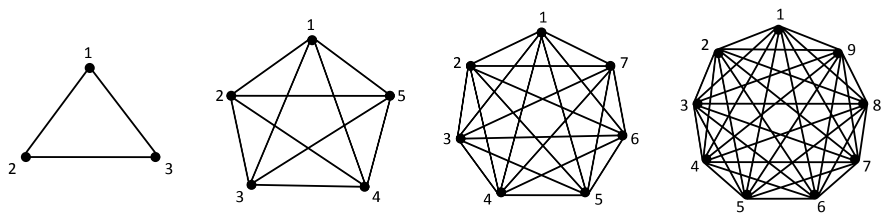

Figure 4.2. Complete graph $C(n)$ for $n = 3, 5, 7$, and 9

Table 4.2. The Results for Complete graph $C(n)$

n	nodes	time	solutions
3	7	0.000442	2
5	50	0.020941	528
7	2168	0.100832	389928960
9	451162	33.361454	3646080228084940800

our proposed algorithm, we are still able to enumerate the Eulerian paths for $n = 100$ in considerably a little time. By comparing the number of nodes with the number of the solutions, we also understand that the compression rate of the DAG in this case is incredibly high. As example for $n = 100$, the number of Eulerian paths in both $a(n)$ and $b(n)$ is 9.332622×10^{157}, and they are stored in a DAG having merely 327603 and 113518 nodes, respectively.

In these kinds of graphs, it is easy to check the number of the Eulerian paths by hand. Especially for graph $b(n)$, the number of Eulerian paths is the same as the number of permutation of the edges $e_1, e_2, \cdots, e_n$, which is $n!$.

4.2 Complete Graph

A Complete graph $C(n)$ is a connected undirected graph having n vertices in which each vertex is connected to the rest of the other vertices of the graph. Examples of such graph are shown in Fig. 4.2. It is known that a Complete graph has an Eulerian cycle if and only if the number of its vertices is odd (which means that the degree of each vertex is even).

The results of our experiments on these graphs are shown in Table 4.2, which confirm the results from McKay and Robinson in [7] up to $n = 9$. Their technique for enumerating Eulerian cycles is specialized only for a Complete graph, and they have been succeeded in computing the Eulerian cycles for $n = 21$. Unfortunately, they did not write the computation time, therefore we could not compare with the results from our algorithm.

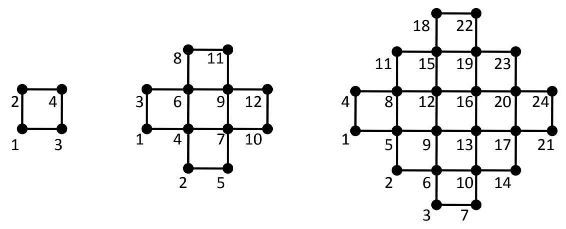

Figure 4.3. Aztec Diamond for $n = 1, 2$, and 3

Table 4.3. The Results 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	1823958835474044219224391680
7	1382302	73.527219	192178269775153104174170778660103782400
8	14083862	942.330957	1495157006436041186484738405257449073460914460033024

4.3 Aztec Diamond

The Aztec diamond in this thesis is a bit different from the usual[2] Aztec diamond. The Aztec diamond here as shown in Figure 4.3 is the same as the graph described by Audibert in his book, Mathematics for Informatics and Computer Science [11]. 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. Audibert also only writes down in his book the number of Eulerian cycles up to $n = 3$. On the contrary, using our proposed algorithm we could count the Eulerian paths up to $n = 8$. The detailed results are presented in Table 4.3.

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

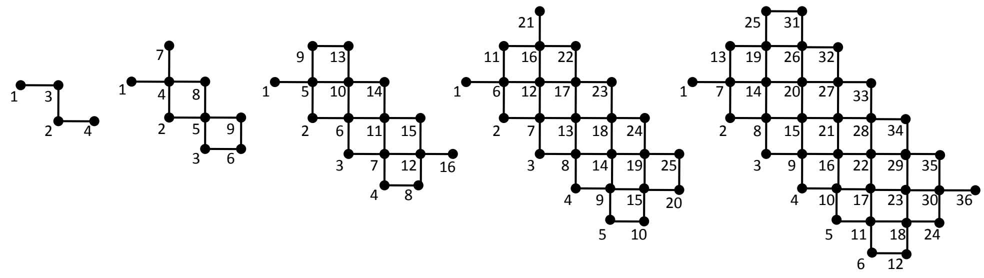

Figure 4.4. Congolese drawings $P(n)$ for $n = 2,3,4,5$, and 6

Table 4.4. The Results for Congolese drawings $P(n)$

n	nodes	time	number of Eulerian paths
2	6	0.000323	1
3	13	0.000662	4
4	58	0.013032	328
5	104	0.023173	111616
6	532	0.031281	632492400
7	769	0.036871	16551741353984
8	4586	0.137933	6727963389195631296
9	5691	0.184710	13207065216910433569559552
10	39989	1.377636	388372864607037921166706218127104
11	44497	1.709373	56689167098449242454996209658960741289984
12	366529	17.958141	121038673236052304885122859695720284551676364284416
13	376365	19.629299	1307909773783592726788040275308591958297845516834378345349120
14	3594666	231.981171	203169716673423007848573790043533223319371983390134005562204298549339136
15	3473985	236.014467	162097094762071097285740713589869499024657858875098611949364517836115641457617010688

4.4 Congolese drawings

The figures in Fig. 4.4 are a kind of Congolese drawings, a pattern that is found in Africa. In page 826 of the same book, Audibert calls these graphs as $P(N,2N)$ graph (in this thesis, we will call this graph as $P(n)$), and writes down the number of Eulerian paths up to $N = 5$. Using our proposed algorithm, we are able to enumerate the Eulerian paths up to $N = 15$ ($n = 15$), which are shown in Table 4.4.

4.5 Ring graph

Ring graph $R(n)$ belongs to multigraph, since there are multiple edges connecting two same vertices in it. Some examples of Ring graph are shown in Fig. 4.5 for $n = 1,2$ and 3. It is named Ring graph, since there are n rings for an $R(n)$. Ring graph has an Eulerian path, since there are exactly two vertices with odd degree. The results of our experiments on Ring graph are shown in Table 4.5.

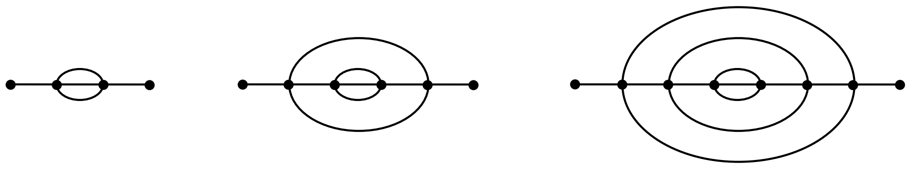

Figure 4.5. Ring graph for $n = 1,2$ and 3

Table 4.5. The Results for Ring graph $R(n)$

n	nodes	time	number of Eulerian paths
1	9	0.002855	6
2	15	0.004163	36
3	21	0.005660	216
4	27	0.006528	1296
5	33	0.007305	7776
6	39	0.009961	46656
7	45	0.010719	279936
8	51	0.012109	1679616
9	57	0.013711	10077696
10	63	0.014786	60466176
20	123	0.019302	3656158440062976
30	183	0.022078	221073919720733357899776
40	243	0.025394	13367494538843734067838845976576
50	303	0.031448	808281277464764060643139600456536293376
100	603	0.045438	6.533186×10^{77}
500	3003	0.450163	1.190214×10^{389}
1000	6003	1.603632	1.416610×10^{778}
5000	30003	37.794911	$5.704951 \times 10^{3890}$
10000	60003	150.083619	$3.254647 \times 10^{7781}$

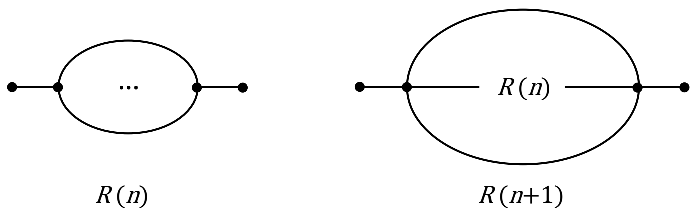

Figure 4.6. Simplifying $R(n+1)$ using $R(n)$

Using mathematical induction, one can prove that the number of Eulerian paths in $R(n)$ is equal to 6^n. For the *base* case $n = 1$, it is clear that the number of Eulerian paths in $R(n)$ is $6^1 = 6$. Assume for $n > 1$, $R(n)$ has 6^n Eulerian paths. To prove that $R(n+1)$ has 6^{n+1} Eulerian paths, at first we need to observe that every Eulerian path in $R(n)$ starts from the leftmost edge and ends at the rightmost edge. Therefore we may replace the $R(n)$ graph inside an $R(n+1)$ graph as an edge, then we can simplify the $R(n+1)$ graph as $R'(1)$, which is shown in Fig. 4.6. This simplified graph $R'(1)$ has 6 Eulerian paths, in which one of its edge is actually $R(n)$. Thus, in overall $R(n+1)$ has $6 \times 6^n = 6^{n+1}$ Eulerian paths.

Chapter 5

Conclusion

We have explained the algorithm for enumerating Eulerian paths in an undirected graph, which can be used for both simple graph and multigraph. The algorithm is based on *simpath* algorithm, which is introduced by Knuth in his book The Art of Computer Programming [9] (exercise 225 in 7.1.4) and used for enumerating simple paths.

The proposed algorithm is much faster compared to a naive backtrack algorithm in many cases, which is confirmed from our experiments. In our algorithm, we store the Eulerian paths of the input graph in a multi-decision DAG, and once it is created, we can retrieve any Eulerian paths in $O(|E| \cdot max(\frac{deg(v)}{2})^2)$.

When using a *frontier* based algorithm, it is preferred to process the edges of the input graph in an order such that the *frontier* size during the graph execution is always fairly small. The reason is because the domain of the possible *mate* values of a vertex is $\{0, s, t, frontier\}$, for a starting vertex s and target vertex t of the Eulerian paths. If the *frontier* size is big, then this domain size also becomes big, which in turn makes the possible combination of *mate* values become bigger, thus reducing the possibility of two nodes being merged. Therefore, the order of edges (variable ordering) is very important in a *frontier* based algorithm. Finding a good variable ordering to be used in our algorithm is still an open problem[1], and beyond the scope of this thesis.

In addition, one problem still remains. In our method we could not extract a set of Eulerian paths satisfying a given condition from the constructed DAG, which we will leave as future work.

[1]For a *Binary Decision Diagram* (BDD) [12], finding a good variable ordering is proved to be in NP-Complete [13]

Acknowledgements

First and foremost, I praise and thank Allah s.w.t, the most beneficent and the most merciful, for giving me strength and ability to complete this thesis. I am indebted to my supervisor, Prof. Shin-ichi Minato, for his advises and helps, which improved both my research life and this thesis. I am grateful to Prof. Zeugmann Thomas, for encouraging and giving me motivation in doing research. I am also indebted all members of Laboratory for Algorithmics and JST ERATO Minato Project, who help me with invaluable suggestions in my research. I express my gratitude to my parents and my family who are living in a far away country, Indonesia, for always supporting me. Finally, I owe my deepest gratitude to my wife, Arfenda Puntia Mustikawati, for her love and support in every aspect of my life.

Bibliography

[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] McKay, B. D., Robinson, R. W.: Asymptotic Enumeration of Eulerian Circuits in the Complete Graph. Combin. Probab. Comput., pp. 437–449 (1998)

[8] Hiroyuki Hanada, Shuhei Denzumi, Yuma Inoue, Hiroshi Aoki, Norihito Yasuda, Shogo Takeuchi and Shin-ichi Minato: Enumerating Eulerian Trails Based on Line Graph Conversion. Hokkaido University, Division of Computer Science, TCS Technical Reports, TCS-TR-A-14-79 (2014)

[9] Donald E. Knuth. *Combinatorial Algorithm*, volume 4A of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, first edition, 2011.

[10] 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)

[11] Audibert, Pierre: Mathematics for Informatics and Computer Science. ISTE Ltd. pp. 832 (2010)

[12] Bryant, R. E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput., 35, pp. 677-691 (1986)

[13] Bollig, B. and Wegener, I.: Improving the variable ordering of OBDDs is NP-complete. IEEE Transactions on Computers, 45(9):993-1002 (1996)