論文 / 著書情報
Article / Book Information

題目(和文)	列挙アルゴリズムの高速化に関する研究
Title(English)	Studies on Speeding up Enumeration Algorithms
著者(和文)	宇野毅明
Author(English)	TAKEAKI UNO
出典(和文)	学位:理学博士, 学位授与機関:東京工業大学, 報告番号:甲第3860号, 授与年月日:1998年3月26日, 学位の種別:課程博士, 審査員:
Citation(English)	Degree:Doctor of Science, Conferring organization: Tokyo Institute of Technology, Report number:甲第3860号, Conferred date:1998/3/26, Degree Type:Course doctor, Examiner:
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Studies on Speeding Up Enumeration Algorithms

Takeaki UNO

Doctoral Thesis

March 1998

Department of Systems Science,
Tokyo Institute of Technology,
2-12-1 Oh-okayama, Meguro-ku,
Tokyo 152, Japan. uno@is.titech.ac.jp

Contents

Acknowledgments

When I was belong to Kojima laboratory, I met two great researchers. One of them is Professor Masakazu Kojima and the other is Professor Akihisa Tamura. From the seminars with Professor Kojima, I had learn what is research. He opened the door to the world of research. His kindly comments and advice had helped me. I would like to express my sincere thanks to him, Professor Masakazu Kojima of Tokyo Institute of Technology.

At that time, Professor Tamura sometimes expressed his interests, favorites, questions and perceptive comments by talking his researches and advising to others in the seminars. His talk seemed very interesting for me. He and his interests lead me to researches of combinatorial optimizations and algorithms. I express my gratitude to Professor Akihisa Tamura of University of Electro-Communications.

My special gratitude is due to Professor Yoshiko Tamura Ikebe of Science University of Tokyo. When I was writing my paper and thesis, she gave me many kindly and useful advice, and checked my papers especially on my poor English. I express my appreciate to her.

The time when I firstly met Professor Tomomi Matsui of Tokyo University is five years ago. He taught me interests of thinking. When he discussed and thought with me about many topics, he dealt with the recognition for me as a researcher. Those days are unforgettable for me. I express my appreciate to him

In my research of algorithms, advice from Professor Tokuyama of IBM Japan can not be avoided. His broad and advanced knowledge of combinatorial optimizations solved the questions on my research. Even for simple or not interest question, his answer had been full kindness. I thank from the bottom of my heart to him.

On network algorithms, Professor Maiko Oe Shigeno of Tsukuba University and Professor Satoru Iwata of Osaka University gave me much advice. She and he taught me the way of logical thinking via studies on network flow problems and submodular functions. I gratefully acknowledge them.

I also express my thanks to Professor Yasuko Matsui of Tokai University and Akiyoshi Shioura of Sophia University. They are good research colleagues and also good friends. I enjoyed researching with them.

One of my research colleagues is professor Mutsutoshi Yagiura of Kyoto University. I have discussed with him about a lot of topics via e-mails. Although it is not so easy to type a number of words to explain his thinking, his e-mails were detailed and kindly. I owe a special debt of gratitude to him.

Special thanks are due to all other members of Professor Kojima's, Professor Takahashi's and Professor Taniguchi's laboratories for their friendship and encouragements. Especially, I express my sincere thanks to super users of computer systems in the laboratories for their considerable work.

Finally, I acknowledge my parents for their supports and encouragement since I was born till now.

Chapter 1

Introduction

There is a well-known problem called "enumeration" in the field of optimization problems. An enumeration problem is to output all elements of a given set explicitly and exactly once. If the set is given by an implicit representation, the problem is non-trivial. For example, such a set consists of all spanning trees in a given undirected graph. The elements to be enumerated are called *objects*. The enumeration problems have been studied for various kinds of objects. For graph objects, we have problems of enumerating all stable sets [2, 30, 34, 45, 52], all cut sets [26, 51], all directed cycles [28], etc. For geometric objects, we also have problems of enumerating all vertices of a given polytope[3, 20, 35], all possible triangulations of a point set [4], etc. Algorithms for solving enumeration problems are called *enumeration algorithms*.

Enumeration problems and their algorithms have been studied since long years ago. Their variations and applications also have been considered. Many improvements have been also proposed for many enumeration algorithms. In this thesis, we discuss about enumeration algorithms and their improvements. Especially, we focus on speeding up of enumeration algorithms.

There are some studies on speeding up of enumeration algorithms, although neither framework nor generalized technique for speeding up has been proposed. The existing studies on speeding up are all ad hoc, and rely on some dynamic data structures deeply. Those data structures are often complicated, and make the algorithms difficult to understand and implement them. From these reasons, we propose two speeding up approaches not based on data structures in this thesis.

The first one is based on the detailed analysis of the amortized time complexity to eliminate the biased excess computation time on some iterations. We consider some distributing rules of computation time on all iterations so that each iteration receives uniform computation time. By this, we can estimate a good upper bound of the time complexity. We can obtain fast algorithms with slight modifications by using the approach. In the later chapter, we propose three enumeration algorithms whose improvements are based on this approach. These three are for enumerating connected induced subgraphs, edge colorings in bipartite graphs and maximal matchings in bipartite graphs. The time complexities of these algorithms are reduced considerably from the original algorithms.

The other one is named "trimming and balancing", which is based on reforming of the input and the structure of recursive calls. The approach adds two new phases to an enumeration algorithm, which are called the trimming phase and the balancing phase. The trimming phase removes unnecessary parts of the input to decrease the computation time to operate them. The balancing phase divides the original problem into subproblems with not so small sizes. By these, we can make a good distributing rule explained the above. We decrease the number of iterations with much computation time. We can reduce the time complexities of trimming and balancing algorithms considerably from the original algorithms. In this thesis, we propose five algorithms based on the approach. These are for enumerating directed spanning trees, matroid bases, perfect and covering matchings in bipartite graphs, and maximal matching in general graphs. The time complexities of these algorithms may not be obtained by the existing techniques for speeding up.

Speeded up enumeration algorithms are practical in some applications. We show several applications of enumeration algorithms as follows.

(1) Enumeration algorithms can be used for solving combinatorial optimization problems. By enumerating all feasible solutions of a combinatorial optimization problem, we can find an optimal solution. Generally, the enumeration takes much more time than the other approaches, but it is one of the simplest and solidest approaches. To solve optimization problems efficiently, we have some modifications of the enumeration. We show one of them in the following, which is called branch-and-bound.

To solve a combinatorial optimization problem, we have a scheme called "branch and bound." The scheme divides the original problem into some subproblems by partitioning the set of feasible solutions into some subsets. By finding optimal solutions in all these subproblems, we obtain an optimal solution of the original problem. The branch and bound scheme applies this dividing operation to the subproblems recursively, until the sizes of subproblems are small enough. This dividing operation is called the "branching".

The "bounding" operation is to avoid solving unnecessary subproblems. During solving the problem, we often obtain some assurances of non-existence of optimal solutions in some subproblems. For minimizing problems, if a lower bound of the objective value of a subproblem is not smaller than an upper bound of the objective value of the original problem, the subproblem cannot include an optimal solution of the original problem. In that case, we stop solving the subproblem. This is called the "bounding".

The branch and bound scheme is considered as a kind of enumeration algorithms. Since the framework of the branching operation is quite similar to the enumeration algorithms, we can apply the techniques of enumeration algorithms to the scheme.

(2) There are some problems of the network reliability which can be solved by enumeration algorithms. Let us consider a network on an undirected graph $G = (V, E)$. Suppose that we have a data base terminal on each vertex, and that any terminal is connected to all the other terminals via edges. In the network, if some edges have troubles and become unavailable, then some terminals will be cut off from the others. Troubles can occur on some edges simultaneously, thus the network may be disconnected. The reliability is important for designing a network system.

Let p_e be the probability of the failure on an edge e. At each time, the edge e is disable with the probability p_e. We assume that each edge fails independently. We define the reliability of the network by the probability of the network being connected, i.e. the probability that all available edges compose a connected subgraph with no isolated vertex. For a subgraph G' of G, the probability of only edges of G' being available in the network is given by $\Pi_{e \in G'}(1 - p_e) \times \Pi_{e \notin G'} p_e$. Thus the reliability is given by the sum of these probabilities over all spanning subgraphs. To compute it, the enumeration is a naive approach. Since there may be a huge number of these spanning subgraphs in a network, we add some modifications to the enumeration in practical cases [46].

(3) There are some applications for investigating facial structures of combinatorial polytopes [42]. For some difficult combinatorial problems, such as the traveling salesman, the maximum stable set, and the minimum steiner tree problems, we have an optimization method called "branch-and-cut". Let us consider an integer programming problem. The above problems are special cases of this problem. The branch-and-cut considers a polytope P which is the convex hull of all feasible solutions of the problem. By solving the linear programming on P, we can obtain the optimal solution of the problem. Generally, P may contain an exponential number of vertices and facets. Thus, the branch-and-cut constructs a polytope P' including P entirely, and cuts off P' by using facets of P. We may find the optimal solution by trimming the polytope P' iteratively. The performance of the approach deeply depends on what kinds of facets we can utilize. Thus to find new kinds of facets is an important study. The enumeration of all facets is a simple and solid approach for finding a new kind of facets.

(4) The enumeration can be used to list up a number of objects. We are often in need to list up all or a number of possible choices for some decision making problems. The possible choices are often expressed by some directed paths or directed spanning trees of a directed graph. The enumerating of all these objects is equivalent to listing up the all possible choices. For small problems, the enumeration is practical to obtain a better choice.

(5) Enumerations are also utilized for generating a given number of objects. Let us consider minimization problems with more than one objective functions. A naive approach for the problem is to find an optimal solution for the first objective function, and generate some feasible solutions not so much different from the optimal solution. We choose the best one among them, which minimizes the other objective functions. This method is also used in some practical problems. We often transform some practical problems to an optimization problem with one objective function. For this kind of problems, an optimal solution of the transformed optimization problem is not always the best choice of the original problem. In such a case, we generate a number of feasible solutions which are not different from the optimal solution so much, and choose a better one.

As we show in the above, enumeration algorithms can be applied to other various problems. They are utilized for sampling of some objects, a subroutine of the other algorithm, column generation algorithms, etc. We can also see the importance of enumeration algorithms from the point of view of their applications. We next show some issues on enumeration studies as follows.

(1) The first is a study on schemes for constructing enumeration algorithms. The enumeration algorithms may be required to output huge number of objects explicitly and exactly

once. For naive algorithms, it is not easy to satisfy this requirement. Hence there are several studies on how to construct enumeration algorithm with simple structures. There are some construction schemes for enumeration algorithms such as binary partition, gray code, and reverse search. Many enumeration algorithms were constructed by using these schemes. Two of these schemes are utilized to construct the algorithms in this thesis. We will explain them in Chapter 3.

(2) The second is on measures of time complexities of enumeration algorithms. For general algorithms, a popular measure of time complexities is the worst case running time for the input size. The goodness of the time complexity has been discussed. Usually, we consider an algorithm as efficient if the worst case running time of the algorithm is bounded by a polynomial of the input size. However many enumeration problems may have exponential numbers of outputting objects. Since the exponential output size hides the difference between the time complexities of distinct algorithms, the measure is not suitable for enumeration algorithms. For example, let us consider the problem of enumerating all s-t paths in a directed graph. The problem is to output all directed paths from a specified vertex s to vertex t. If the graph is a path from s to t, then, we have only one s-t path in the graph. In the case that the graph is complete, there are an exponential number of s-t paths. Therefore it may be not good to bound the running time by only the input size.

In some existing studies, time complexities of enumeration algorithms are bounded by using the largest number of possible outputs. It seems a good bound, although the bound may be quite far from the tight bound if the number of outputs is small.

The recently popular measure of the time complexity of enumeration algorithms is the worst case running time for the input size and the output size. In the case that the functions are polynomial in these sizes, we consider these algorithms as good. Almost all existing enumeration algorithms terminate in linear time of the output size. Generally, the number of outputting objects is huge, thus algorithms are tend to be impractical if the running time is not linear in the number of outputting objects.

There is also the other measure of running time. It is given by an upper bound of the computation time required after finding an object until finding the next one. If the upper bound is a polynomial in the input size, the algorithm is called a *polynomial delay algorithm*. The time complexity of a polynomial delay algorithm is linear in the number of outputs. By using these measures, there have been discussed speeding up of various enumeration algorithms, especially for spanning trees [44, 23, 32, 37, 48, 49, 33, 16, 14, 53] and paths [44, 38, 13].

(3) The third is on the style of outputting. One of the most naive outputting style is to output all the objects explicitly. If we want to output them in a short size, we have to use another outputting style. For example, some algorithms use a technique called the *compact output method* that is to output all objects by the difference from the one just outputted before. In some problems, we can reduce the total size of output by this method.

On the other hand, we have an outputting method called binary decision diagram, written by BDD (see [46]). An algorithm based on BDD outputs a directed acyclic graph with two specified vertices. In the graph, we have a one-to-one correspondence between all objects and all directed paths from one of the specified vertices to the other. The BDD method can reduce the output size drastically for some problems, although it may take much memory space, especially in the worst case, exponential in the input size.

(4) The fourth is on the ordering of outputting, such as the lexicographical order. In the case that objects have some weights, to output some objects in the order of weights is often required. Especially, the problem of outputting the first kth minimum objects is called *finding k-best problems*. These problems are also popular and have been studied much [7, 11, 13, 14, 16].

In this thesis, we improve enumeration algorithms focusing on speeding up, and propose some general approaches for speeding up enumeration algorithms. We also propose speeded up algorithms based on general approaches. We evaluate these algorithms by the worst case running time for the input size and the output size. These algorithms output all objects explicitly, and some of them use the compact output method. The improving methods of these algorithms are based on dynamic data structures, detailed analysis of time complexity, and our new approach for speeding up. These improving methods are described in the following three chapters.

In Chapter 2, we write some notation and definitions utilized in this thesis. Some algorithms are proposed with those notation and definitions in later chapters. We will often use the same symbols there to express the same objects. We also show those notations in Chapter 2.

In Chapter 3, we describe some algorithms improved by data structures. They are for spanning trees in undirected graphs, directed spanning trees in directed graphs, common intervals of two permutations, and distance preserving trees.

In Chapter 4, we propose an approach for speeding up by analyzing the time complexity in detail. We describe some enumeration algorithms improved by the approach. Those algorithms are modified slightly, but terminate in shorter time than the original algorithms. These algorithms are for connected vertex induced subgraphs in undirected graphs, maximal matchings in bipartite graphs and edge colorings in bipartite graphs.

In Chapter 5, we propose a new approach "trimming and balancing" for speeding up enumeration algorithms. The approach is an extension of the approach proposed in Chapter 4. Some algorithms are speeded up, which have not been able to be speeded up by existing studies. These algorithms are for directed spanning trees in directed graphs, bases of matroids, perfect matchings in bipartite graphs, covering matchings in bipartite graphs, and maximal matchings in general graphs.

Chapter 2

Preliminaries

In this chapter, we show some definitions and notations used in this thesis. We often use the same letter to express the same object in the following sections. For example, u and v often denote a vertex of a graph. Here we also show these letters to read this thesis smoothly. To see the details of these definition in this section, see some references of graph theory and algorithms [5, 9].

Firstly we define a *graph*, which is often denoted by G. A graph is composed of two sets. One of them is called a *vertex set* and the other is called an *edge set*. A vertex set is a set of *vertices*. We often denote the vertex set by V. In the following chapters, we often denote a vertex by u, v or w. An *edge* is an element of $V \times V$. An edge is denoted by (u, v) for two vertices u and v. In the following chapters, we often denote an edge by e or f. Edges are adjacency informations among the vertices of the graph. For an edge (u, v), u and v are called its *endpoints*. An edge is called a *self loop* if its two endpoints are the same vertex. An edge set is a correction of edges. An edge set is often written by E. With these sets, a graph is often denoted by $G = (V, E)$.

Graphs are classified in two groups. One of them is *undirected graphs* and the other is *directed graphs*. The difference between them is whether we consider the edge set as an ordered set or not. For two vertices u and v, edges (v, u) and (u, v) are considered as the same edge in undirected graphs, and not in directed graphs. In directed graphs, we call edges *arcs*, and edge sets *arc sets*. We often denote an arc set and a directed graph by A and $G = (V, A)$. In the following chapters, we often use a for the notation of an arc. For a graph G, we use notations $V(G)$, $E(G)$ and $A(G)$ to denote their vertex set, edge set and arc set. An edge or arc (u, v) is called called a *multiple edge* or a *multiple arc* if there is the other edge or arc sharing both its endpoints with (u, v). An edge set or an arc set of a graph G is a subset of $V \times V$ if it contains neither multiple edge nor multiple arc. Moreover, if it contains no self loop, the graph is called a *simple graph*. In the case that the edge set or arc set includes some multiple edges or multiple arcs, the graph G is called a *multiple graph*.

The vertices and edges may have some weights. The weights of vertices are given by a function $V \to R$. The weights of edges are also given by a function $E \to R$. The weightings are often denoted by w. The weights w of a vertex v and an edge e are denoted by $w(v)$ and $w(e)$.

Some of our algorithms are given these graphs as its input. When we show the time

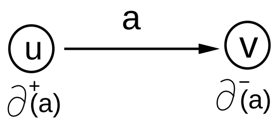

Figure 2.1: Vertices v and u, and an arc $a = (u, v)$. The tail $\partial^{+}(a)$ of a is u and the head $\partial^{-}(a)$ of a is v. a is an in-coming arc of v, and an out-going arc of u.

complexity of algorithm, we often bound the time complexity by some polynomials of the input size. In the case that the input is a graph, the input size is depend on the number of vertices, edges and arcs. To write the time complexities simply, we denote the number of vertices in the graph G by n, and the number of edges and arcs by m. Generally, the input graph includes no isolated vertex, and we have $n = O(m)$. Hence m is often used to express the size of the graph. We also use the notation $|G|$ to denote the number of vertices and edges in the graph G.

For vertices and edges of graphs, we introduce the incidence and the adjacency of them. If there is an edge (u, v), we say that the vertex v is *incident* to the edge and the edge is incident to v. For a vertex u, we say that u is *adjacent* to a vertex v if there is an edge (u, v) in the graph. We also say that u and v are adjacent. Similarly, for an edge or arc e, we say that e is *adjacent* to an edge or arc f if e and f share one of their endpoints. They are also said to be adjacent. For a vertex v, the *degree* of v is the number of edges incident to v. The degree of v is denoted by $d(v)$. The largest degree among all vertices in the graph G is called *maximum degree*, and denoted by $\Delta(G)$. We often denote it simply by Δ especially for the graph G. The number of maximum degree vertices in G is denoted by $\sigma(G)$. A vertex with the degree 0 is called an *isolated vertex*.

For directed graphs, we have some notations related to the directions. For an arc $a = (u, v)$, let the *direction* of the arc be from u to v. We call the endpoints u the *tail* of a, and v the *head* of a. We denote them by $\partial^{+}(a)$ and $\partial^{-}(a)$, respectively. We often assume that we trace an arc from its tail to its head when we search or traverse the graph by some graph search algorithms (see Figure 2.1). For a vertex v of a directed graph, the *in-coming arcs* of v are the arcs whose heads are v. Similarly, the *out-going arcs* of v are the arcs whose tails are v. We define the *in-degree* and *out-degree* of v by the number of in-coming arcs and out-going arcs of v, respectively.

For a graph G, we define *subgraphs* of G by the graph $G' = (V', E')$ such that $V' \subseteq V, E' \subseteq E$, and an endpoint of any edge in E' is in V'. For a given edge subset E', the subgraph with no isolated vertex is uniquely defined. We often say that the subgraph is induced by E'. For a subset $V' \subseteq V$ of G, we define its *induced subgraph*. The induced graph by V' is given by the subgraph with vertex set V' and the set of all edges of E whose both endpoints are included in V'. For a vertex v of G, $G \setminus v$ denotes the graph obtained by removing v and all edges incident to v from G. For an edge e of G, we consider two types of subgraphs of G obtained by removing e. The first one is $G \setminus e$ which denotes the subgraph obtained by removing e. The second one $G^{+}(e)$ is obtained by removing e, both endpoints of e and all edges adjacent to e from G.

For two vertices u and v, to *contract* u and v is to replace the vertices by a vertex w. For all edges whose endpoints are u or v, the endpoints are replaced by w. In the contracted subgraph, some self loops, multiple edges and arcs may occur however the original graph is simple. The contract operation is extended to vertex subsets. For a subset V' of vertices of G, to contract G by V' is to replace all the vertices of V' by a vertex. For any edge or arc, its endpoints are replaced by the vertex if the endpoints are in V'. The obtained graph is called a contracted graph by V', and denoted by G/V'. For an edge or arc e, we also consider the contract operation. The contracted graph by an edge is given by the contracted graph by its two endpoints. It is also extended to an edge or arc subset E'. The contracting operation by the edge subset is done by contracting each edge or arc iteratively. These contracted graphs are denoted by G/e and G/E'.

We have some classes of graphs. One of them is *bipartite graphs*. A bipartite graph is a graph satisfying that there is a partition V_1 and V_2 of its vertex set such that all edges of the graph have their endpoints in both V_1 and V_2. Bipartite graphs can be considered in both undirected and directed graphs.

We also have another class called *planer graphs*. A planer graph is a graph satisfying that there is an embedding of the graph in the Euclidean plane such that no two edges cross each other except for on their endpoints. Edges of a planer graph are not in need to be embedded by straight lines. Planer graphs have some good properties. One of them is that any simple planer graph has only $\mathrm{O}(|V|)$ edges or arcs. It is often used to bound the time complexity tightly.

The *line graph* of a graph G is a graph representation of the adjacency of edges or arcs of G. It is given by the graph whose vertex set is $E(G)$, and any its edge connects two vertices of $E(G)$ if the corresponding edges of G are adjacent. Line graphs can be also considered for directed graphs. For these characterized graphs, such as bipartite, planer and line graphs, the other normal graphs are called *general graphs*.

For an undirected graph G, we have some characterizations of subgraphs. A *path* is a subgraph of G induced by edges $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)$. It is called a path from v_1 to v_k, or from v_k to v_1. We call the vertices v_1 and v_k the *endpoints* of the path, and others the *internal vertices* of the path. The edges (v_1, v_2) and (v_{k-1}, v_k), which are the edges adjacent to the endpoints in the path, are called the *ends* of the path. Especially if any pair of vertices of the path are distinct, we say that the path is *simple*. Usually, a path denotes a simple path if we do not denote them explicitly.

The directed version of paths is also considered. A *directed path* is defined by a subgraph induced by arcs $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)$. It is called a directed path from v_1 to v_k. Since the directed path is composed of directed arcs, we do not say that the path is from v_k to v_1. A directed path from a vertex v to an arc a is a directed path from v to the head of a including a. Similarly, we define a directed path from an arc to a vertex, and a directed path from an arc to the other arc. A directed path is called simple if no vertex is appear in the path twice. In directed graphs, directed paths are often simply called paths if we do not denote them explicitly. A directed or undirected path is denoted by P mainly. A path is uniquely expressed by its vertices or edges and their order, hence we sometimes denote them by sequences of vertices, edges or arcs.

A *cycle* is a subgraph of G composed of edges $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), (v_k, v_1)$. A cycle can be considered as a path whose both endpoints are the same vertex. A cycle is said to be simple if any vertex of the cycle is incident to exactly two edges of the cycle. A cycle denotes a simple cycle if we do not denote them explicitly. The directed version of the cycles is also considered. We have *directed cycles* composed of arcs $(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), (v_k, v_1)$ in directed graphs. If any pair of arcs of the cycle do not share their heads and tails, the directed cycle is called simple. In directed graphs, directed cycles are simply called cycles if we do not denote them explicitly. These cycles are denoted by C mainly. They are also expressed by the sequences of their vertices, edges or arcs. A directed graph with no directed cycle is called an *acyclic graph.*

Two vertices u and v of an undirected graph G are said to be *connected* if there is a path from u to v. If any pair of vertices in G are connected, we say that G is connected. A vertex set of an undirected graph can be partitioned into some subsets uniquely such that two vertices of the graph are connected if and only if they are in the same subset of them. We call the partition the *connected component decomposition* of G, and the vertex induced subgraph by a subset of them a *connected component.* Any edge of G is included in one of the connected components.

In a directed graph G, we say that v is *reachable* from u if there is a path from u to v in G. In this case, we also say that u can *reach* to v. If u and v can reach to each other, we say that u and v are *strongly connected.* If any vertex of G is reachable from all the other vertices, we say that G is strongly connected. Similar to the connected component decomposition, the vertex set of a directed graph can be partitioned into some vertex subsets uniquely such that any two vertices are strongly connected if and only if they are in the same vertex subset. We call the partition the *strongly connected component decomposition* of the graph, and the subgraphs induced by these vertex sets *strongly connected components.* Different from the connected component decomposition, there may be some arcs included in no strongly connected component. We consider a directed graph obtained by contracting all arcs included in strongly connected components. The vertices of the obtained graph correspond to all the strongly connected components. The graph is called the *strongly connected decomposition graph* of G. The graph contains no directed cycle from the definition. Some multiple arcs may occur in the graph however the original graph contains no multiple arc.

We next introduce the *length* of a path. For given weights, the length of a path is given by the sum of weights over all edges in the path. Without any weight, the length expresses the number of edges in the path. The length is defined for both undirected and directed graphs. A *shortest path* from a vertex u to v of a graph is a path with the minimum length among all paths from u to v. The shortest path is also considered in directed graphs. In the case, the shortest paths are directed paths with the minimum weight among all directed paths from u to v.

We have some classes of subgraphs other than paths. One of them is called *forests* and *trees.* A forest is a subgraph of G without any undirected cycle. If a forest is connected, it is called a tree. A forest with the maximal cardinality in the graph is called a *maximal forest* or *spanning forest*. Especially, if a tree spans all vertices of G, i.e. all vertices are incident to some edges of the tree, the tree is called a *spanning tree*. A spanning tree always contains exactly $n - 1$ edges. We denote these forests and trees by T mainly. The end vertices of trees and forests, which are incident to only one edge of the tree, are called *leaves* of the tree. The other vertices are called the *internal vertices* of the tree. Since a forest contains edges at most the number of its vertices minus one, the number of internal vertices with degrees more than two is less than the half of the number of its vertices. In a tree, the path from a vertex u to the other vertex v is unique. We denote the path by P_{uv}. We call the length of P_{uv} *distance* between u and v.

The directed version of trees are also considered. A *directed tree* of a directed graph is a tree such that all whose vertices are reachable from a certain vertex called the *root*. If a tree spans all vertices, it is called a directed spanning tree. We often denote the root by r, and a directed tree by T. A directed spanning tree satisfies the condition that no pair of its arcs share their heads. Any tree satisfying this condition forms a directed spanning tree. A general tree is called *undirected tree* for directed trees. We sometimes consider the root vertex in undirected trees to define the following parent-child relationship among vertices of the tree.

A parent-child relationship is defined for an undirected or directed tree with a root vertex r. For a vertex v of a tree, its *parent* is given by the vertex adjacent to v in P_{rv}. The vertices on P_{rv} are called the *ancestors* of v. Especially the ancestors of v except v are called *proper ancestors* of v. The root r is an ancestor of any vertex in the tree. Conversely, a vertex whose parent is v is called a *child* of v. The vertices that v is their ancestor are called *descendants* of v. The descendants of v except v are called *proper descendants* of v.

By using these parent-child relationships, we can classify all edges of the graphs into some groups. For an undirected spanning tree T of a graph G, we call the edges of T *tree edges*, and the other edges *non-tree edges*. Non-tree edges are classified in some groups with a root vertex of T. The first group is composed of edges called *back edges*. A back edge is an edge connecting a vertex and its ancestor. The second is composed of *non-back edges* which are connecting two vertices of T which are not ancestors of the others each other. Similarly, we define *tree arcs* and *non-tree arcs* for a directed graph and a directed spanning tree. Non-tree arcs are also classified in some groups by the similar way. A non-tree arc is called a *back arc* if its head is an ancestor of its tail. In the case that its tail is an ancestor of its head, it is called a *forward arc*. The rest non-tree arcs are called *cross arcs*. A cross arc connects two vertices such that each them is not an ancestor of the other. Forward arcs and cross arcs are called *non-back arcs*.

In some problems and algorithms, we are in need to visit all vertices and edges by traversing the graph. The *graph search algorithms* are for these purposes. For a vertex of the given graph, we can find the other vertex by tracing an edge or arc. By using this tracing operation iteratively, graph search algorithms traverse the whole of the graph from a specified vertex r. To avoid visiting a vertex twice, the algorithms mark the visited vertices, edges and arcs. When all vertices, edges and arcs reachable from the starting vertex are marked, the algorithm stops. If the given graph is neither connected nor strongly connected, we may have some non-mark vertices however we marked all reachable vertices. In the case, we choose one of these vertices, and continue the searching starting from the vertex.

There have been proposed some graph search schemes. They are *depth-first search*[50, 9] and *breadth-first search* [39, 9]. We simply denote them by DFS and BFS.

DFS firstly finds an edge or out-going arc e incident to r, and puts a mark to e. Next it gets the other endpoint u of e by tracing e. If both e and u are unmarked, we continue the search from u recursively. After searching u or in the case that u is marked, it finds the other out-going arc or edge incident to v, and continue the search. After tracing all edges or arcs incident to v, it terminates the searching form v. We describe the details of the undirected version of the algorithm as follows.

```
ALGORITHM: DEPTH_FIRST_SEARCH ($G$, $r$)
Step 1: Put a mark to $r$.
Step 2: Choose an unmarked edge $e$ incident to $r$. Put a mark to $e$. Otherwise stop.
Step 3: Trace $e$ and get the other endpoint $u$.
Step 4: If $u$ is not marked, call DEPTH_FIRST_SEARCH ($G$, $u$).
Step 5: If there are some unmarked edges incident to $v$, go to Step 2.
```

The algorithm terminates in $O(m + n)$ time [50, 9]. A graph search algorithm generates a tree by its traversing route. We call the tree the *search tree*. For DFS, the search tree is obtained by outputting the edge e in Step 4 if u is unmarked. Since the algorithm never visit the marked vertices, the outputted edges certainly compose a tree. A search tree of DFS is called a *depth-first search tree*.

By using DFS, we often put indices to vertices in the order of visiting of the search. We put indices to the vertex r in Step 1 in the increasing order from one to n, and obtain indices for all vertices after terminating the search. In the depth-first search tree, the index of a vertex is known to be smaller than the index of any its descendant. There is also the other manner of putting indices called the *post ordering*. In the manner, we put an index to a vertex when we finished searching for all edges incident to the vertex, i.e. we put an index to r in the increasing order after Step 5 of each iteration. In the ordering, it is known that the index of a vertex is larger than any index of its descendants in the DFS tree [50, 9].

It is known that DFS trees satisfy some conditions [9]. In the case that the given graph is undirected, a DFS tree has no non-back edge. If the graph is directed, a DFS tree does not have cross arcs whose head has an index larger then its tail. The other cross arcs, forward arcs and back arcs may occur for DFS trees. In a DFS tree with post ordering indices, any path from a vertex v to the other vertex with an index larger than v includes some ancestors of v, from the way of searching. DFS trees also satisfies the following property.

Property 1 *Suppose that three vertices v, w and u satisfy that the index of v is less than w, and the index of w is less than u. Then, any directed path from v to a vertex u includes a common ancestor of u and w.*

Proof: Suppose that there exists a simple directed path P from v to u not including common ancestors. Let c be the ancestor of v which is the child of the nearest common ancestor of v and w. By connecting the directed path from c to v on T and P, we obtain a directed path from w to u without any ancestor of c. This contradicts that T is a depth-first search tree. ∎

Breadth-first search is also a scheme for graph search. It also traces an unmarked edge or out-going arc e incident to r, but does not continue the searching from the other endpoint of e immediately. Instead of the search from the endpoint, we insert the endpoint in a queue, and mark the vertex and edge. After tracing all edges or out-going arcs incident to r, the BFS extracts a vertex from the queue, and continue the search from the vertex until the queue is empty. To say, this is breadth-firstness. We describe the details of the undirected version of a BFS algorithm as follows.

```
ALGORITHM: BREADTH_FIRST_SEARCH ($G$, $r$)
Step 1: Put a mark to $r$.
Step 2: For all unmarked edge $e$ incident to $r$, get the other endpoint $u$. Put a mark to $e$.
        Insert $u$ to the queue if it is unmarked. Put a mark to $u$.
Step 3: Extract a vertex $u$ from the queue. If the queue is empty, stop.
Step 4: If $u$ is marked, go to Step 3.
Step 5: Set $r$ to $u$. Go to Step 1.
```

The search terminates in $O(m+n)$ time [9]. The search tree of BFS is called a *breadth-first search tree*. BFS trees also satisfy some good properties. If the given graph is undirected, a BFS tree has no back edge. Further more, any edge connecting two vertices such that the distances from r to them differ at most one. In the case of directed graphs, forward arcs of the tree satisfy the condition, but cross arcs and back arcs may not satisfy. Any directed path in the tree from r to a vertex has the minimum length among all paths connecting r and the vertex in the given graph.

For a graph G, a *matching* is a set of edges in $E(G)$ such that no two edges of it share their endpoints. Edges of the matching are called *matching edges*. A matching is called *maximal* if it is contained in no other matching properly. The matching is also called *maximum* if it has the maximum cardinality among all matchings contained in G. Especially, we say that the matching is *perfect* if all vertices of G are incident to matching edges. Matchings have been studied in bipartite graphs rather than general graphs.

For a matching M of G, we call a cycle *alternating cycle* if matching edges and the other edges appear in the cycle alternatively. Similarly, we call a path an *alternating path* if matching edges and the other edges appear in the path alternatively, and the edges on the end edges of the path are adjacent no matching edge. We can obtain the other matching by exchanging the matching edges and the other edges along an alternating path or cycle. An alternating cycle gives a matching with the same cardinality. In the case that the length of the alternating path is even, the obtained matching has the same cardinality to the matching M. Otherwise, their cardinalities differ just one. On the other hand, the symmetric difference between M and another matching is composed of some alternating cycles and paths. Thus a necessary and sufficient condition to exist the other matching is that there are some alternating cycles or paths.

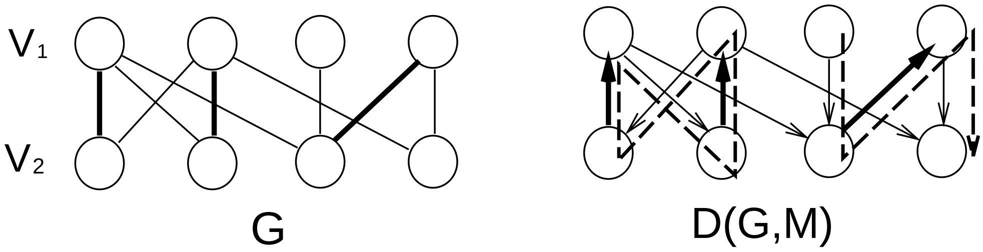

Figure 2.2: The graph $D(G, M)$ and a cycle and a path corresponding an alternating cycle and path in G.

For bipartite graphs, we utilize a directed graph $D(G, M)$ for finding alternating paths and cycles. It is defined for a bipartite graph $G = (V_1 \cup V_2, E)$ and a matching M. The vertex set of $D(G, M)$ is given by $V_1 \cup V_2$. The arc set of $D(G, M)$ is given by orienting edges of M from V_1 to V_2, and the other edges of G in the opposite direction. For any directed cycle in $D(G, M)$, edges of M and the other edges appear alternatively in the cycle of G corresponding to it. The condition also holds for any directed path in $D(G, M)$ satisfying that the ends of the path are not adjacent to arcs from V_1 to V_2. Conversely, any alternating cycle or path is a directed cycle or path in $D(G, M)$. Thus, there is a one-to-one correspondence between directed cycles in $D(G, M)$ and alternating cycles of G and M. We also have one-to-one correspondence between directed paths in $D(G, M)$ and alternating paths of G and M (see figure 2.2).

Similar to the matchings, we define *stable sets* of G. A stable set is given by a vertex set such that no pair of its vertices are connected by an edge of G. A stable set is called a *maximal stable set* if it is included in no other stable set properly. A matching of G is a stable set in the line graph of G. Hence we can transform the problems of matchings to the problems of stable sets.

For these graph objects, the weights of them are defined by the sum of weights over all vertices or edges included in them. For spanning trees, their weights are given by the sum of all their edge weightings. In some optimization problems, the weights have been considered to obtain a minimum cost one. A spanning tree is called a *minimum spanning tree* if the weight of the tree is minimum among all the spanning trees in the graph. Similarly, a *minimum directed spanning tree*, a *minimum weight maximum matching*, and a *minimum weight perfect matching* are given by those whose sums of edge weights are not greater than all the others. The stable set with the minimum weight of vertices is called a *minimum weight stable set.*

An *edge coloring* is the way of coloring of all edges in G such that no two same color edges are adjacent. Since the set of same color edges forms a matching, we express an edge coloring by a set of matchings corresponding to each color. A bipartite graph G is known to have an edge coloring with $\Delta(G)$ colors. Clearly, it is the fewest possible number of colors. In this thesis, we treat only these colorings with the fewest number of colors, and call them simply edge colorings.

Chapter 3

Improved Enumeration Algorithms with the Use of Data Structures

In this chapter, we propose some enumeration algorithms with the use of data structures. Almost all recently improvements for enumeration algorithms are depend on some dynamic data structures and algorithms for maintaining them. Especially, for algorithms based on recursive calls, the approach often works efficiently since the input of a parent-problem and its child-problem usually differ not so much. The time complexities of improved algorithms are depend on the updating time of their data structures when a recursive call occurs. The important parts of this improving method are how to update the data structures and how to utilize the good rules to generate subproblems requiring a few changes of the data structures.

This approach is a naive one in improving methods of general algorithms. Thus it has been applied to some enumerating algorithms naturally. For example, enumeration algorithms for spanning trees of undirected or directed spanning trees, and finding the (not simple) k-best directed path are improved by some data structures [32, 49, 16, 15, 14, 13]. In this thesis, we also have four algorithms improved by data structures. These are for spanning trees of undirected graphs, directed spanning trees in directed graphs, common intervals of two permutations and distance preserving trees. They are described in the later part of this chapter.

Before describing our algorithm, we explain some construction schemes for enumeration algorithms, which are also utilized to construct our algorithms. Enumeration problems have been studied for long years. Construction schemes of enumeration algorithms are one of those studies. Almost all enumeration algorithms in existing studies were constructed with these schemes. Nowaday, there are proposed several construction schemes. Our algorithms in this thesis are also based on two of these schemes. They are "binary partition" and "reverse search".

The first one "binary partition" is simplest among those schemes. It is also called "back tracking." For a given set of objects to be enumerated, the scheme partitions it into two subsets. Generally, the number of these subsets does not have to be two. One of our algorithms divides the set into a number of subsets. After dividing the problem, we generate two subproblems of enumerating all objects in each of these two subsets. If these subproblems can be generated in a simple way, we can enumerate all objects in the original set by solving the two subproblems recursively. For enumerating combinatorial objects, we often partition the original set to the set of objects including a specified element and the set of objects not including the element. This element is often called a *partitioning element.* To enumerate a set composed of permutations, we often partition the original set to two sets of those whose kth index is i, and those whose kth index is not i. The scheme does only partitioning the problem, and needs neither complicated algorithm nor special data structure. The required memory space is usually not so large. Many binary partition algorithms requires memory space at most proportional to the input size. The feature also helps us to construct useful algorithms easily.

Let us show an enumeration algorithm as an example of a binary partition algorithm. We consider the problem of enumerating all spanning trees in an undirected graph. For this problem, we have a simple binary partition algorithm [44]. For a given undirected graph G, the algorithm selects an edge e of G satisfying that some spanning trees includes e and some spanning trees does not include e. If no such edge exists, then there is at most one spanning tree. The algorithm stops in this case. By using e, the algorithm generates two subproblems of enumerating all spanning trees including e and all these not including e. To solve the subproblems, the algorithm generates two graphs G/e and $G \setminus e$, which are obtained by contracting e and removing e. For any spanning tree in G/e, we can obtain a spanning tree of G by adding e to the tree. Conversely, for any spanning tree of G including e, the tree obtained by contracting e from the tree is a spanning tree of G/e. Therefore we can enumerate all spanning trees including e by solving the subproblem with G/e. Similarly, we can also enumerate all spanning trees not including e by solving the subproblem with $G \setminus e$. Thus all spanning trees can be enumerated by solving the subproblems recursively. Hence we can construct a polynomial delay algorithm for solving the problem.

In this example, e partitions the original problem efficiently, and both subproblems can be constructed easily. Since the algorithm always generates two subproblems if the given problem includes more than one object, the number of iterations is bounded by twice the number of outputting objects. Thus the time complexity of the algorithm is linear in the output size, and polynomial in the input size if an iteration takes polynomial time. Therefore keys to construct efficient binary partition algorithms are how to find the "partitioning element" and how to generate "simple subproblems". Some binary partition algorithms are shown in [18, 19, 23].

To show the movement of an enumeration algorithm, we often use a virtual tree called an *enumeration tree*, which expresses the structure of the recursive calls. We often show the correctness, the time complexity, the movement, and the other properties of the algorithm by using the tree. For a given binary partition algorithm and its input, let $\mathcal{V}$ be a vertex set whose elements correspond to all recursive calls occurring in the algorithm. We consider an edge set $\mathcal{E}$ on $\mathcal{V} \times \mathcal{V}$ such that each edge connects two vertices if and only if a recursive call corresponding to one of the vertices occurs in the recursive call corresponding to the other. Since the structure of a recursive algorithm contains no circulation, the graph $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ forms a tree. This tree is called the enumeration tree of the algorithm. The root vertex of the tree corresponds to the start of the algorithm. Since a binary partition algorithm generates subproblems if and only if the given subproblem includes more than one outputting object, all internal vertices of the tree have at least two children, and all leaves of the tree correspond to all outputting objects. To analyze enumeration algorithms, we show some properties of enumeration trees which satisfies for any input. By using these properties, we often bound the time complexity.

Since the above definition of the enumeration tree is for binary partition algorithms mainly, we have some other definitions for the other kinds of enumeration algorithms. The second scheme "reverse search" utilizes such enumeration trees defined by the other way. To construct a reverse search algorithm, we define "parents" for all objects except a specified one. The parents are defined uniquely for each object. We introduce a parent-child relationship among all objects by these parents. The parent-child relationship has to satisfy that each object is not a proper ancestor of itself. Under the condition, the graph representation of the relationship forms a tree. The tree is also called the enumeration tree. Reverse search traverses the tree in the depth-first search manner, and outputs all objects in the order of visiting.

Generally, the size of the enumeration tree is quite huge, hence usually it is hard to store the whole of the tree in memory space. Reverse search traverses enumeration trees, although it requires not so much memory space. A depth-first search is not in need of the whole tree in memory space, but only the current visiting vertex. In each iteration, a reverse search algorithm lists all children of the vertex, and continues the traverse by moving to the children as the same manner of depth-first search. Therefore, to obtain a reverse search algorithm, we are in need to construct only algorithm for finding all children of a vertex of the enumeration tree. If we construct the depth-first search without recursive calls, the memory complexity of a reverse search algorithm is depends on only the input size.

Now let us see an example of a reverse search algorithm [3]. The problem is to enumerate all feasible dictionaries of a polytope given by a set of linear inequalities. To define the parent-child relationship among all feasible dictionaries, we use the pivoting rule of a simplex method. By giving a certain objective function, we make a linear programming problem on the polytope with exactly one optimal solution. For any feasible base x, the simplex method make a pivot moving to the other feasible dictionary y. By using this rule, we define the parent of x by y. Since simplex methods generate no circulation of pivots, the parent-child relationship forms a tree. The reverse search generates all children of a feasible dictionary x by finding all dictionaries adjacent to x, and checking whether their parents are x or not by constructing its parent with the pivoting rule. Since the number of dictionaries adjacent to x is polynomial in the input size, we can obtain a polynomial delay algorithm for enumerating all feasible dictionaries.

The performance of a reverse search algorithm is deeply depend on the parent-child relationship. If we have to spend much time to list all children of the current visiting object, the time complexity will be huge. Some complicated algorithms for listing all children also make difficult us to implement the algorithm. The total time complexity will be reduced by some efficient algorithms with good parent-child relationships. Some reverse search algorithms are proposed in [3, 48, 53, 56].

In the following sections, we show our four enumeration algorithms utilizing data structures. The first algorithm is for enumerating spanning trees [49]. The algorithm achieves the optimal time and space complexities in the sense of the worst case time complexity. It is obtained by modifying the algorithm of A. Shioura and A. Tamura [48]. Their algorithm is optimal in the time complexity but not in the memory complexity. The improved part of the algorithm is its data structures. Their algorithm reconstructs their data structures at the end of each iteration. To reconstruct the data structures, they store some informations in memory space. It is the bottle neck part of the memory complexity. In our algorithm, we store the informations in the other manner, and success reducing the memory complexity from $\mathrm{O}(nm)$ to $\mathrm{O}(m+n)$.

The second algorithm is for enumerating directed spanning trees [53]. Our algorithm is an improved version of the algorithm of H. N. Kapoor and H. Ramesh [32]. The algorithm spends $\mathrm{O}(n)$ time to update its data structures in each iteration. It is the bottle neck part of the time complexity. Our algorithm is based on reverse search and their data structures. To update the data structure quickly, our algorithm has one more new data structure which utilizes the algorithm of maintaining a minimum spanning tree of an undirected graph. Our algorithm updates the data structure in $\mathrm{O}(n^{1/2})$ time, thus the time complexity is reduced to the order. In the case of that the given graph is planar, the updating is done in $\mathrm{O}(\log^3 n)$ time, and our algorithm runs in $\mathrm{O}(\log^3 n)$ time per a directed spanning tree.

The last algorithm is for enumerating common intervals of two permutations [57]. For the problem, a simple algorithm was proposed in [58]. It terminates in $\mathrm{O}(n^2)$ running time, which is not depend on the output size. We improve this algorithm by using a data structure with linked lists. It saves the time for unnecessary search operations, and reduces the time complexity to the optimal $\mathrm{O}(n+N)$ time where N is the number of common intervals.

The last one is for distance preserving rectilinear steiner trees. The algorithm utilizes a reverse search. This reverse search may require much computation time for an iteration with only naive implementation. By using some linked lists, we reduce the time complexity to $\mathrm{O}(1)$ per a distance preserving tree. It is optimal in the sense of time complexity. In existing studies, no algorithm is proposed for this problem.

3.1 Enumerating Spanning Trees in Undirected Graphs

The problem of enumerating all spanning trees in an undirected graph is the most major one among various enumeration problems. Many algorithms have been proposed for the problem. In 1975, R. C. Read and R. E. Tarjan [44] proposed a polynomial delay algorithm. Its time and space complexities are $O(m + n + mN)$ and $O(m + n)$. Here N denotes the number of spanning trees in the given graph. The algorithm is based on binary partition. In 1978 H. N. Gabow and E. W. Myers [23] improved the algorithm. Their improvements reduce the time complexity from $O(m+n+mN)$ to $O(m+n+nN)$. In 1992, H. N. Kapoor and H. Ramesh [32] proposed an algorithm running in $O(n + m + N)$ time and $O(nm)$ memory with the use of compact output method. The time complexity is optimal in the sense of worst case analyzing, although the space complexity is not so. Recently, A. Shioura and A. Tamura [48] proposed an algorithm with reverse search. Its time and space complexities are same to the algorithm of Kapoor and Ramesh, although their algorithm utilizes data structures composed of only simple linked lists.

In this section, we propose an algorithm for enumerating all spanning trees in a given undirected graph $G = (V, E)$. Our result is that we reduce the memory complexity from $O(nm)$ to $O(m+n)$ without loss of the optimality of the time complexity. By introducing a new data structure, we modify the algorithm of Shioura and Tamura [48]. Since the algorithm of Shioura and Tamura utilizes reverse search, we firstly explain their reverse search and its parent-child relationship in the next subsection.

3.1.1 A Reverse Search Algorithm

In this subsection, we show the parent-child relationship of Shioura and Tamura. To define their parent-child relationship, we introduce some notations. Let T_0 be a DFS tree starting from a specified vertex r. We put indices on the vertices of the tree by the order of visiting of the search. To see the detailed properties of DFS trees and their indices, refer chapter 2 "Preliminaries." For an edge, we call its endpoint the *head* if it has an index larger than the other endpoint, and the *tail* otherwise. We define the index of an edge by the index of its head. All edges of T_0 have distinct indices. For two edges e and f in a tree, we say that e is an ancestor of f if e is on the path from r to the head of f. The *parent edge* of an edge e in T is the ancestor edge which is adjacent to e. For a spanning tree $T \neq T_0$, let $e^*(T)$ be the minimum index edge among $T_0 \setminus T$. We call vertices and edges of T with smaller indices than $e^*(T)$ *valid*. Exceptionally for T_0, we call all edges of T_0 *valid edges*. The tail of any valid edge is a valid vertex. For an edge e of T, let $Des(T, e)$ denote the set of vertices which are descendants of the tail of e, and $Anc(T, e)$ denote the set of vertices which are ancestors of the head of e.

We introduce the parent-child relationship by using T_0 and e^*. For a spanning tree $T_c \neq T_0$, let C be the cycle obtained by adding $e^*(T_c)$ to T_c. We define the parent T_p of T_c by the spanning tree obtained by adding $e^*(T_c)$ to T_c and removing an edge $e \in C \setminus T_0$ incident to a vertex of $Anc(T_c, e^*(T_c))$ (see Figure 3.1). The parent satisfies the following condition.

Lemma 1 *For any spanning tree $T_c \neq T_0$, e and the parent are uniquely defined.*

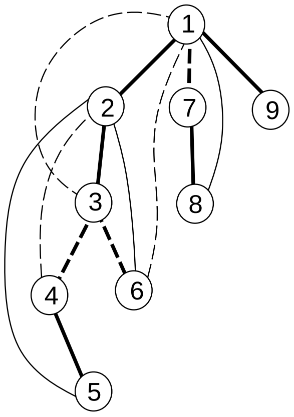

Figure 3.1: The parent of a spanning tree: Emphasized straight lines are edges of T_0. Dotted lines are non-tree edges. We obtain the parent of the tree by removing the edge (2, 5) and adding the edge (3, 4).

Proof : Let P be the path from r to the head of $e^*(T_c)$ in T_0. Since T_c includes all edges of T_0 with indices less than $e^*(T_c)$, T_c includes P. Thus any vertex in $Anc(T_c, e^*)$ is in P. Let e be an edge in $C \setminus P$ which is incident to a vertex of $Anc(T_c, e^*(T_c))$. e is unique, since if there are two such edges, then T_c includes two distinct paths connecting endpoints of them. If e is in T_0, then there is a path from the head of $e^*(T_c)$ to a vertex not in $Des(T_0, e^*(T_c))$. It contradicts that T_0 is a DFS tree. Hence e is uniquely defined. The parent T_p of T_c is also uniquely defined. ▮

Since T_p includes exactly one more edge of T_0 than T_c, T_c never be an ancestor of itself. We can obtain an enumeration tree from this relationship. A reverse search algorithm traverses the enumeration tree by finding all children of the current traversing spanning tree T_p. Next, let us see the method for finding all children of T_p.

For any child of T_p, its parent T_p is obtained by removing an edge e and adding an edge f. e is a valid edge of T_p. The head of f is in $Anc(T_p, e)$. Therefore any child of T_p is obtained by removing a valid edge e and adding an edge f incident to a vertex of $Anc(T_p, e)$.

For a valid edge e of T_p, we call an edge whose endpoints are in $Des(T_p, e)$ and $Anc(T_p, e)$ an *entering edge* of e. We denote the set of entering edges of a valid edge e in T_p by $Entr(T_p, e)$. Let T_c be a spanning tree obtained by removing a valid edge e and adding an edge f of $Entr(T_p, e)$ to T_p. Since any edge of T_0 with an index less than e is included in T_c, e is $e^*(T_c)$. Hence T_c is a child of T_p. Conversely, for any child $T_c = T_p \setminus e \cup f$, f is an entering edge of e. We obtain a child by exchanging a valid edge and its entering edge, hence a pair of a valid edge e and an edge of $Entr(T_p, e)$ corresponds to a child of T_p one-to-one.

By using these definitions and properties, we can easily construct a polynomial delay algorithm for enumerating all spanning trees in a given undirected graph. A naive implementation takes much time, hence we utilize some data structures to speed up. Our algorithm utilizes data structures of A. Shioura and A. Tamura to reduce the time complexity. Let us see their data structures in the following subsection.

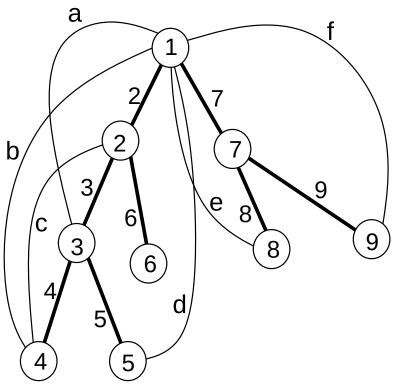

Figure 3.2: A spanning tree T_0: $Entr(T_0, e_3) = \{a, b, c, d\}$, $Entr(T_0, e_4) = \{b, c\}$, $Entr(T_0, e_5) = \{d\}$, and $Can(e_3; T_0, 4) = \{a, d\}$.

3.1.2 A Simple Data Structure

This subsection explains the data structures of Shioura and Tamura. In our algorithm, we set k to the index of $e^*(T_p)$ minus one. We obtain all edges of $Entr(T_p, e_k)$, and generate all children by exchanging e_k and an entering edge of e_k. After finding all children obtained by exchanging e_k, we decrease k by one, and generate children similarly until $k = 0$.

To obtain $Entr$ for a valid edge, we spend $\Theta(m+n)$ in the worst case by naive algorithms. The computation time is not short efficiently, hence we decompose $Entr$ into some edge sets, and maintain the edge sets in each iteration. The edge sets are defined as follows. For T_p, an index k and edge e_j, $j \leq k$, we define $Can(e_j; T_p, k)$ by

$$Can(e_j; T_p, k) = Entr(T_p, e_j) \setminus \bigcup_{h=j+1}^{k} Entr(T_p, e_h).$$

Suppose that i is the largest index satisfying that $Entr(T_p, e_i)$ includes e. e is included only in $Can(e_i; T_p, k)$ among all $Can(e_j; T_p, k)$ for any $k \geq j$. For a non-tree edge f, f is included in $Entr(T_p, e)$ for any edge e in the cycle obtained by adding f to T_p. Therefore for any T_p and k, any non-tree edge is included in a $Can(e_j; T_p, k)$, and any $Can(e_j; T_p, k)$ does not share its elements with the others (see Figure 3.2).

Since $\sum_{j \leq k} |Can(e_j; T_p, k)| \leq m$ for any T_p and k, hence we can update them in shorter time than $Entr$. To find all children of T_p, we set k to the index of $e^*(T_p)$ minus one, and obtain $Can(e_j; T_p, k)$ for all e_j. Since $Can(e_k; T_p, k) = Entr(e_k; T_p)$, we can find all children obtained by exchanging e_k and each edge of $Entr(e_k; T_p)$. After finding them, we decrease k by one, and update all $Can(e_j; T_p, k)$ to obtain $Can(e_j; T_p, k-1)$. The updating operation is based on the following lemma.

Lemma 2 *For any k, if e_j is the parent edge of e_k, then $Can(e_j; T_p, k-1)$ is obtained by removing all edges whose both endpoints are in $Des(T_p, e_k)$ from $Can(e_j; T_p, k) \cup Can(e_k; T_p, k)$. If e_j is not the parent edge of e_k, $Can(e_j; T_p, k-1) = Can(e_j; T_p, k)$.*

Proof : From the definition, we have

$$Can(e_j; T_p, k-1) \subseteq Can(e_j; T_p, k) \cup Can(e_k; T_p, k).$$

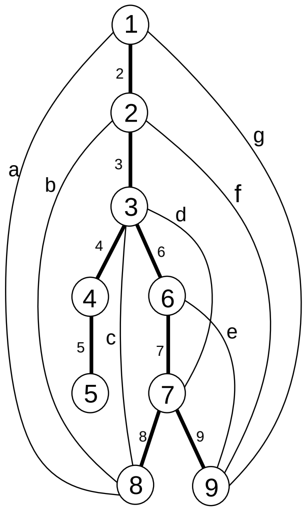

Figure 3.3: A spanning tree T and the changes of Can: For $k = 7$, all non-tree edges are included in $Can(T, e_7, k)$. By decreasing k by one, the edges except e move to $Can(T, e_6, k)$. In the tree T' obtained by exchanging e_7 and the edge b, the edges a, f and g move to $Can(T', e_2, k-1)$.

Hence we consider which $Can(e_j; T_p, k-1)$ includes the edges of $Can(e_k; T_p, k)$. Since any edge e of $Can(e_k; T_p, k)$ connects a vertex of $Anc(T_p, e_k)$ and a vertex of $Des(T_p, e_k)$, e is included in $Can(e_j; T_p, k-1)$ for some ancestor edges of e_k. If the tail of e is in $Anc(T_p, e_p)$, e is included in $Can(e_p; T_p, k-1)$ since the parent edge e_p of e_k has the largest index among ancestor edges of e_k. Otherwise e is included in no $Can(e_j; T_p, k-1)$ (see Figure 3.3). ∎

When we find a child T_c of T_p, we generate a subproblem of enumerating all descendants of T_c, and solve the subproblem by a recursive call. In the subproblem, we are in need of $Can(e_j; T_c, k-1)$ for all e_j, $j \leq k-1$. Note that the index of $e^*(T_c)$ is k. To obtain them, we also update $Can(e_j, T_p, k)$. Suppose that T_c is obtained by exchanging an edge f and e_k. To show the updating operation, we state the following lemma.

Lemma 3 *We denote the valid edge sharing its head with f by e_j. $Can(e_j; T_c, k-1)$ is obtained by removing all edges satisfying that the indices of their tails are larger than the tail of e_j from $Can(e_j; T_c, k) \cup Can(e_k; T_p, k)$. For the other valid arcs e', $Can(e'; T_c, k-1) = Can(e'; T_p, k)$.*

Proof: Firstly, we see the changes of $Entr$ by the exchange. For an ancestor edge e' of f in T_c, a vertex of $Des(T_p, e')$ in T_p is also in $Des(T_c, e')$. The condition also holds for any valid edge which is not an ancestor edge of e' in T_p. Hence we have $Entr(T_c, e') = Entr(T_p, e')$. For ancestor edges of e' which are not ancestor edges of f, the condition may not hold. Let D be the set of all edges incident to vertices of $Des(T_p, e_k)$. Then $Entr(T_p, e') \setminus D = Entr(T_c, e')$ holds for any ancestor edge e' of e_k which is not an ancestor edge of f. Hence we have $Can(e'; T_p, k) = Can(e'; T_c, k-1)$. Let e_j be the parent edge of f in T_c. Since the above equation holds for any edge one of whose ancestor edge is e_j, $Can(e_j; T_c, k-1)$ is obtained by removing all edges whose both endpoints are not in $Anc(T_p, e_j)$ from $Can(e_j; T_c, k) \cup Can(e_k; T_p, k)$ (see Figure 3.3). ∎

From these lemmas, we can update Can by only moving edges from $Can(e_k; T_p, k)$ to $Can(e_j; T_p, k-1)$ or $Can(e_{j'}; T_c, k-1)$ for certain edges e_j and $e_{j'}$. To find the edges to be moved easily, we sort all the elements of $Can(e_j; T_p, k)$ in the order of the indices of their tails. Note that the tails are in $Anc(T_p, e_j)$. By this, we can find all edges of Can to be moved by tracing the sorted edges from the minimum one. We merge the edges of C and the edges to be moved. It takes $O(|Can(e_j; T_p, k-1)|)$ or $O(|Can(e_{j'}; T_c, k-1)|)$ time. For any edge of $Can(e_j; T_p, k-1)$, we have a child of T_p obtained by exchanging e_j and the edge. Hence all the children of T_p obtained by exchanging e_k and the other edges can be found in the linear time in $|Entr(e_j; T_p)| + |Entr(e_{j'}; T_p)|$. We take at least $\Theta(1)$ time to find all children obtained by exchanging e_k. Hence, we can see that the sum of updating time of Can from k through 1 is linear in the number of children of T_p plus k. For any edge of $Can(e_{j'}; T_c, k-1)$, we have a child of T_c obtained by exchanging $e_{j'}$ and the edge. Hence we spend linear time in the number of children of T_c to obtain $Can(e_i; T_c, k-1)$ from $Can(e_i; T_p, k)$ for all i.

In the case that T_p has few children, the algorithm will take $\Omega(n)$ time to find all children of T_p. This will be the bottle neck part of the time complexity. Hence we have a linked list `leave` to reduce the computation time. `leave` avoids valid edges e_i with $Can(e_i; T_p, i) = \emptyset$. `leave` is composed of all valid edges satisfying $Can(e_i; T_p, i) \neq \emptyset$. All the elements of `leave` are stored in the order of their indices. For any $i < k$, $Can(e_i; T_p, k) = Can(e_i; T_p, k-1)$ holds if $Can(e_k; T_p, k) = \emptyset$ (see Figure 3.4). Thus if $Can(e_i; T_p, k) = \emptyset$ holds for any $i, k' \leq i \leq k$, then $Can(e_i; T_p, i) = \emptyset$ holds for any $i, k' \leq i \leq k$. Hence we can skip the iteration with T_p and i for all $i, k' \leq i \leq k$, and start the iteration with $i = k' - 1$. We extract the valid edge with the largest index from `leave` in each iteration, and find all children obtained by exchanging the edge. After finding all these children, we update `leave`.

By updating Can, we may have to update `leave`. A deletion of an edge from the `leave` is done in $O(1)$ time, hence we consider the time to insert an edge to `leave`. In the case that some edges of $Can(e_k; T_p, k)$ are moved to $Can(e_{k-1}; T_p, k-1)$, e_{k-1} is inserted in `leave`. The index of e_{k-1} is largest among all valid edges, thus e_{k-1} is always at the end of the `leave`. This insertion is also done in $O(1)$ time. When we generate a subproblem with a child T_c of T_p, some edges of $Can(e_k; T_p, k)$ are moved to $Can(e_j; T_p, k-1)$. In the case, we trace `leave` to find the position which we insert e_j. Hence the insertion takes linear time in the number of elements in the `leave` whose indices are smaller than j. Since T_c has children obtained by exchanging any edge of `leave`, this computation time is bounded by the linear time in the number of children of T_c.

When the enumeration of all descendants of T_c is done, we restore Can and `leave`, and find the other children of T_p. To restore them, we utilize some techniques because naive techniques take much time to restore them. To recover `leave`, we record the extracted and inserted edges to `leave` in each iteration. By using these records, we can restore `leave` in the time proportional to the time to update `leave`. Since the algorithm extract the maximum index edge e from `leave` and insert edges with indices less than e to `leave`, the total size of the records is $O(n)$ in any iteration.

Next we see how to restore Can after enumerating all descendants of T_c. When the recursive call with T_c terminates, we have only $Can(e_1; T_c, 1)$. Hence, we firstly restore all $Can(e_i; T_c, h)$

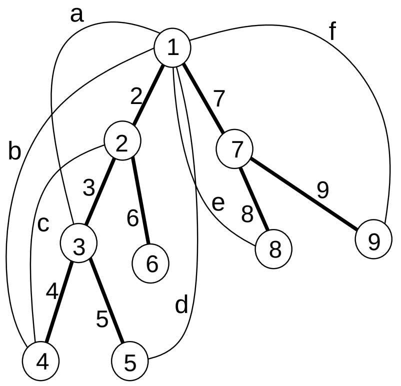

Figure 3.4: A spanning tree T_0: for $k = 4$, $Can(e_2; T_0, 4) = \emptyset$, $Can(e_3; T_0, 4) = \{a, d\}$ and $Can(e_3; T_0, 4) = \{b, c\}$. `leave` is composed of e_3 and e_4. The `head list` of e_3 is composed of vertices with indices 3 and 5. The `head list` of e_4 is composed of a vertex with the index 4.

from $Can(e_i; T_c, h-1)$ iteratively, and obtain $Can(e_i; T_p, k)$ from $Can(e_i; T_c, k-1)$ where k is the index of $e^*(T_c)$ minus one. The algorithm of Shioura and Tamura records all changes of Can in each iteration to restore Can. Since the number of edges to be moved in an iteration may be up to $O(m)$, the total required memory space is $O(nm)$. It is the bottle neck part of the memory complexity of their algorithm. In our algorithm, we utilize one more data structure based on linked lists. In next subsection, we show the details of the data structure.

3.1.3 A Technique for Reducing the Space Complexity

In this subsection, we show our data structure called `head lists` which are linked lists composed of heads of edges. For a spanning tree T_p and an index k, a `head list` of $Can(e; T_p, k)$ is composed of all heads of the edges in $Can(e; T_p, k)$. For T_0 and $k = n$, the heads of two edges of $Can(e_i; T_0, k)$ and $Can(e_j; T_0, k)$ are distinct. We move edges of $Can(e_k; T_p, k)$ to the other Can in any iteration. Hence the heads of two edges of $Can(e_i; T_p, k)$ and $Can(e_j; T_0, k)$ are always distinct. All `head lists` have distinct components (see Figure 3.4). The sum of lengths of `head lists` for all edges of T_0 does not exceed n in all iterations of the algorithm.

By looking `head lists` of $Can(e_k; T_p, k)$, we can identify which edges of $Can(e_j; T_c, k-1)$ come from $Can(e_k; T_c, k)$ in $O(|Can(e_j; T_c, k-1)|)$ time. It does not exceed the time to update Can. When we update or restore Can, we also have to update and restore `head lists`. We can easily update `head lists`, but can not restore it so easily. Hence, we use some techniques to update `head lists`. When we update a `head list`, we mark the all heads in the `heads list` of $Can(e_k, T_p, k)$ which leave from $Can(e_k, T_p, k)$. We connect two consecutive marked heads by looking all the heads adjacent to marked heads in the `head list`. The `head list` is partitioned into some sublists composed of only marked heads, and only unmarked heads. We append the sublists composed of marked heads to the `head list` of $Can(e_j; T_p, k)$. To restore them, we store two pointers indicating both ends of these sublists on e_k. When we restore the `head list` of e_k, we take these sublists from the `head`

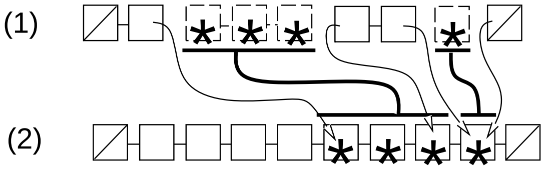

Figure 3.5: The movements of maximal sublists of a `head list` (1). Marked heads move to the other `head list` (2). Some unmarked heads has pointers indicating the moved sublists.

`list` of e_j (see Figure 3.5).

Since the numbers of sublists and marked heads do not exceed the number of the moved edges in $Can(e_k, T_p, k)$, the updating and restoring time of `head lists` does not exceed the updating time of Can. The number of sublists does not exceed the number of sublists composed of unmarked heads. Hence the total number of the above pointers indicating ends of these sublists is $O(n)$.

Now we describe the details of the algorithm in the following.

```
ALGORITHM: ENUM_SPANNING_TREES( G )
Step 1: Find a depth-first search tree $T_0$.
Step 2: Set $Can(e_i; T_0, n)$ to all the edges not in $T_0$ which share their heads with $e_i$.
Step 3: Insert all edges $e$ of $T_0$ in leave satisfying that $Can(e; T_0, n) \neq \emptyset$.
Step 4: For any edge $e$ in leave, insert the head of $e$ in $Can(e; T_0, n)$ to the head lists of $e$.
Step 5: Call ENUM_SPANNING_TREES_ITER( $T_0$ )
.
```

```
ALGORITHM: ENUM_SPANNING_TREES_ITER( $T_p$ )
Step 1: Extract the edge $e_k$ with the maximum index from leave. If leave is empty,
        go to Step 14.
Step 2: For each edge $f$ of $Can(e_k; T_p, k)$ do
Step 3:     Generate a child $T_c$ by exchanging $e_k$ and $f$.
Step 4:     Move edges of $Can(e_k; T_p, k)$ whose tails have indices smaller than $k$ to the
            parent edge $g$ of $f$ on $T_c$.
Step 5:     If some edges are moved and $g$ is not in leave, insert $g$ to leave.
Step 6:     Update the head list of $g$.
Step 7:     Call ENUM_SPANNING_TREES_ITER( $T_c$ ).
Step 8:     Recover head lists, leave and $Can$.
Step 9: End do
Step 10: Move edges of $Can(e_k; T_p, k)$ whose tails have indices smaller than $k$ to the
         parent edge $h$ of $e_k$.
Step 11: If some edges are moved and $h$ is not in leave, insert $h$ to leave.
Step 12: Update the head list of $h$.
Step 13: Go to Step 1.
```

Step 14: Recover `head lists`, `leave` and Can iteratively.

Theorem 1 *The algorithm* ENUM_SPANNING_TREES *enumerates all spanning trees in an undirected graph in $O(m + n + N)$ time and $O(m + n)$ memory where N is the number of spanning trees in the graph.*

Proof : We already see the correctness and the space complexity of the algorithm, thus we show the time complexity. The algorithm ENUM_SPANNING_TREES terminates $\mathrm{O}(m + n)$ time. The time complexity of an iteration of ENUM_SPANNING_TREES_ITER (T_p) is proportional to the number of children and grandchildren of T_p. In the enumeration tree, a vertex is a child of only one vertex, and also a grandchild of only one vertex. Hence the sum of the number of children and grandchildren over all vertices is at most $2N$. Therefore the total time spent to enumerate all spanning trees is proportional to N. ∎

3.2 Enumerating Directed Spanning Trees in Directed Graphs

The second algorithm in this chapter is for directed spanning trees. For a directed graph $G = (V, A)$ and a root vertex r, we consider the problem of enumerating all directed spanning trees in G whose root vertices are r. Directed spanning trees have been studied in many fields. For instance, many problems on road and telephone networks have been formulated as some optimization problems of directed spanning trees. For some decision making problems, directed spanning trees often express some possible choices. The enumeration is practical for optimization on these problems.

The enumeration problem of directed spanning trees has been studied for nearly 20 years. In 1978, H. N. Gabow and E. W. Myers [23] proposed an algorithm which runs in $\mathrm{O}(m+n+(m+n)N)$ time and $\mathrm{O}(m+n)$ space. Here N denotes the number of directed spanning trees in G. The algorithm is based on binary partition. It finds an arc which is included and not included in some directed spanning trees, and partitions the problem into two subproblems. In each iteration, it spends $\mathrm{O}(m+n)$ time to find the arc, thus the time complexity is $\mathrm{O}(m+n)$ per a directed spanning tree. In 1992, H. N. Kapoor and H. Ramesh [32] improved the time complexity to $\mathrm{O}(mn++nN)$. Their improvement utilizes some data structures. The data structures make us be enable to obtain the above arc in $\mathrm{O}(n)$ time. Their improvement does not lose the optimality of memory complexity.

Since a directed spanning tree requires $\mathrm{O}(n)$ time for outputting, the algorithm is the optimal algorithm in the sense of both time and space complexities if we have to output all of them explicitly. On the other hand, in undirected graphs, *compact output methods* have been studied to shorten the output size [32, 49]. They output a spanning tree by the differences between it and the previous outputted tree. The total size of outputs is reduced to $\mathrm{O}(N)$ by the method with a certain order of spanning trees.

For the problem of undirected spanning trees, A. Shioura, A. Tamura and T. Uno proposed an optimal algorithm [49]. They utilizes reverse search and compact output method. Our algorithm is based on their reverse search and compact output method. Our algorithm is obtained by applying their techniques to directed spanning trees, and uses the data structures of Kapoor and Ramesh. The movement of our algorithm is quite similar to the algorithm of Kapoor and Ramesh, since the structure of the enumeration trees and the orders of outputting of these algorithms are same. Thus our reverse search algorithm can be also considered as a kind of binary partition method which is used in [32]. Hence our algorithm can be considered as an improved version of the algorithm of Kapoor and Ramesh.

Our improvements of the algorithm is on data structures. Recently, there have been proposed some good data structures for spanning trees especially their updating. For example, we have some fast algorithms and data structures for updating a minimum weighted spanning tree. By using these good data structures, we improve the existing algorithms. The improvements reduce the time complexity to $\mathrm{O}(m+n+N\mathrm{D}(n,m))$. Here $\mathrm{D}(n,m)$ denotes the time to update a minimum spanning tree of an undirected graph for a change on its edges.

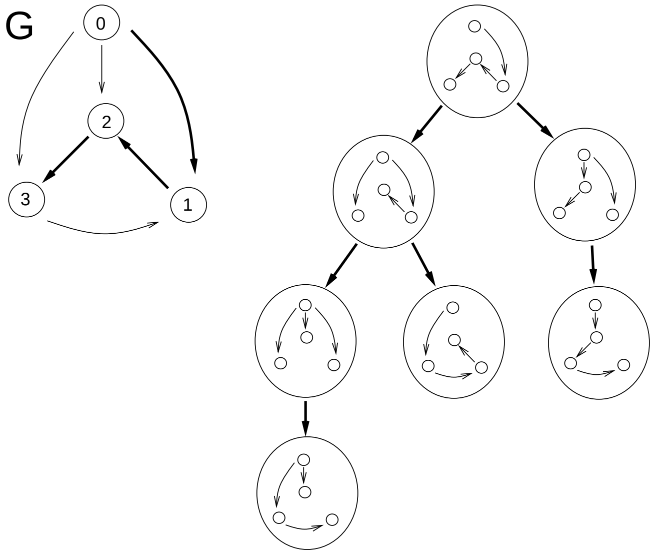

Figure 3.6: The enumeration tree. The emphasized line of the graph G is T_0.

Our algorithm constructs and preserves an undirected graph with n vertices and at most m weighted edges. This graph is modified after each directed spanning tree is encountered. To update the data structure for finding new directed spanning trees, we delete, add or change the weight of an edge, and find a minimum spanning tree of the graph in each iteration of the algorithm. Since only few edges are changed in each iteration, we construct a minimum spanning tree of the changed graph from the previous one by some spanning tree updating algorithm.

Therefore the time complexity of our algorithm depends on the time complexity of the updating algorithm. The space complexity also depends on it. We denote its space complexity by $\mathrm{DS}(n, m)$. The update operations are adding, deleting and changing the weight of an edge. Some data structures are proposed for these updating operations of the minimum spanning tree. G. N. Frederickson proposed an $\mathrm{O}(m^{1/2})$ time data structure [16], and D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig improved the time complexity into $\mathrm{O}(n^{1/2} \log(m/n))$ [15]. Recently, they have also improved their data structure, and the time complexity is reduced to $\mathrm{O}(n^{1/2})$. In the case of planer graphs, the time complexity of their data structure can be reduced to $\mathrm{O}(\log^3 n)$. Their time complexities are smaller than $\mathrm{O}(n)$. All these algorithms take only $\mathrm{O}(m+n)$ time for preprocessing, thus the time complexity of our algorithm is $\mathrm{O}(m+n+n^{1/2}N)$ in general graphs and $\mathrm{O}(n+(\log^3 n)N)$ in planer graphs. These data structures use only $\mathrm{O}(m+n)$ space, thus we do not lose the optimality of space complexity by our improvement.

3.2.1 A Reverse Search Algorithm and Parent-Child Relationship

Our algorithm is based on reverse search. Before describing the data structures of our result, we show our reverse search and its parent-child relationship.

Let T_0 be a depth-first search tree of the given directed graph G. In the relationship, T_0 corresponds to the root of the enumeration tree. We put indices to all vertices of G in the order of the traversal of the depth-first search. Similarly, we define the indices of arcs of G by indices of their heads. For a directed spanning tree $T \neq T_0$, let $e^*(T)$ be the minimum index arc in $T_0 \setminus T$. The parent T_p of a directed spanning tree $T_c \neq T_0$ is defined with these indices. The parent T_p is given by the directed spanning tree $T_c \setminus f \cup e^*(T_c)$ where f is the arc of T_c sharing its head with $e^*(T_c)$. Since $e^*(T_c)$ shares its head only one arc of T_c, T_p is uniquely defined. From the definition of the index, there always exists a directed path from r to the tail of $e^*(T_c)$ in T_p. Hence $e^*(T_c)$ generates no cycle in T_p. Therefore T_p forms a directed spanning tree. In this parent-child relationship, T_c is not a proper ancestor of itself, as T_p contains just one more arc of T_0 than T_c. Since these facts hold, the relationship gives an enumeration tree. The algorithm traverses the enumeration tree in the depth-first search manner, and output all directed spanning trees. We show an example of an enumeration tree in Figure 3.6.

To output directed spanning trees compactly, we consider differences between a directed spanning tree and the one outputted just before by the reverse search. Some of them may have $O(n)$ differences, but the total differences over all directed spanning trees is not up to $O(nN)$. Since our reverse search traverses the enumeration tree in the depth-first search manner, the total amount is twice the sum of differences between a child and its parent over all directed spanning trees. Any directed spanning tree differs by only two arcs from its parent, thus the total the size of outputs can be reduced into $O(N)$ by outputting only differences.

Next we show the method of finding all children of a directed spanning tree T_p. Let us construct T_c by removing an arc e from T_p whose index is less than $e^*(T_p)$ and adding an arc $f \neq e$ sharing its head with e. If T_c forms a tree, it is a child of T_p from the definition. Conversely, in the case that $T_p = T_c \setminus f \cup e$ is the parent of T_c, the index of e is less than $e^*(T_p)$. Thus all children of T_p can be found by the above method. We deal with only vertices and arcs with smaller indices than $e^*(T_p)$ to find children, hence we call them *valid.* Exceptionally for T_0, we call all arcs and vertices valid.

If f is a back-arc of T_p in the above method, then $T_c = T_p \setminus e \cup f$ contains a cycle and is not a directed spanning tree. If f is a non-back arc of T_p, there always exists a directed path from r to the tail of f in T_c, thus T_c contains no cycle. Therefore each child of T_p is obtained by adding a valid non-back arc f and removing an arc e sharing its head with f. Valid non-back arcs have a one-to-one correspondence with the children, thus the number of children is same as the number of valid non-back arcs. By maintaining the set of all valid non-back arcs, finding all children can be accomplished sufficiently easily.

We now describe the framework of our algorithm.

```
ALGORITHM: ENUM_DIRECTED_SPANNING_TREES($G$)
Step 1: Find $T_0$ by a depth-first search.
Step 2: Put indices for each vertex.
Step 3: Classify arcs not in $T_0$ into the back arc set and the non-back arc set.
        Sort them in the order of their indices.
```

Step 4: Call ENUM_DIRECTED_SPANNING_TREES_ITER(T_0).

```
ALGORITHM: ENUM_DIRECTED_SPANNING_TREES_ITER($T_p$)
Step 1: For each valid non-back arc $f$ of $T_p$ do
Step 2:     Construct $T_c$ by adding $f$ and removing an arc $e$.
            Output it by the difference from the previous outputted one.
Step 3:     List all valid non-back arcs of $T_c$ in the order of their indices.
Step 4:     Call ENUM_DIRECTED_SPANNING_TREES($T_c$) recursively.
Step 5: End for
```

Step 1 to 3 of the algorithm ENUM_DIRECTED_SPANNING_TREES(G) runs in O($m + n$) time. The construction of a child in Step 2 of ENUM_DIRECTED_SPANNING_TREES_ITER is done in O(1) time by exchanging two arcs. Step 3 lists all valid non-back arcs of T_c, and the number of them is equal to the number of children of T_c. Thus the total time spent until Step 3 is linear in the time to find one valid non-back arc of T_c per one outputted directed tree.

The algorithm in [32] takes O(n) time for finding one non-back arc in the worst case. It is the bottle neck of the time complexity while the other parts of the algorithm take only O(1) time. Our improvement on this part reduces its time complexity. We find these non-back arcs in shorter time by using a data structure for maintaining an undirected minimum spanning tree. In the next subsection, we show a data structure which is the key to the improvement.

3.2.2 An Improved Data Structure for Finding Non-Back Arcs

To enumerate all valid non-back arcs within sufficiently short time in Step 3, we show the following two conditions. They are also stated in [32]. The followings consider the case that T_c is constructed from T_p by removing an arc e and adding an arc f. We denote the head of e by v, and the nearest common ancestor of v and the tail of f by w. Let P be the vertex set of the interior points of the path from w to the tail of e in T_p. Here interior points of the path are vertices of the path which are not its endpoints.

Lemma 4 *A valid non-back arc of T_p is a valid non-back arc of T_c if its index is less than $e^*(T_c)$.*

Proof: Since T_0 is a depth-first search tree, only descendants of v obtain some new ancestors in T_c. Hence, if a non-back arc g of T_p is a back arc of T_c, then its tail is a descendant of v. Therefore its index is larger than $e^*(T_c)$. ∎

Lemma 5 *A back arc of T_p is a non-back arc of T_c if and only if its tail is a descendant of v and its head is in P.*

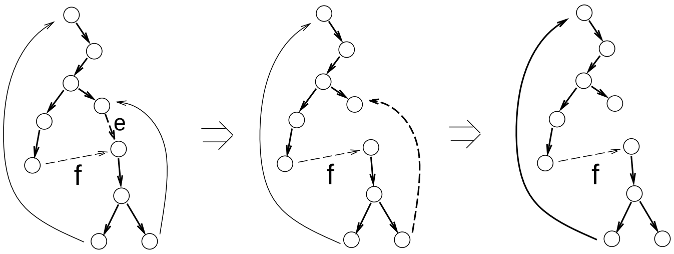

Figure 3.7: The way of $B(T)$ changes.

Proof : A descendant of v is also a descendant of all vertices of P but not in T_c. Thus if the tail of a back arc is a descendant of v and the head of it is in P, it is a non-back arc of T_c. Conversely, only descendants of v have ancestors which are not ancestors in T_c, and only vertices of P have descendants which are not descendants in T_c. Thus any back arc of T_p has its tail in descendants of v and its head in P if it is a non-back arc of T_c. ▮

From these lemmas, the valid non-back arc set of T_c is composed of those back arcs of T_p and valid non-back arcs of T_p whose indices are less than $v^*(T_c)$. By listing both of them in the order of their indices, we can obtain the non-back arc set sorted by their indices. The former can be listed easily if we have the set of valid non-back arcs of T_p sorted by their indices. But the latter is hard to list with only simple data structures. Thus we use a data structure shown as below.

For a directed spanning tree T_p, let $B(T_p)$ be an undirected graph with vertex set V and an edge set composed of arcs of T_p, valid back arcs of T_p and some of other rest back arcs. These arcs are considered as undirected edges in $B(T_p)$. $B(T_0)$ is an undirected graph composed of T_0 and all of its back arcs. The weight of edges in the graph $B(T_p)$ is defined by 0 for arcs of T_p and $(n + 1 -$ index $)$ for others. The minimum spanning tree of $B(T_p)$ is T_p.

By removing a valid arc e of T_p from $B(T_p)$, the minimum spanning tree of $B(T_p)$ is split into two trees. One of them contains all descendants of the head v of e and the other contains the rest. The minimum spanning tree of the graph $B(T_p) \setminus e$ is obtained by adding the minimum weight cut edge b, which has endpoints in both of those two trees. From the above lemmas, b is a valid non-back arc of T_c if and only if the head of b is in P. In the case that b does not exist, no back arc of T_p is a non-back arc of T_c. Since b is a back arc of T_p, the head of b is in the path from r to v and its tail is a descendant of v. From the definition of the weight, no back arc connects a descendant of v and an interior point of the path which is from v to the head of b. Hence if b is not a non-back arc in T_c, then no back arc of T_p is a non-back arc of T_c. We show an example of the way of $B(T)$ changes in Figure 3.7. Emphasized lines are a part of the minimum spanning tree. Doted lines are eliminated and a new edge will be inserted.

Now we can find a new valid non-back arc of T_c by updating the minimum spanning tree. By removing b and applying the method repeatedly, we can enumerate all new valid non-back arcs of T_c. The algorithm is described as follows. It outputs all new non-back arcs and updates the graph from $B(T_p)$ to $B(T_c)$.

```
ALGORITHM: ENUM_NON-BACK_ARCS($T_p$, $B(T_p)$, $e$, $f$)
Step 1: Remove $e$ from $B(T_p)$.
Step 2: Update the minimum spanning tree by adding some edge $b$.
        If $b$ does not exist, then add $f$ to $B(T_p)$, and stop.
Step 3: If $b$ is not a back-arc of $T_c = T_p \setminus e \cup f$, then remove $b$ and add $f$ to $B(T_p)$. Stop.
Step 4: Output $b$. Remove $b$ from $B(T_p)$. Go to Step 2.
```

Lemma 6 *The algorithm* ENUM_NON-BACK_ARCS *outputs all back arcs of T_p which are valid non-back arcs of T_c in the order of their indices. It takes $O(D(n,m))$ time and $O(D(n,m))$ time per one non-back arc. Its space complexity is $O(m + DS(n,m))$.*

Proof: Since the algorithm finds the maximum index one among those back arcs, back arcs are outputted in the order of their indices. To identify both the endpoints of the path P, we have to find the nearest common ancestor of the tail of e and the tail of f. It can be found in $\mathrm{O}(\log n)$ time by D. D. Slater and R. E. Tarjan's dynamic tree data structure [47]. The updating operations, which are adding an edge, deleting an edge and changing the root, are also done in $\mathrm{O}(\log n)$ time. The time complexity of the rest of the algorithm is $\mathrm{O}(\mathrm{D}(n,m))$ per one iteration. Because of the number of iterations, the time complexity is $\mathrm{O}(\mathrm{D}(n,m))$ per one directed spanning tree. ∎

By using this algorithm, we obtain the following theorem.

Theorem 2 *Algorithm* ENUM_DIRECTED_SPANNING_TREES *enumerates all directed spanning trees in $O(D(n,m))$ time per a directed spanning tree. It uses $O(m + n + DS(n,m))$ space.*

Proof: By the above lemmas, the time complexity of algorithm is $\mathrm{O}(m+n)$ for preprocessing, $\mathrm{O}(\log n + \mathrm{D}(n,m))$ time per one directed spanning tree and $\mathrm{O}(\mathrm{D}(n,m))$ time per a child. Thus the time complexity satisfies the condition. The algorithm changes the data structure for updating the minimum spanning tree if a recursive call occurs. When the recursive call ends, we have to restore it. The restoring operation is done by adding the deleted edge, deleting the added edge and changing the modified weight. The total amount of these changes may increase by recursive calls, but not up to $3m$, since any edge is added, deleted and changed its weight at most once, respectively. Therefore they are stored in $\mathrm{O}(m)$ space.

The set of valid non-back arcs and back arcs are treated similarly. After a recursive call, we can restore them by adding and removing the removed or added arcs. The total space for storing them is $\mathrm{O}(m)$. Hence the space complexity is $\mathrm{O}(m + n + \mathrm{DS}(n,m))$. ∎

3.3 Enumerating Common Intervals of Two Permutations

Let p be a permutation with the length n. We denote the ith element of p by $p[i]$. For two indices $1 \leq i_1 \leq i_2 \leq n$, an *interval* $[i_1, i_2]$ is a set of indices i satisfying $i_1 \leq i \leq i_2$. For a permutation p, the interval $[i_1, i_2]$ of p is the set $\{p[i_1], p[i_1+1], ..., p[i_2]\}$. For given two permutations p and q, a *common interval* is a pair of intervals $[i_1, i_2]$ of p and $[j_1, j_2]$ of q such that the intervals are equal. This section considers a problem of enumerating all common intervals of given two length n permutations.

Common intervals are utilized in some algorithms for the traveling salesman problems and the job shop scheduling problems [6, 40]. Some efficient genetic algorithms utilize common intervals of two current feasible solutions to generate their children. The speedup enumeration algorithms for common intervals is practical for these algorithms.

The enumeration problem of common intervals can be transformed to simpler one. Let us consider two permutations e and π. e is given by $e[i] = i$ and $\pi[i] = j$ where j satisfies $p[i] = q[j]$. From the definition of common intervals, if $[i_1, i_2]$ of p and $[j_1, j_2]$ of q are a common interval, they are also a common interval of e and π. The problem of p and q is equivalent to that of e and π. Hence we treat only the problem of enumerating all the common intervals of e and a given permutation π. We simply call intervals of π composing common intervals with some intervals of e common intervals.

Since we are not given two general strings but permutations, the problem has some good properties. These properties helps us to construct a simple and fast algorithm. Let us see one of them. Let $u(x, y) = \max_{x \leq i \leq y}\{\pi[i]\}$, and $l(x, y) = \min_{x \leq i \leq y}\{\pi[i]\}$. Then the following lemma holds [58].

Lemma 7 *An interval $[x, y]$ forms a common interval if and only if the equation $u(x, y) - l(x, y) = x - y$ holds.*

Proof : Since $\pi[i] \neq \pi[j]$ holds for any i and j with $x \leq i < j \leq y$, $u(x, y) - l(x, y) \geq x - y$ always holds. If $u(x, y) - l(x, y) > x - y$ holds, then there is an index $j \in [l(x, y), u(x, y)]$ satisfying that $\pi[i] \neq j$ for any i in $[x, y]$. Therefore $[x, y]$ is not a common interval. Conversely, if $[x, y]$ is a common interval with an interval $[x', y']$ of e, $x' = u(x, y)$ and $y' = l(x, y)$ holds. Since $x' - y' = x - y$ holds, $u(x, y) - l(x, y) = x - y$. ▮

By using this property, we can construct a simple basic algorithm [58]. The algorithm checks the above condition for all pairs of x and y iteratively. Since $u(x, y) = \max\{u(x, y-1), \pi[y]\}$ and $l(x, y) = \min\{u(x, y-1), \pi[y]\}$ hold for any $0 < x < y \leq n$, we can compute $u(x, y)$ and $l(x, y)$ in O(1) time from $u(x, y-1)$ and $l(x, y-1)$. Since the total number of iterations is $(n+1)n/2$, the time complexity of the algorithm is $O(n^2)$. We describe this basic algorithm as follows.

```
ALGORITHM: BASIC_ALGORITHM
Step 1: For $x = 1, \cdots, n-1$ do
Step 2:     For $y = x, \cdots, n-1$ do
Step 3:         Compute $u(x,y)$ and $l(x,y)$.
Step 4:         If $u(x,y) - l(x,y) = x - y$, then output $x, y$.
Step 5:     End for
Step 6: End for
```

Since there may exist $(n+1)n/2$ common intervals, the time complexity is optimal in the input size. It is also optimal in the output size if the number of the outputs is $\Theta(n^2)$, but if there are few common intervals, it is not optimal. An optimal algorithm in the sense of time complexity has to terminate in $\mathrm{O}(n+N)$ time where N is the number of common intervals.

Our algorithm in this section attains $\mathrm{O}(n+N)$ running time by adding some modifications to the basic algorithm. In the next subsection, we see some properties which the modifications are based on. Our algorithm changes the way of search to utilize the properties efficiently, and reduces the time complexity by introducing some data structures based on linked lists. The details of the algorithm is described in the next subsection.

3.3.1 Some Properties and Linked List Data Structures

To check whether an interval is a common interval or not, we define a function $f(x,y)$ of an interval $[x,y]$ by $u(x,y) - l(x,y) - (y-x)$. If $f(x,y) = 0$ holds, then $[x,y]$ is a common interval. Our algorithm finds all intervals satisfying $f(x,y) = 0$ iteratively. The main idea of the algorithm is to save the time to check whether $f(x,y) = 0$ or not for some indices y which can be concluded as *unnecessary* from the past search information. For a fixed x, we call an index y unnecessary if it satisfies $f(x',y) > 0$ for all $x' \leq x$.

The framework of the algorithm is described as follows.

```
ALGORITHM: ENUM_COMMON_INTERVALS
Step 1: $Y := \{1, ..., n\}$.
Step 2: For $x = n-1, \cdots, 1$ do
Step 3:     Output all $y > x$ in $Y$ satisfying $f(x,y) = 0$.
Step 4:     Remove all “unnecessary” $y$ from $Y$, which satisfy $f(x',y) > 0$ for all $x' < x$.
Step 5: End for
```

The key to this algorithm is the way to find unnecessary indices. The following lemmas help us to identify them.

Lemma 8 *Suppose that we are given $x > 1$ and $y > x$. If $u(x,y) < u(x,y')$ and $u(x-1,y) = u(x-1,y')$ hold for some $y' > y$, y satisfies $f(x',y) > 0$ for all $x' < x$.*

Proof: From $u(x,y) < u(x,y')$, there exists some $y'' \in [y+1, y']$ satisfying $\pi(y'') \in [u(x,y)+1, u(x,y')]$. From $u(x-1,y) = u(x-1,y')$, we have $[u(x,y)+1, u(x,y')] \subseteq [l(x',y), u(x',y)]$ and $\pi(y'') \in [l(x',y), u(x',y)]$. As $y'' \notin [x',y]$, $f(x',y) > 0$. ■

Lemma 9 *Suppose that we are given $x > 1$ and $y > x$. If $f(x,y) > f(x,y')$ hold for some $y' > y$, y satisfies $f(x',y) > 0$ for all $x' < x$.*

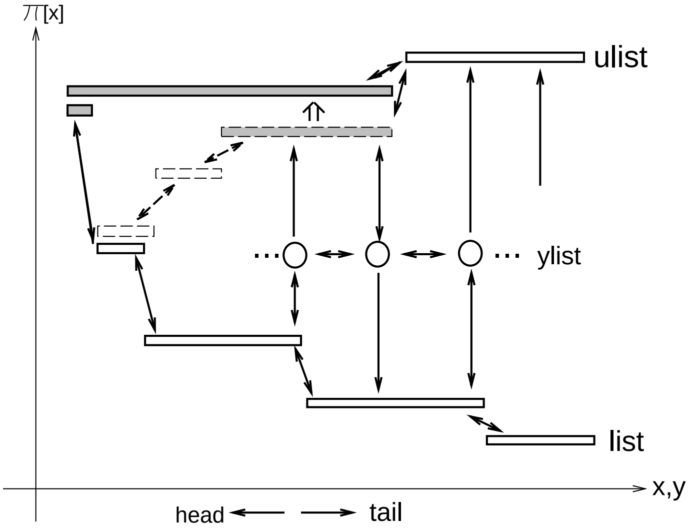

Figure 3.8: The process of updating *ulist*, *llist* and *ylist*. The cells represented by dotted line are deleted.

Proof : From $f(x,y) > f(x,y')$, there exists $y'' \in [y+1, y']$ which satisfies $\pi(y'') \in [l(x,y), u(x,y)]$. Since $y'' \notin [x', y]$, $f(x', y) > 0$. ▮

Now we can find a part of unnecessary indices from these properties. Although these properties do not find all the unnecessary indices, those indices are sufficient to obtain a fast algorithm. In the following, we describe an algorithm that removes all indices satisfying Lemma 8 or 9 from Y at Step 5 of the algorithm ENUM_COMMON_INTERVALS. Without loss of generality, we consider only the case that $\pi[x-1] > \pi[x]$ throughout this section. The opposite case is treated similarly.

To maintain Y, the algorithm uses a double linked list called *ylist* composed of cells $y_1, ..., y_r$ corresponding to each y of Y. These cells are sorted in the increasing order of their indices in the *ylist*. The algorithm for removing the unnecessary cells from *ylist* is as follows.

```
ALGORITHM: TRIMMING_YLIST
Step 1: Find $y^*$ which is maximum among $y$ satisfying $u(x,y) < u(x-1,y)$.
Step 2: If the cell $y$ on the head of ylist satisfies $u(x,y) < u(x,y^*)$, then remove it from
        ylist ( from Lemma 8). Go to Step 2.
Step 3: Let $y_i$ and $y_{i+1}$ be the consecutive cells of ylist satisfying $y_i \le y^* < y_{i+1}$.
        If $f(x,y_i) > f(x,y_{i+1})$ then remove $y_i$ from ylist ( from Lemma 9). Go to Step 3.
```

By executing the algorithm at the end of each iteration, any i and $j > i$ satisfy $f(x,i) \le f(x,j)$. Therefore Step 3 of this algorithm removes all y satisfying Lemma 9 exactly.

In Step 1 and 2, we have to spend O$(y-x)$ time to calculate $u(x,y)$ without any data structure. To achieve linear time algorithm, we have to obtain them in sufficiently short time. In our algorithm, we represent the functions u and l by linked lists called *ulist* and *llist*. For a fixed x, u (resp., l) is a monotonically non-decreasing (resp., non-increasing) function in y. All $y \in [x,n]$ are partitioned into groups by their values of $u(x,y)$. The interval

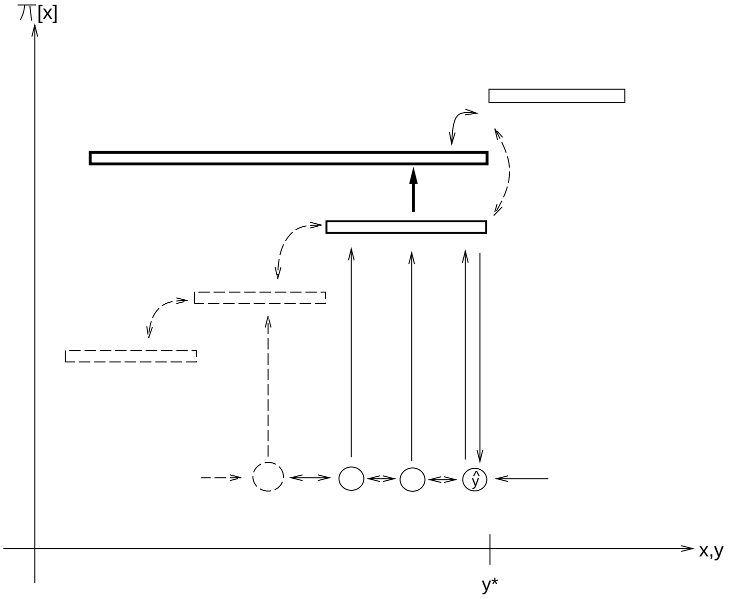

Figure 3.9: Finding y_i from the cell includes y^*.

$[x, n]$ is decomposed into intervals $[y_0 = x + 1, y_1 - 1], [y_1, y_2 - 1], ..., [y_{r-1}, y_r = n]$ uniquely such that $u(x, y') = u(x, y'')$ holds if and only if both y' and y'' are included in $[y'_i, y'_{i+1} - 1]$. By using this partition, we represent u by *ulist* composed of cells corresponding to each interval. Each cell stores the corresponding interval and the value $u(x, y)$ for y which the interval includes. All cells are linked in the increasing order of indices in their intervals. We say that y is included in the cell of *ulist* if the interval corresponding to it includes y.

To get the value of $u(x, y)$, we have to find the cell including y. To realize the operation in short time, we make pointers between cells of *ylist* and *ulist* if the former is included in the latter. If a cell includes more than two pointers to cells of *ylist*, we choose the maximum one among them (see Figure 3.9).

The update of *ulist* and *llist* when x changes to $x - 1$ is executed as follows. The update of *llist* is done by adding a cell corresponding to the interval $[x - 1, x - 1]$ on the head of it. Recall that we treat only the case $\pi[x - 1] > \pi[x]$. We delete the cell of *ulist* which includes the index y satisfying $u(x, y) < u(x, y^*)$. For the cell including y^*, we change its interval to $[x - 1, y^*]$ and its value from $u(x, y^*)$ to $u(x - 1, y^*)$ (see the figure 3.8). Note that we do not remove the cell corresponding to $u(x, y^*)$, but use it to store the interval corresponding to $u(x - 1, y^*)$. By doing this, pointers from the index y included in the cell need not to be changed. It is one of the key to speeding up the algorithm.

In Step 2 of TRIMMING_YLIST, if the pointer of the cell y of *ylist* indicates a deleted cell of *ulist*, we remove it from *ylist*, since this implies $u(x, y) < u(x, y^*)$. Thus the update of pointers between *ylist* and *ulist* is unnecessary. If there exists y satisfies Lemma 8, the head of *ylist* also satisfies it. Thus Step 2 removes all y satisfying Lemma 8 exactly.

Let us consider the time complexity of the algorithm TRIMMING_YLIST. Since those update operations of *ulist* are done by tracing *ulist* from its head to the cell including y^*,

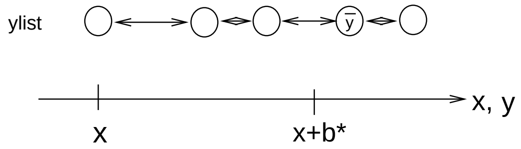

Figure 3.10:

Step 1 and 2 take O$(d+1)$ time where d is the number of deleted cells in Step 2. In each iteration, only one new cell is created. It takes O(1) time.

In Step 3, we find y_i and y_{i+1}. It is done in O(1) time by finding the cell of *ylist* corresponding to the maximum index y included in the cell including y^* (see the figure 3.9). Step 3 is repeated while the current tracing cell is deleted. It is done in O(1) time per a deleted cell. Thus the total time spent for TRIMMING_YLIST in all iterations of ENUM_COMMON_INTERVALS is proportional to the total number of deleted cells. It does not exceed the number of created cells, thus the total time is O(n).

Theorem 3 *The algorithm* ENUM_COMMON_INTERVALS *with* TRIMMING_YLIST *outputs all common intervals in $O(n+N)$ time and $O(n)$ space.*

Proof : We have already seen the correctness of the algorithm. The time complexity of the algorithm is O(n) except for Step 3. The cells $y_1, \cdots, y_r$ of *ylist* satisfy $f(y_i) \leq f(y_{i+1})$. Thus we can enumerate all y satisfying $f(x,y) = 0$ by tracing *ylist* from its head for each x. Since $f(x,y)$ is monotonically increasing in *ylist*, we can list them without scanning y with $f(x,y) > 0$ in the middle of *ylist*. When we encounter y with $f(x,y) > 0$, we stop the tracing since $f(x,y') > 0$ holds for all $y' > y$. Therefore the time complexity is O(1) per one common interval. ▮

3.3.2 Modifications for Some Classes of Common Intervals

In the previous subsection, we saw an algorithm for enumerating all common intervals. By using the algorithm, we consider some kinds of common intervals in this subsection. The problems are to enumerate all common intervals with lengths in the specified range, and with the maximum length. We show some modifications to obtain optimal algorithms for these problems. Both their time complexities are linear in the input size and the output size.

Firstly, we explain the algorithm for the former problem. For two specified lengths $b_l < b_u$, we consider the problem of enumerating the common intervals whose lengths are not smaller than b_l and not greater than b_u. We enumerate them by using the above algorithm with slight modifications. In each iteration, we keep the minimum index $\hat{y}$ in the *ylist* satisfying $\hat{y} - x + 1 \geq b_l$.

When we insert or delete a cell to *ylist*, we update $\hat{y}$. Since the updated $\hat{y}$ is always adjacent to the original one, hence we do not increase the time complexity of the algorithm. The enumeration of y satisfying $f(x,y) = 0$ and $b_l \leq y - x + 1 \leq b_u$ can be done in O$(n + N')$ time by tracing *ylist* from $\hat{y}$, where N' is the number of the outputted common intervals.

Next we consider the problem of finding a common interval with the maximum length whose length is not greater than the specified length b_u. The basic idea is similar to the above algorithm. We keep the maximum cell $\bar{y}$ of *ylist* satisfying $\bar{y} - x \geq b^*$ where b^* is the maximum length of common intervals with the length not greater than b_u among those found in the previous iterations of the algorithm (see Figure 3.10).

At the end of each iteration x, we update $\bar{y}$ if there is an index y in *ylist* satisfying that $f(x, y) = 0$, $\bar{y} < y$ and $y - x < b_u$. In such a case, we update $\bar{y}$ to y by tracing *ylist* from $\bar{y}$ while the current traversing cell y' satisfies $f(x, y') = 0$ and $y' - x < b_u$. Through the algorithm, a cell never be traced twice by this operation. Hence the algorithm terminates in O(n) time.

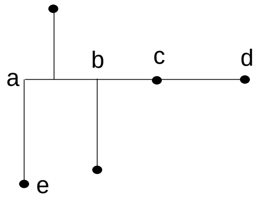

Figure 3.11: Nodes a, b, c, d and e: a is a corner and b is a steiner point. The line connecting a and d is a maximal horizontal line. The line connecting b and c is a segment.

3.4 Enumerating Distance Preserving Rectilinear Steiner Trees of Point Sets in the Plane

Let V be a set of n points on the Euclidean plane. A point of V is called a *terminal.* We consider networks obtained by connecting some terminals of V with only horizontal and vertical lines. These lines are allowed to cross with each other. The networks are called *rectilinear networks.* In this section, we propose an enumeration algorithm for tree shaped rectilinear networks composed of these terminals, horizontal lines and vertical lines.

To define the problem in detail, we introduce some notations and definitions. For a point v, we denote its x-coordinate and y-coordinate by $x(v)$ and $y(v)$, and denote a point with the x-coordinate x and the y-coordinate y by (x, y). We define a *maximal line* in a rectilinear network by a line which is not included in any other line of the network. A point of a line is called an *interior point* if it is not an endpoint of the line. Let $x \in V$ be a point included in both a maximal horizontal line and a maximal vertical line. If x is their endpoint, we call x a *corner.* Otherwise we call x a *steiner point.* In a rectilinear network, terminals, corners and steiner points are called *nodes.* For a point and a line of the network, we say that the point is *incident* to the line if it is included in the line, and vice versa. A line sharing a point with another line l is said to be *adjacent* to l. Especially, if they share their interior points, we say that they *cross.* A line both whose endpoints are nodes is called a *segment* if no node of the network is on its interior points (see Figure 3.11). A maximal line is decomposed into some segments uniquely. For a segment, we say that its endpoints are adjacent. Let the *degree* of a node be the number of segments incident to it. The degree of any node does not exceed four. The degree of a corner is 2, and the degree of a steiner point is at least 3. To represent a rectilinear network, we use the sets of its nodes and segments, and their adjacency. Since this representation is similar to that of graphs, we can utilize various data structures of graphs.

Similar to general networks or graphs, we define paths and trees. We define a *path* by a set of segments $\{s_1, s_2, ..., s_k\}$ such that s_i shares one of its endpoints with s_{i-1} and the other endpoint with s_{i+1}. s_1 and s_k have endpoints not incident to s_2 or s_{k-1}. We call them the *endpoints* of the path. In a rectilinear network, we say that a node is *connected* to another node if the network includes some paths whose endpoints are these nodes. We also define a *cycle* by a path whose both endpoints are the same point. From these, a forest is defined by a rectilinear network without any cycle. Especially, a forest is called a *rectilinear steiner*

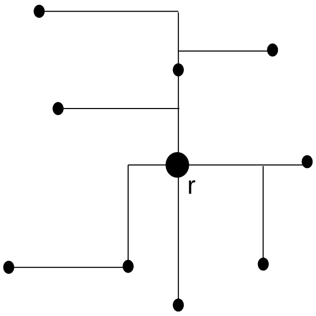

Figure 3.12: A distance preserving tree

tree if any pair of nodes of V are connected in the forest and any its leaf is a terminal. We denote a rectilinear steiner tree by an RST. For an RST, we assume that we are given a *root terminal* r of the tree, which is a specified terminal of V. Without loss of generality, we assume that $x(r), y(r) = 0$.

We next introduce the length of a path in a tree. The length of a line is given by the Euclidean distance between its two endpoints. The length of the path is given by the sum of the lengths over all segments composing the path. For a node v of an RST T, we define the *distance* of v by the length of the unique path from r to v in T.

An RST T with the root terminal r is called a *distance preserving tree*, denoted by DPT, if the distance of any node v of T is $|x(v) - x(r)| + |y(v) - y(r)| = |x(v)| + |y(v)|$. In a DPT, we can see that any node v except r is adjacent to exactly one node u such that $|x(v)| \leq |x(u)|$ and $|y(v)| \leq |y(u)|$. If a node v does not satisfy the condition, the DPT will contain a cycle, or the distance from the root to the node is not $|x(v)| + |y(v)|$ (see Figure 3.12).

Now we consider the problem of enumerating all DPTs for a given terminal set V and a root terminal r. For a terminal set, there are infinitely many RSTs. Because, for any RST, we can generate a new one by moving a segment slightly and adjusting the lengths of the segments adjacent to the segment. Hence we consider only trees satisfying the condition that any maximal line of the tree is incident to some terminals. We call this condition the *incidence condition*. Under the condition, for any endpoint v of any segment, there are terminals u and u' such that $x(v) = x(u)$ and $y(v) = y(u')$. Hence the number of possible segments appear in RSTs is finite. The number of combinations of them is also finite. In the rest of this section, we treat only those rectilinear steiner trees. By introducing this condition, we have no two consecutive corners in an RST. Therefore the number of segments in an RST is bounded by $4n$. Hence the required memory space for storing an RST is $O(n)$.

3.4.1 Reverse Search for Distance Preserving Trees

In this subsection, we consider a reverse search algorithm for enumerating all DPTs satisfying the incidence condition. We firstly introduce a parent-child relationship among all DPTs which is utilized to construct our reverse search. Let the *center path* of a point v be the path composed of two lines from v to $(0, y(v))$ and from $(0, y(v))$ to r. We define a DPT T_0 by the union of the center paths of all terminals in V. T_0 has only one maximal vertical line whose

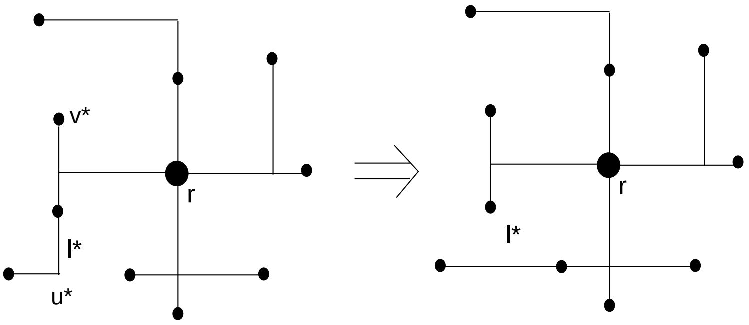

Figure 3.13: Generating the parent of a DPT by deleting the parent path and adding the hook path.

x-coordinate is 0. From the way of the construction, T_0 must be a DPT. T_0 will be the root of the enumeration tree.

For a point v of an RST T, let the *parent path* of v be the path from v to the node v' where v' is the terminal or the steiner point next to v in the path from v to r in T. Let us consider the network obtained by adding the center path of v to T. We define the *hook path* of v by the subpath of the center path of v which is from v to the terminal or the steiner point next to v. If the hook path is in T, then it is the parent path of v. We call a node whose center path is not entirely included in T an *active node.*

For all points of the plane, we introduce a total ordering among them. Let u and v be two points. If $x(v) \neq x(u)$ holds, then we say that u is *higher* than v if $x(v) < x(u) \leq 0$, $x(v) > x(u) \geq 0$, or $x(v) > 0$ and $x(u) \leq 0$ holds. In the case that $x(u) = x(v)$, we say that u is higher than v if $y(v) < y(u) \leq 0$, $y(v) > y(u) \geq 0$, or $y(v) > 0$ and $y(u) \leq 0$ holds.

For an RST T, we define $v^*(T)$ by the highest active node of T. We denote the maximal vertical line of T including $v^*(T)$ by $l^*(T)$. If no vertical line is incident to $v^*(T)$, then we treat $v^*(T)$ as a vertical line with the length zero, and define $l^*(T)$ by $v^*(T)$. For a DPT, $x(v^*(T)) \neq 0$, since a DPT includes the hook path of any node with the x-coordinate zero. For any DPT T, the length of $l^*(T)$ is not zero.

By using these, we define the parent of a DPT. For a DPT $T_c \neq T_0$, let $u^*(T_c)$ be $v^*(T_c)$ if the lowest node of $l^*(T_c)$ is a terminal, and otherwise the lowest active node of $l^*(T_c)$. In any DPT T, $u^*(T)$ is an endpoint of $l^*(T)$ or adjacent to an endpoint of $l^*(T)$. Otherwise, at least two nodes of $l^*(T)$ have their center paths in T, hence T includes a cycle. The parent T_p of T_c is given by the tree obtained by deleting the parent path of $u^*(T_c)$, and adding the hook path of $u^*(T_c)$ (see Figure 3.13). T_p satisfies the following property.

Lemma 10 *The tree T_p obtained by the above operations is a DPT satisfying the incidence condition.*

Proof : Since T_p is obtained by adding the hook path of $u^*(T_c)$, no node of T_c gains its distance. Thus we check the incidence condition of T_p. The maximal lines including a segment of the parent path of $u^*(T_c)$ may lose some terminals in T_p. The segments of the hook path of $u^*(T_c)$ may include no terminal if they are maximal lines in T_p. We check these in the

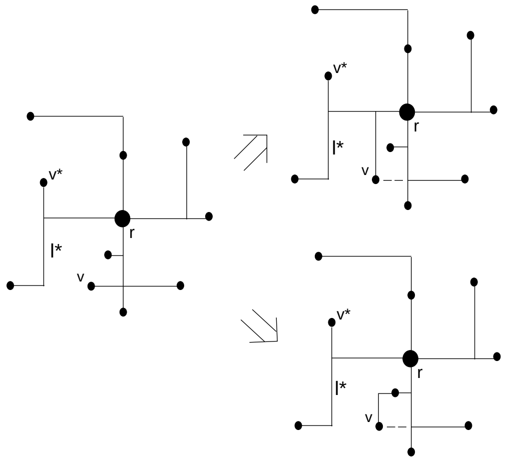

Figure 3.14: Generating children of a DPT by the operations (1) and (2).

following. Note that the other maximal lines include terminals since they include maximal lines of T_c.

If the hook path includes a vertical segment, then the segment is included in the maximal line of T_p including r. If $u^*(T_c)$ is not a terminal, then there is a horizontal line including $u^*(T_c)$ in T_c. Hence, if the horizontal segment of the hook path is a maximal line in T_p, then $u^*(T_c)$ is a terminal.

By deleting the parent path, $l^*(T_c)$ loses one of the endpoints. If the endpoint is a terminal, then the other endpoint of $l^*(T_c)$ is a terminal because of the selection rule of $u^*(T_c)$. If the parent path includes a horizontal segment, then a maximal horizontal line may lose one of its endpoint. The endpoint is a corner included in the parent path. Therefore T_p satisfies the incidence condition. ∎

From the lemma, the parent of a DPT is uniquely defined. Since one active node of T_c is not active in T_p and no non-active node of T_c is active in T_p, no DPT is its own ancestor in this parent-child relationship. By using this relationship, we can construct an enumerating algorithm based on reverse search. The algorithm requires an algorithm for listing all children of a DPT. We describe the method for finding all children of a DPT T_p in the following.

Let $\hat{V}(T_p)$ be the node set of the endpoints of $l^*(T_p)$ and terminals v higher than $v^*(T_p)$ with $x(v) \neq 0$. Especially, we define $\hat{V}(T_0)$ by all terminals v of T_0 with $x(v) \neq 0$. For a point v in T_p, let $C_1(v)$ be the set of points u in T_p such that the vertical line connecting v and u includes neither point of T_p nor the point $(x(v), 0)$ in its interior points. $|C_1(v)|$ is at most 2. We also define $C_2(v)$ by the set of nodes u of T_p satisfying that (a) $|x(u)| < |x(v)|$, (b) $|y(u)| < |y(v)|$, and (c) the path composed of the vertical line from v to $(x(v), y(u))$ and the horizontal line from $(x(v), y(u))$ to u includes neither point of T_p, the point $(x(v), 0)$, nor $(0, y(u))$ except on its endpoints.

For each node $v \in \hat{V}(T_p)$, we consider trees obtained by the following two operations (see Figure 3.14)

(1) Let s be the vertical line connecting v and a node $u \in C_1(v)$. Let $\bar{v}$ be v if $|y(v)| > |y(u)|$ and otherwise u. We obtain a tree by adding s and deleting the parent path of $\bar{v}$ from T_p.

(2) Let s' be the vertical line from a node $u \in C_2$ to $(x(v), y(u))$, and s be the horizontal

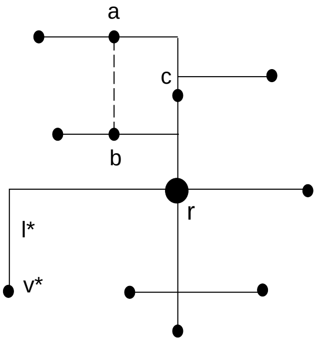

Figure 3.15: A child generated doubly: a is in $C_1(b)$, and b is in $C_1(a)$. The child obtained by adding the line from a to b and deleting the path from a to c is obtained by the operation (1) with both points of $C_1(a)$ and $C_1(b)$.

line from v to $(x(v), y(u))$. We set $\bar{v}$ to v. We obtain a tree by adding s and s', and deleting the parent path of $\bar{v}$ from T_p.
In the obtained tree, the distances of $\bar{v}$ and v are preserved. Hence the obtained tree is a DPT. The obtained tree does not always satisfy the incidence condition, but satisfies the following condition.

Lemma 11 *For any tree T obtained by the operation (1) or (2) with v from T_p, only the maximal horizontal line h incident to $\bar{v}$ may include no terminal.*

Proof : By generating T, the maximal lines of T_p incident to $\bar{v}$ may lose some terminals in T_p. The segments s and s' may include no terminal if they are maximal lines in T. We check these in the following.

If v is a terminal, then s includes at least one terminal. Otherwise, v is an endpoint of $l^*(T_p)$, hence s is not a maximal line of T. In the operation (2), if u is not a terminal, then it is incident to a maximal horizontal line. Hence s' is not a maximal line of T. If the hook path of $\bar{v}$ includes a vertical segment, then it is included in a maximal vertical line of T including r. Therefore, only the maximal horizontal line of T including $\bar{v}$ may include no terminal. ∎

The obtained tree may not be a child of T_p, but we can obtain any child of T_p by the operations (1) and (2). We prove it in the following lemma.

Lemma 12 *All children of T_p are generated by the operations (1) and (2).*

Proof : Suppose that T_c is a child of T_p. T_p is obtained by adding the hook path of $u^*(T_c)$ and deleting the parent path of $u^*(T_c)$ from T_c. We denote the other endpoint of the parent path by u. If the parent path is a vertical segment included in $l^*(T_c)$, then either u is in $C_1(v)$ or v is in $C_1(u)$ hold. At least one node among u and v is in $\hat{V}(T_p)$. The vertical line connecting u and v does not include $(x(v), 0)$. Hence T_c is generated by the operation (1). In the other case, the degree of v is at least 3, hence v is a node of T_p. Since $|x(u)| < |x(v)|$, $|y(u)| < |y(v)|$, and the parent path of $u^*(T_c)$ includes neither point of T_p nor the point $(x(v), 0)$ except its endpoints, u is included in $C_2(v)$. Hence T_c is generated by the operation (2). ∎

Now we can generate all children of T_p by these operations by applying the operations (1) and (2) to all nodes of C_1 and C_2. However, we may generate a child twice, if two nodes v and v' of $\hat{V}$ satisfy that $v \in C_1(v')$ and $v' \in C_1(v)$. In this case, we stop generating one of them.

The children of T_p satisfy some other properties. We show them in the following.

Lemma 13 *If a node u of $C_2(v)$ generates a child T_c by the operation (2), then any node u' of $C_2(v)$ generates a child.*

Proof : From the assumption, $u^*(T_c)$ is v, and the maximal horizontal line of T_c including v always includes a terminal. Let T' be the tree obtained by the operation (2) with u'. Since the maximal horizontal line including v in T' is same as that of T_c, it always includes a terminal. Hence T' satisfies the incidence condition from Lemma 11. $l^*(T_c)$ and $l^*(T')$ include the same nodes except for their endpoints which are added to generate them. Since the endpoints are not active, $u^*(T_c) = u^*(T') = v$. Hence the network obtained by deleting the parent path of $u^*(T_c)$ from T_c is same as the network obtained by deleting the parent path of $u^*(T')$ from T'. Therefore their parent is T_p. ▮

Lemma 14 *T_p has $\Omega(|\hat{V}(T_p)|)$ children.*

Proof : For any terminal $v \in \hat{V}(T_p)$ which is not an endpoint of $l^*(T_p)$, if the vertical line from v to $(x(v), 0)$ includes a point of T_p not v, then the node u next to v in the vertical line is included in $C_1(v)$ or $l^*(T_p)$. If u is in $C_1(v)$, then the tree T obtained by the operation (1) satisfies the incidence condition since the horizontal line including v includes a terminal v. Since (1) $l^*(T)$ is the deleted line to obtain T, and (2) $u^*(T)$ is v, T is a child of T_p. In the case that the vertical line from v to $(x(v), 0)$ includes no point of T_p except v, a node $(a, 0)$ is included in $C_2(v)$. Since v is a terminal, the DPT T obtained by the operation (2) with v and $(a, 0)$ always satisfies the incidence condition. $u^*(T)$ is v. The parent of T is obtained by removing $l^*(T)$ and adding the hook path of $u^*(T)$, thus, it is T_p. A child may be generated from at most two nodes of $\hat{V}(T_p)$, hence the number of children is at least $\Omega((|\hat{V}(T_p)| - 2)/2) = \Omega(|\hat{V}(T_p)|)$. ▮

By estimating these lemmas, we can construct a polynomial delay algorithm for enumerating all DPTs. Some naive implementations can be considered, but they may be not efficiently fast. In the following subsection, we show some data structures for speeding up.

3.4.2 Data Structures for Speeding Up

In the previous subsection, we described a simple reverse search. In this subsection, we show some data structures for finding all children of a DPT in efficiently short time.

For all nodes v in $\hat{V}(T_p)$, we have to obtain all the nodes of $C_1(v)$ and $C_2(v)$ which generate children by the operations (1) and (2). If we have these nodes, we can easily construct children in O(1) time per a child, since we have only to add and delete a constant number of lines from T_p. In the following, we consider the algorithm and the data structure for finding nodes of C_1 and C_2 which generate children of T_p in O(1) time per a node.

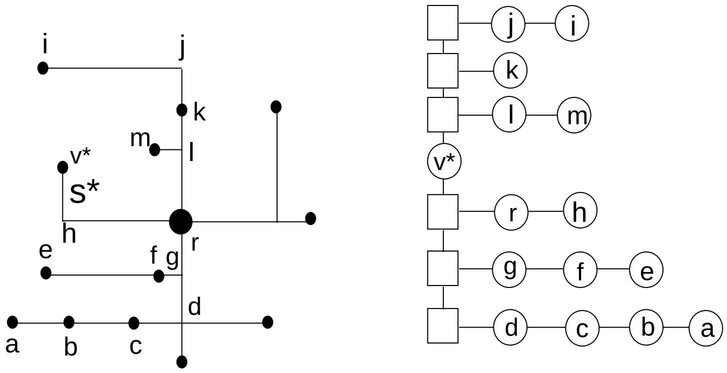

Figure 3.16: An instance of the linked list W

We have two data structures which are for nodes v with $x(v) < 0$, and for nodes v with $x(v) > 0$. Since they are symmetric, we show only the former data structure. Suppose that a node $v \in \hat{V}(T_p)$ satisfies $x(v) < 0$. Let T' be the network obtained by deleting all points in the halfplane $x > 0$ from T_p. To find the nodes of $C_2(v)$, we maintain a linked list W such that each whose cell corresponds to each endpoint of $l^*(T)$ and maximal horizontal line of T' one of whose endpoint has the x-coordinate 0. If T_p includes some nodes with the x-coordinates 0 which are not nodes in this network, we treat them as maximal horizontal lines with the length 0, and insert them in W. We store all components of W in the decreasing order of their y-coordinates. Each horizontal line h of W consists of a sequence of nodes included in h. We also store them in the order of their x-coordinates. For a horizontal line h of W, let $x(h)$ be the x-coordinate of the endpoint of h with the smaller x-coordinate (see Figure 3.16).

By using W, we can find all the nodes of $C_2(v)$ which generate children. Suppose that the horizontal line h of W includes v. From the definition of $C_2(v)$, any node u in $C_2(v)$ is on the end of the horizontal line h' with $x(h') > x(v)$. Since the vertical line from v to $(x(v), y(u))$ and the horizontal line from $(x(v), y(u))$ to u include no point of T_p in their interior points, there is no maximal line l with $x(l) \leq x(v)$ between h and h' in W. If $x(v) = x(v^*(T_p))$, then no endpoint of $l^*(T_p)$ is also there. Hence, by tracing W from h until encountering a horizontal line l with $x(l) \leq x(v)$, the point $(x(v), 0)$, or an endpoint of $l^*(T_p)$ if $x(v) = x(v^*(T_p))$, we can find all the horizontal lines h' which include nodes of C_2.

From Lemma 13, if one node of $C_2(v)$ generates a child, then all the other nodes of $C_2(v)$ generate children. Hence we can list all the nodes of $C_2(v)$ generating children by checking whether the generated tree T with a node u of $C_2(v)$ is a child of T_p or not. It can be done by checking the incidence condition of T and whether v is $u^*(T)$ or not. Since $u^*(T)$ is in $l^*(T)$, we can check the latter in O(1) time. From Lemma 11, we check the incidence condition by checking whether the horizontal line including $\bar{v}$ includes a terminal or not. To check it quickly, we store the nearest terminal on the center path for each node in horizontal lines of W. We can check whether the horizontal line including v satisfies the incidence condition or not by looking the x-coordinate of the nearest terminal from the endpoint of the horizontal line. It takes O(1) time. The trace of W takes O(1) time per a node of $C_2(v)$, since the

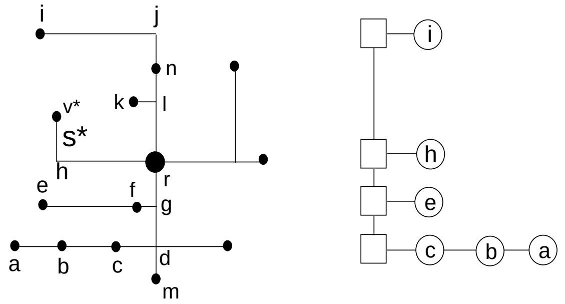

Figure 3.17: An instance of a linked list Y for the node c

endpoints of l^* is at most two. Therefore we can find all nodes of $C_2(v)$ generating children in linear time in the number of them.

To find the nodes of $C_1(v)$, we use a linked list Y which is composed of all horizontal lines l of W with $x(l) < 0$ (see Figure 3.17). We suppose that h is the horizontal line of Y including v. A node u of $C_1(v)$ is included in a horizontal line h' with $x(h') \le x(v)$ such that no horizontal line l between h and h' in Y satisfies $x(l) \le x(v)$. It can be found by tracing Y from h until meeting h'. After finding h', we find the segment including u by tracing the node included in h'. By this method, we can find all the nodes of $C_1(v)$.

If there are either many horizontal lines l with $x(l) > x(v)$, or many nodes w with $x(w) > x(v)$ in h', then this method may take much time. If Y includes no horizontal line l with $x(l) > x(v)$, and no node w with $x(w) > x(v)$ is included in horizontal lines of Y, we can find all the nodes of $C_1(v)$ in O(1) time. Hence those lines and nodes can be concluded as unnecessary to find the nodes of $C_1(v)$. Therefore we delete all these unnecessary lines and nodes from Y when we find the nodes of $C_1(v)$.

To reduce the time for the deletion of those unnecessary lines and nodes, we operate all nodes $v, x(v) < 0$ of $\hat{V}(T_p)$ in the decreasing order of their x-coordinates, and update Y in each iteration. In this ordering, any node or line is deleted at most once. Hence the total number of the deleted lines and nodes does not exceed $2|\hat{V}(T_p)|$. The sum of updating time of Y is $O(|\hat{V}(T_p)|)$.

By using Y, we can find all the nodes of C_1 and C_2 generating children of T_p in $O(|\hat{V}(T_p)|)$ time and additional O(1) time per a node. Next we see how to maintain W and Y.

When we obtain a child T_c generated by $v \in \hat{V}(T_p)$ and $u \in C_1(v) \cup C_2(v)$, we start enumerating all the descendants of T_c by a recursive call. The recursive call also requires the linked list W and Y for T_c. Hence we update W and Y. The symmetric difference between T_p and T_c includes at most two horizontal lines, hence we update at most two horizontal lines and at most one endpoint of $l^*(T_p)$ of W. Each of these update operations is done by inserting a segment, deleting a segment, or splitting a horizontal line. These can be done in O(1) time respectively, since W and Y is composed of linked lists. The deletion of a node can be also done in O(1) time. The insertion of the nodes to horizontal lines of W and Y is also done in O(1) time since the nodes are u or the endpoints of the horizontal lines. Since any new node of a horizontal line of W is the endpoint of the line, the update of the nearest terminals in the center path for all nodes in W is done in O(1) time.

To update Y, we spend additional computation time. If T_c is generated with a node of C_1, then some nodes and lines may be deleted from Y. In this case, we restore them to their original positions in Y. This restore operation can be done in the time proportional to the number of the deleted nodes and lines. For a horizontal line h with $x(v^*(T_p)) < x(h) < 0$, its endpoint u with $x(u) < 0$ is a terminal since there is no vertical line with the x-coordinate smaller than $x(v^*(T_p))$ and larger than 0. Hence all deleted lines include some nodes of $\hat{V}(T_p)$. Therefore the number does not exceed twice the number of nodes higher than v in $\hat{V}(T_p)$. Since a node of $\hat{V}(T_p)$ higher than v is included in $\hat{V}(T_c)$, the updating time is at most $\mathrm{O}(|\hat{V}(T_c)|)$.

Now we describe our enumeration algorithm.

```
ALGORITHM: ENUM_DPT( $T_p$ )
Step 1: For all nodes $v \in \hat{V}(T_p)$ do
Step 2:     For all nodes in $C_1(v)$ and $C_2(v)$ generating children do
Step 3:         Construct $T_c$ by the operation (1) or (2).
Step 4:         Update the linked lists $W$ and $Y$ for $T_c$.
Step 5:         Enumerate all descendants of $T_c$ by ENUM_DPT( $T_c$ ).
Step 6:         Restore the linked list $W$ and $Y$.
Step 7:     End for
Step 8: End for
```

Theorem 4 *All distance preserving trees constructed by the node set V in a plane with satisfying the incidence condition can be enumerated in $O(n \log n + N)$ time and $O(n)$ memory where N is the number of distance preserving trees satisfying the incidence condition.*

Proof: We already saw the correctness of the algorithm, hence we show the time and space complexities. We can see that the space complexity is $\mathrm{O}(n)$, since each data structure uses only $\mathrm{O}(n)$ storages. The algorithm requires $\mathrm{O}(n \log n)$ preprocessing time for sorting the nodes of V in x-coordinates and y-coordinates. By this, we construct T_0, W and Y. We also find the nearest terminal on the center path for each terminal, and sort all the terminals in the order of hight.

By using W and Y, we can find all children of T_p in $\mathrm{O}(|\hat{V}(T_p)|)$ time and $\mathrm{O}(1)$ time per a child. From Lemma 14, the number of children is at least $\Omega(|\hat{V}(T_p)|)$. Hence the time complexity of an iteration is $\mathrm{O}(1)$ per a child. The update operations of W and Y are done in $\mathrm{O}(|\hat{V}(T_c)|)$ time.

From these facts, the total time spent for a DPT is proportional to the number of its children and grandchildren. Since the sum of the number of the children and the grandchildren over all vertices in the enumeration tree is $\mathrm{O}(N)$, the time complexity of the algorithm is $\mathrm{O}(n \log n + N)$. ∎

Chapter 4

Bounding the Time Complexity by Detailed Analysis

In the existing studies, techniques for speeding up enumeration algorithms are deeply relied on the dynamic data structures. Since improvements by using data structures are very naive, we can apply them to various enumeration algorithms. However they have some week points, which are that data structures tend to be complicated, and the computation time to update them may be a bottle neck part of the time complexities of algorithms. Complicated data structures also make algorithms difficult to be implemented.

In the existing studies, many enumeration algorithms are bounded their time complexities by the worst case time of an iteration. The total computation time is often bounded by the product of this worst case computation time and the number of iterations. The bound is often far from the tight bound, if the computation time of each iteration is biased. In this chapter, we introduce an approach which is not based on data structures, and propose several algorithms by using this approach. Our idea is to bound the total computation time by using an "amortized" analysis.

Generally, an enumeration algorithm spends much computation time in few iterations, and few computation time in many iterations. To correct the bias, we consider a "distributing rule" of computation time from the former iterations to the latter iterations. By making a good distributing rule, we can obtain uniform computation time on all iterations. Thus we can bound the time complexity tightly. Such an analysis is called an amortized analysis. The idea is quite fundamental, hence we can apply the approach to some enumeration algorithms, especially binary partition algorithms.

For example, suppose that we have an enumeration tree of a binary partition algorithm. Binary partition algorithms partition the problem in each iteration, hence the number of vertices around the top of the enumeration tree is often small rather than those on the bottom. The generated subproblems are often smaller than the original problem, hence vertices around the top have much computation time than vertices on the bottom. Thus, by distributing the computation time spent on vertices around the top to vertices around the bottom, we can estimate a good amortized time complexity. These properties of enumeration trees are general, thus the technique can be applied to the other types of algorithms satisfying the above conditions.

Let us see the framework of the analysis. Let $T(x)$ be the upper bound of computation time spent on a vertex x of the enumeration tree. We use a parameter T^*. By an amortized analysis, required time for each iteration is bounded by $O(T^*)$, i.e. the total computation time is $O(NT^*)$ where N is the number of output. For candidates of T^*, we can utilize $\max\{T(x)/\bar{D}(x)\}$ over all vertices x. Here $\bar{D}(x)$ is a lower bound of the number of descendants of the vertex x in the enumeration tree. This parameter is practical for some algorithms.

By using these parameters, we distribute the computation time as follows. For the root vertex x of the enumeration tree, we store T^* of $T(x)$ on x, and distribute the rest computation time $T^* - T(x)$ to the children of x by a certain distributing rule. For each child y of x, let $T_p(y)$ denote the computation time distributed by x to y. If $T_p(y) + T(y) > T^*$, then we call $T_p(y) + T(y) - T^*$ *excess time* on y. The child y also stores T^* of computation time $T_p(y) + T(y)$, and distribute the excess time to its children by the distributing rule. We apply the distributing rule to all internal vertices from the top of the enumeration tree in the top-down manner. If some distributing rules satisfy that any leaf of the enumeration tree has at most $O(T^*)$ time after distributing, then the total time complexity is bounded by $O(N \times T^*)$.

In the following sections, we describe three algorithms based on this approach. The first one is an algorithm for connected induced subgraphs. For the problem of enumerating all connected induced subgraphs, a simple reverse search algorithm is proposed by D. Avis and K. Fukuda [4]. We propose a binary partition algorithm for the problem, and analyze the time complexity by the above approach. The algorithm reduces the time complexity from $O(n)$ to $O(1)$ per a connected induced subgraph.

The second one is for enumerating all edge colorings in bipartite graphs. For this problem, Y. Matsui and T. Matsui [59, 60] proposed some polynomial delay algorithms. One of them [60] requires about $O(m \log^3 n)$ time per an edge coloring. They bound the depth and the maximum computation time on a vertex of the enumeration tree. The total computation time is bounded by the product of them, since the leaves of the enumeration tree correspond to all the edge colorings. In this section, we modify their algorithm slightly, and bound the computation time more tightly by using the above analysis. The time complexity is reduced to $O(n)$ per an edge coloring.

The third algorithm is for maximal matchings in bipartite graphs. For the problem, Tsukiyama et al. [52] proposed an algorithm in 1977 which runs in $O(nm^2)$ time per a maximal matching. Our algorithm is based on binary partition. The algorithm partitions the problem to two subproblems of enumerating all maximal matchings covering a specified vertex v and those not covering v. To generate these subproblems, we utilize an enumeration algorithm for maximum matchings in bipartite graphs described in the later section. The algorithm may take $O(n^{1/2}m)$ time for one iteration in the worst case. By some analysis of the enumeration tree, the excess time on some vertices are distributed to the other vertices, and the time complexity is reduced to $O(n)$ per a maximal matching.

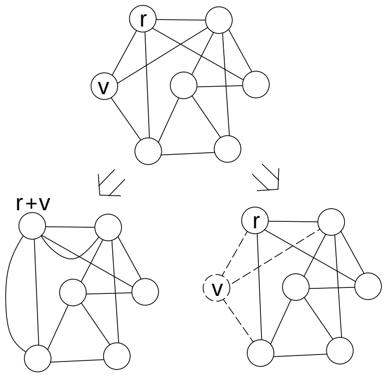

Figure 4.1: Partitioning the problem: The left graph is obtained by contracting v and r, and the right graph is obtained by removing v.

4.1 Enumerating Connected Induced Subgraphs

For an undirected graph $G = (V, E)$, a *vertex induced subgraph* of a vertex set V' is a subgraph whose vertex set is V' and whose edge set is composed of all edges of E connecting two vertices of V'. If a vertex induced subgraph is connected, it is called a *connected induced subgraph*. We consider the problem of enumerating all connected induced subgraphs in a given graph. For this problem, a simple reverse search algorithm was proposed [4]. The time complexity of the algorithm is $O(|V|)$ per a connected induced subgraph. In this section, we propose a simple binary partition algorithm, and bound the time complexity by some analysis.

Let us consider the following simple binary partition method. Suppose that we are given an undirected simple graph G. Multiple edges can be replaced by an edge, hence the simpleness of the input graph does not lose the generality. A connected induced subgraph is induced by a vertex set uniquely. Hence for a specified vertex r, we consider two kinds of connected induced subgraphs not including r, and those including r. The problem is divided into two subproblems of enumerating these subgraphs. We call the former subproblem *type 1*, and the latter *type 2*. The connected induced subgraphs not including r are connected induced subgraphs in $G \setminus r$, hence type 1 subproblems can be enumerated recursively. Here $G \setminus r$ denotes the graph obtained by removing r and all edges adjacent to r from G.

To solve the type 2 subproblem, we consider the other partitioning rule. We choose a vertex v adjacent to r, and consider two subproblems. One of them is to enumerate all connected induced subgraphs including both v and r, and the other is to enumerate those including r and not including v. The latter subproblem is a type 2 subproblem with the graph $G \setminus v$, hence it can be solved recursively. For the former subproblem, we consider the graph obtained by contracting v and r, and replacing multiple edges by an edge. In the graph, all connected induced subgraphs including r is also including v, thus the subproblem is also a type 2 subproblem. Therefore all connected induced subgraphs can be enumerated by a binary partition algorithm efficiently.

Let us consider the enumeration tree of the algorithm. Each vertex of the tree corresponds

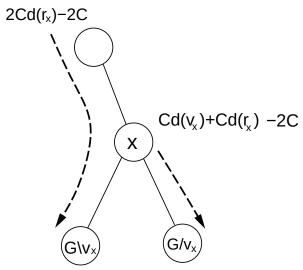

Figure 4.2: The distributing rule: the computation time $Cd(v_x) - 2C$ is distributed to the child G/v_x obtained by contracting v_x and r_x, and the rest computation time $T_p(x) - 2C$ is distributed to the child inputting $G \setminus v_x$.

to each iteration of the algorithm, and an edge connects two vertices corresponding to two iterations if and only if one of the iterations is called in the other. Suppose that the algorithm inputs a graph, and chooses a vertex r_x in an iteration x if x corresponds to a type 1 subproblem. Otherwise, we suppose that the algorithm inputs a graph and a vertex r_x, and chooses a vertex v_x. We denote the degree of v_x and r_x in the input graph by $d(v_x)$ and $d(r_x)$. The algorithm generates two subproblems by removing r_x, or removing v_x and contracting r_x and v_x. Thus, an iteration takes $Cd(r_x)$ or $C(d(v_x) + d(r_x))$ time. Since any internal vertex has two children, the number of vertices in the enumeration tree does not exceed twice the number of connected induced subgraphs. Therefore a simple upper bound is established by $\mathrm{O}(|V|N)$ for the time complexity. Here N denotes the number of connected induced subgraphs included in G. This bound is very simple, and not tight. By using a distributing rule of excess time, we can reduce it as follows.

We set T^* to $4C$, and consider a distribution rule of excess computation time. For a vertex x of the enumeration tree, we store $2C$ of computation time spent on x, and distribute the rest computation time to its child inputting the graph $G \setminus r$ or $G \setminus v$ in the top down manner. Suppose that $T_p(y)$ denotes the computation time which a child y receives from its parent. We store $2C$ of $T_p(y)$ on y, and distribute $T_p(y) - 2C$ to the child of y inputting G or the graph obtained by contracting r and v.

By this distributing rule, we have T^* on any internal vertex. The excess time $T_p(x) + C(d(r_x) - 4)$ or $T_p(x) + C(d(v_x) + d(r_x) - 4)$ is distributed completely. In the distributing rule, we can state the following property.

Property 2 *In the distributing rule, for any vertex x of the enumeration tree, we have $T_p(x) = 0$ if x corresponds to a type 1 subproblem, otherwise $T_p(x) \leq 2Cd(r_x)$.*

Proof : We assume that the condition holds for a vertex x. If x corresponds to a type 1 subproblem, then the child y of x corresponding to a type 1 subproblem satisfies $T_p(y) = 0$. The other child y' of x also satisfies that $T_p(y') = Cd(r_x)$. If x corresponds to a type 2 subproblem, then for the child y obtained by removing v_x, we have $d(r_y) = d(r_x) - 1$. For the other child y', we also have $d(r_{y'}) \geq \frac{d(v_x)+d(r_x)-2}{2}$. Therefore, both children y receives at most $2Cd(r_y)$ of computation time from x. Since the root vertex of the enumeration tree receives no computation time, all vertices of the enumeration tree satisfy the condition. ∎

From this property, any internal vertex of the enumeration tree stores $4C$ of computation time on it, and any leaf receives no computation time since $d(r_x) = 0$ holds for any leaf x. Therefore we have the following theorem.

Theorem 5 *All connected induced subgraphs can be enumerated in $O(N)$ time and $O(m+n)$ memory where N is the number of them in the given graph.*

Proof: We already see the correctness of the algorithm and its time complexity. The memory complexity is obviously $\mathrm{O}(m + n)$. We show the method for outputting all the connected induced subgraphs in $\mathrm{O}(1)$ time per one. In each iteration of the algorithm, we output all the contracted or removed edges when a subproblem occurs. By this, we can obtain a subgraph at the leaf of the enumeration tree from the previous outputted edges. Since the number of these output edges does not exceed the computation time of the algorithm, the total time to output is $\mathrm{O}(N)$. ▮

4.2 Enumerating Edge Colorings in Bipartite Graphs

In this section, we propose an algorithm for enumerating all edge colorings in a bipartite graph $G = (V_1, V_2, E)$. We allow multiple edges in the graph. An edge coloring of a graph is a way of coloring all edges in the graph such that no pair of edges with the same color share their endpoints. We denote an edge coloring C by $\{M_1, M_2, ..., M_k\}$ where M_i is the set of edges with the same color. All edges incident to a vertex have distinct colors, hence each M_i forms a matching. An edge coloring with the minimum number of colors is called a minimum edge coloring. In 1916, König [31] proved that a minimum edge coloring of a bipartite graph uses exactly Δ colors where Δ denotes the maximum number of the degree among all vertices in the graph. Here we treat only those minimum edge colorings, and simply call them edge colorings. The current best time complexity algorithm for finding a minimum edge coloring in a bipartite graph is proposed by Cole and Hopcroft [8]. Their algorithm runs in $\mathrm{O}(m \log n + n(\log n)(\log^3 \Delta))$ time and $O(m+n)$ space.

There are many applications for the edge coloring problem, such as the scheduling problems and the timetable problems. They have been discussed in [5], [10], [25] and [61]. In a bipartite graph whose vertex sets are corresponds to workers and jobs, edge colorings of the graph correspond to a schedule. Optimization problems for schedules are often hard, hence enumeration is a naive and simple approach for the problems.

In this section, we consider the problem of enumerating all minimum edge colorings of a given bipartite graph. Since an edge coloring is a set of matchings with Δ distinct colors, there are many colorings which are composed of the same matchings. We consider all these to be the same. Some algorithms for this problem have been proposed by Y. Matsui and T. Matsui [59, 60]. Their current best one runs in $\mathrm{O}(m \log n + n(\log n)(\log^3 \Delta) + \min\{n^2 + m, m \log n + n(\log n)(\log^3 \Delta)\}N)$ time and $\mathrm{O}(m\Delta)$ memory space. Here N denotes the number of minimum edge colorings in the given graph. They discussed a simple basic algorithm, and obtain their algorithm by improving the basic algorithm. We improve the basic algorithm by the other manner. The algorithm in [60] seems to terminate in shorter time since its enumeration tree is quite balanced, and the worst case of the time complexity may occur in the very rare cases. Thus we add some modifications to avoid these biased rare cases. Our algorithm is quite simple, although its time complexity is bounded by $\mathrm{O}(nN)$ with compact output technique. We also show a technique for saving the memory space. Our improvements also reduce the memory complexity to $\mathrm{O}(m+n)$.

4.2.1 Framework of Enumeration Algorithm

In this subsection, we describe the framework of the basic algorithm. Before that, we show a method for finding a minimum edge coloring of G. By König's theorem, an edge coloring with Δ colors exists for any bipartite graph. Hence, in an edge coloring, each maximum degree vertex of G is incident to all Δ colored edges. Conversely, for any matching composing the edge coloring, any maximum degree vertex is incident to an edge of the matching. We call those matchings which covers all maximum degree vertices *covering matchings* of G. Any matching composing a minimum edge coloring is a covering matching.

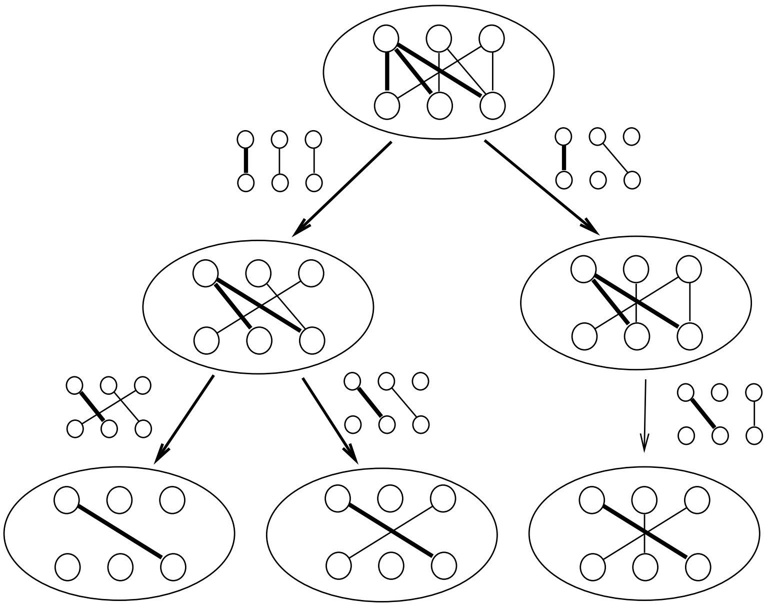

Figure 4.3: The way of enumerating edge colorings. Emphasized lines are edges colored at the beginning of the enumeration.

This property motivates the following simple approach to find a minimum edge coloring of a bipartite graph. For a covering matching M of G, the subgraph $G \setminus M$ is a bipartite graph with the maximum degree $\Delta - 1$. In the subgraph $G \setminus M$, we always have an edge coloring with $\Delta - 1$ colors. By adding M to the coloring, we obtain an edge coloring of G with Δ colors. Therefore by finding a covering matching and removing it iteratively, we can find a minimum edge coloring of G. An algorithm is proposed in [8] for finding a covering matching in $\mathrm{O}(m \log n)$ time and $\mathrm{O}(m + n)$ space.

The simple algorithm is based on the above method. Since any edge coloring $C = \{M_1, M_2, ..., M_\Delta\}$ is found with the above method by removing the covering matchings $M_1, M_2, ..., M_\Delta$ iteratively, we can enumerate edge colorings by using binary partition method. For the given graph G, we list all covering matchings of G. For each M of these, we enumerate all edge colorings including M. This is done by recursive calls with the subgraph $G \setminus M$.

Since an edge coloring is a set of matchings with Δ distinct colors, there are many colorings which are composed of the same matchings with different colors. We consider all these to be the same. If two covering matchings M and M' have no intersection, the same colorings $\{M, ..., M', ..., M_\Delta\}$ and $\{M', ..., M, ..., M_\Delta\}$ may be enumerated by the above method. To avoid this, we choose a maximum degree vertex v^* and color edges incident to v^* with Δ different colors before starting the enumeration. In each recursive call, we choose the minimum index edge among these colored edges, and list only covering matchings including the edge. The intersection of any pair of covering matchings including an edge e is not empty. Thus, we do not find redundant colorings.

To list covering matchings, an algorithm is proposed in [60]. Its time complexity is $\mathrm{O}(m+n)$

per one additional covering matching. Its space complexity is $\mathrm{O}(m+n)$. Our algorithm is based on their algorithm, hence here we explain it.

Their enumeration algorithm also utilizes the binary partition method. For a given graph G and a covering matching M, we firstly find a covering matching $M' \neq M$. Next we choose an edge e included in $M \setminus M'$. We partition the problem into two subproblems, which are to enumerate all the covering matchings including e and to enumerate those not including e. The set of covering matchings not including e is equal to the set of covering matchings of $G \setminus e$. Similarly, the set of covering matchings including e is equal to the set of matchings of $G^{+}(e)$ plus e which cover all the maximum degree vertices of G. Therefore these subproblems are solved by enumerating all the matchings of the subgraph $G^{+}(e)$ and $G \setminus e$ covering all the maximum degree vertices of G. Here $G^{+}(e)$ is the subgraph of G obtained by removing e, both ends of e and all edges sharing their endpoints with e. In each recursive call, we enumerate only covering matchings including a colored edge e^*. To enumerate such matchings, we also use the subgraph $G^{+}(e^*)$.

For a given covering matching, the algorithm requires another covering matching. To find another covering matching, we use the following lemma which is stated in [60].

Lemma 15 *For a covering matching M, a matching $M' \neq M$ is a covering matching if and only if $M \triangle M'$ is composed of cycles and paths connecting two non-maximum degree vertices.*

▮

Since edges of M do not share their endpoints with each other, edges of M appear in these paths and cycles alternately. The end edges of the paths are not adjacent to edges of M. We call them *alternating paths* and *alternating cycles* with respect to M. If we have an alternating cycle or path, we can obtain the other matching by exchanging edges along the cycle or path. An alternating path respect to M generates a covering matching if and only if both its endpoints are not maximum degree.

For finding these, a linear time algorithm is proposed in [60]. In our algorithm, we utilize a directed graph $\hat{D}(G,M)$. The arc set of $\hat{D}(G,M)$ is given by orienting edges of G from V_1 to V_2 if they are in M, and otherwise in the opposite direction. We contract all uncovered vertices of V_1 by M to a vertex s_1. If V_1 has no uncovered vertex, then we make a new vertex s_1 in V_1. Similarly, we obtain a vertex s_2 of V_2. We make arcs from all covered non-maximum vertices of V_2 to s_1, and from s_2 to all covered non-maximum vertices of V_1. We also make an arc $\hat{e}$ from s_1 to s_2. A directed cycle of $\hat{D}(G,M)$ corresponds to an alternating cycle if it does not include $\hat{e}$, and otherwise corresponds to an alternating path connecting two non-maximum vertices. Hence, by finding a directed cycle of $\hat{D}(G,M)$, we can obtain the other covering matching in $\mathrm{O}(m+n)$ time.

Our enumeration algorithm for edge colorings is obtained by adding some modifications to this basic algorithm. We show them as follows. If the given graph G satisfies $\sigma(G) \leq 2$ or $\Delta(G) \leq 2$, we can find the initial covering matching directly in $\mathrm{O}(m+n)$ time. Hence we do not spend $\mathrm{O}(m \log n)$ time in this case. If the given graph contains only one covering matching including the colored edge, we enumerate all the edge colorings in the graph by brute-force method. In that case, all the edges of the graph incident to the edge, hence we can enumerate all edge colorings easily. We prove it as follows .

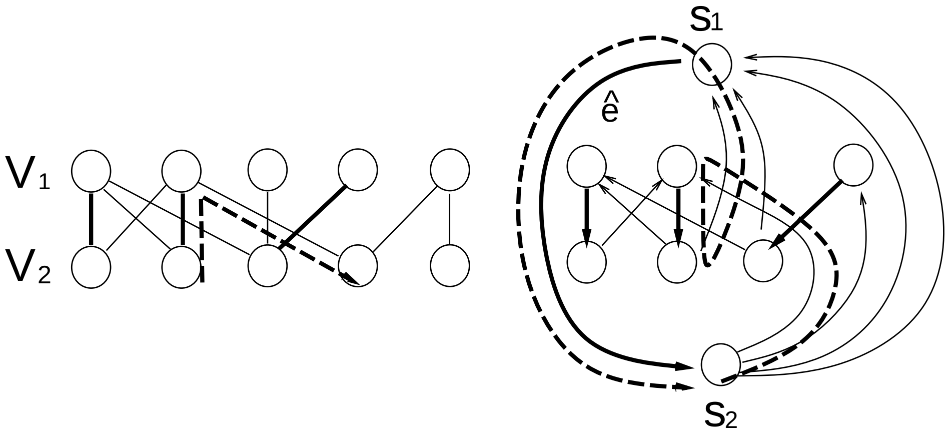

Figure 4.4: $\hat{D}(G, M)$ and an alternating path.

Lemma 16 *Assume that $\Delta \geq 3$. Then for an edge e incident to a maximum degree vertex, there exists only one covering matching of G including e if and only if all edges of G share at least one of their endpoints with e.*

Proof: The 'if' part is obvious, thus we prove 'only if' part by its contraposition. Let M be a covering matching containing e. From the assumption, the degree of any maximum degree vertex v of G is at least 2 in $G^+(e)$ except for the endpoints of e. If an edge is not adjacent to e, $G^+(e)$ contains at least one edge. We recall that G contains another covering matching including e if and only if $G^+(e)$ contains an alternating path or cycle.

Let v be a vertex of $G^+(e)$ which is non-maximum degree in G. If $G^+(e)$ includes no such vertex, let v be a maximum degree vertex of $G^+(e)$. We apply the depth-first search to $\hat{D}(G^+(e), M)$ with starting from v. As maximum degree vertices of G are incident to at least two edges in $G^+(e)$, the depth-first search does not terminate until it reaches to another non-maximum degree vertex of G or a vertex which has already been visited. In both cases, we have an alternating cycle or an alternating path connecting two non-maximum vertices. Therefore G includes another covering matching including e. ▮

We now describe the details of our algorithm.

```
ALGORITHM: ENUM_EDGE_COLORING($G = (V_1, V_2, E)$)
Step 1: Choose a maximum degree vertex $v$ and color edges incident to $v$ with $\Delta(G)$
        distinct colors.
Step 2: Call ENUM_EDGE_COLORING_ITER($G$).
```

```
ALGORITHM: ENUM_EDGE_COLORING_ITER($G = (V_1, V_2, E)$)
Step 1: Select a colored edge $e^*$.
Step 2: If the covering matching including $e^*$ is unique, output all edge colorings by
        brute-force method. Stop.
Step 3: If $\sigma(G) \leq 2$ or $\Delta(G) \leq 2$, then find a covering matching $M$ of $G$
        including $e^*$ by brute-force method. Go to Step 4.
Step 4: Find a covering matching $M$ in a graph obtained by deleting
```

```
        all edges not incident to maximum degree vertices from $G^+(e^*)$.
Step 5: List all covering matchings including $e^*$ by
        Enum_Covering_Matchings ( $G, G^+(e^*), M$ ).

ALGORITHM: Enum_Covering_Matchings ( $G, G', M$ )
Step 1: If there is no other covering matching in $G'$, then call
        Enum_Edge_Coloring_Iter($G \setminus M$). Stop.
Step 2: Find a directed cycle $C$ of $\hat{D}(G', M)$, and
        generate a covering matching $M' \neq M$.
Step 3: Choose an edge $e \in C \cap M$ corresponding to an edge of $G$.
Step 4: Call Enum_Covering_Matchings ( $G, G^+(e), M$ ), and
        Enum_Covering_Matchings ( $G, G \setminus e, M'$ ).
```

The algorithm Enum_Edge_Coloring spends $O(m+n)$ time except for the time spent by Enum_Edge_Coloring_Iter. Since there exists a covering matching such that all its edges are incident to some maximum degree vertices, we can delete all edges not incident to any maximum degree vertices before finding a covering matching in Step 4. Therefore the time to find a covering matching in Step 4 is bounded by $O(m + \Delta(G)\sigma(G)\log n)$. The other parts of the algorithm take $O(m+n)$ time per a recursive call, hence we can estimate a simple upper bound $O((m \log n)\Delta N)$ of the time complexity of our algorithm. Here N denotes the number of edge coloring in G. In the next subsection, we analyze the time and space complexities to reduce them.

4.2.2 An Analysis of the Time Complexity of the Algorithm

Our algorithm is simpler than the algorithm of [60] and runs in shorter time. The most important part of our results is that we establish a new analysis of the time complexity. Our algorithm may spend $O(nm \log n)$ time between outputting two edge colorings. Hence it may seem to run in $\Theta((nm \log n)N)$ time in the worst case. To show that it terminates in shorter time, we utilize amortized analysis with some properties.

To simplify the analysis, we introduce the following rooted tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ called the *enumeration tree*, which captures the movement of the algorithm. The root vertex of the tree corresponds to the start of the algorithm, which is the first time that Enum_Edge_Coloring_Iter is called. Other vertices of $\mathcal{T}$ correspond to each recursive call. Two vertices of $\mathcal{T}$ are connected by an edge if the recursive call corresponding to one of them occurs in the other. If a recursive call generates no recursive call, then the algorithm outputs some edge colorings. Hence each leaf of $\mathcal{T}$ corresponds to edge colorings (see Figure 4.3). For a vertex x of $\mathcal{T}$, let G_x denote the graph inputed by the recursive call corresponding to x.

The enumeration tree has some good properties. We can see one of them by stating the following lemma.

Lemma 17 *A graph G with $\Delta = 3$ and $\sigma(G) \geq 3$ has at least $\lceil n/6 \rceil$ edge colorings.*

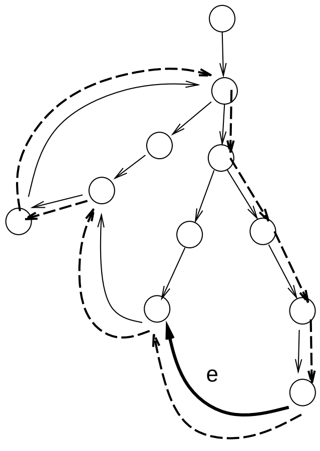

Figure 4.5: A cycle generated by non-back arc e in H.

Proof: For a covering matching M, we show that $\hat{D}(G,M)$ includes at least $|E(\hat{D}(G,M))| - |V(\hat{D}(G,M))|$ distinct directed cycles. Let H be a strongly connected component of $\hat{D}(G,M)$, and T be a depth-first-search tree of H. We assume that the search traces an arc from its tail to its head. We define indices of vertices in T by the post-ordering of the depth-first-search. For each arc f of T, we define the index of f by 0 if f is in T, and otherwise by the index of the head of f. We show that there exist distinct directed cycles in H for all arcs of $H \setminus T$.

For a back arc e of T, we can obtain a directed cycle by adding e to T. In the cycle, the maximum index arc is e. For a non-back arc e, let P be a directed path from the head of e to the tail of e. As T is a depth-first-search tree, P contains an ancestor of the tail of e. Let v be the first vertex appearing in P among ancestors of the tail of e. By connecting, (1) the directed path from the head of e to v in P, (2) e, and (3) the path from v to the tail of e in T, we can generate a directed cycle in H.

In the cycle, e is the maximum index arc. We prove it by the contradiction. We assume that there is an arc in P with a larger index than e. Then we have an arc e' in P with a larger index than e on its tail and a smaller index than e on its head. Because of the post-ordering, e' is a back arc. From the way of constructing P, the head of e' is not an ancestor of the tail of e. But, for any vertex with an index smaller than the tail of e, its ancestor has a larger index than the tail of e if the ancestor is also an ancestor of the tail of e. Therefore it contradicts the assumption.

For each arc $e \in H \setminus T$, we have generated a directed cycle such that the maximum index arc in the cycle is e. Therefore there are $|E(H)| - |E(T)| = |E(H)| - |V(H)| + 1$ directed cycles. Since heads of arcs in a directed cycle are different from each other, those generated cycles are distinct.

Since $\hat{D}(G,M)$ is obtained by adding s_1, s_2, $\hat{e}$ and arcs for each non-maximum vertex, $\hat{D}(G,M)$ has at most $n+2$ vertices and at least $m + (n - \sigma(G)) + 1$ edges. Since $\Delta(G) = 3$, m is at least $(3\sigma(G) + (n - \sigma(G)))/2 = \sigma(G) + n/2$. From the above observation, $\hat{D}(G,M)$ contains at least $\sigma(G) + n/2 + (n - \sigma(G)) + 1) - (n+2) + 1 = n/2$ directed cycles, hence G has at least $n/2 + 1$ covering matchings. Since any covering matching is included in some edge colorings of G and any edge coloring of G includes exactly 3 covering matchings, G

includes at least $\lceil n/2+1 \rceil /3 \geq n/6$ edge colorings. ▮

Now we show that the algorithm takes $O(nN)$ time. Firstly we bound the time complexity except for outputting. The algorithm ENUM_EDGE_COLORING terminates in $O(n)$ time per an ourputted edge coloring if no recursive call occurs. Otherwise, it generates at least two subproblems. Hence all leaves of the enumeration tree have $O(n)$ computation time.

For a vertex x of the enumeration tree, we denote its input graph by G_x. The computation time on x is $O(|E(G_x)| + \Delta(G_x)\sigma(G_x)\log\sigma(G_x))$ if both $\Delta(G_x)$ and $\sigma(G_x)$ are not less than 3, and otherwise $O(|E(G_x)|)$. We also spends $O(|E(G_x)|)$ time per a child of x. We suppose that the computation time is spent on each child of x. Since the number of edges differ at most n between the input of x and that of the parent of x, the time complexity of an iteration is bounded by $C(|E(G_x)| + \Delta(G_x)\sigma(G_x)\log\sigma(G_x))$ with a certain constant C. We denote it by $T(x)$. We distribute this computation time to the children of x uniformly if $\Delta(x) > 3$, and to all descendants of x uniformly if $\Delta(G_x) = 3$. The distribution is done in the top-down manner. Since a vertex x with $\Delta(G_x) < 3$ spends $O(n)$ time, we do not distribute the computation time to the children of x.

For a child y of x, we denote the computation time distributed from x to y by $T_p(y)$. We bound $T_p(y)$ by using $T(y)$. Since x has at least two children, $T_p(y)$ is bounded by $(T_p(x)+T(x))/2$. Since $|E(G_x)| - |E(G_y)| \leq n/2$ and $\Delta(G_x) = \Delta(G_y)+1$, if $\sigma(G_x) \geq 3$, then we have $T(x) \leq C(|E(G_y)| + \Delta(G_y)\sigma(G_y)\log\sigma(G_y) + 2n)$. Let $x_0 = y, x_1, .., x_k$ be the sequence of the iterations appearing in the path from y to the root in the enumeration tree. x_{i+1} is the parent of x_i. From the above inequations, we have

$$\begin{aligned} T_p(y) &\leq \sum_{i=1}^{k} C(|E(G_{x_i})| + \Delta(G_{x_i})\sigma(G_{x_i})\log\sigma(G_{x_i}))/2^i \\ &\leq \sum_{i=1}^{\infty} C(|E(G_y)| + \Delta(G_y)\sigma(G_y)\log\sigma(G_y) + (2n)i)/2^i \\ &\leq C(|E(G_y)| + \Delta(G_y)\sigma(G_y)\log\sigma(G_y) + 4n). \end{aligned}$$

Hence an iteration y with $\Delta(y) = 3$ receives at most $C(2n + 3\sigma(G_y)\log\sigma(G_y) + 4n)$. By distributing it to all descendants of y, each descendant receives at most $O(\log n)$ computation time since y has at least $O(n)$ descendants from Lemma 17. Hence, after the distribution, we have $O(n)$ computation time on any vertex.

Next we bound the outputting time. In general, an edge coloring is outputted by all covering matchings composing it. It takes $\Theta(m)$ time. Since the algorithm terminates in strictly shorter time than outputting time, the outputting process is the bottleneck of the time complexity. In some other enumeration problems, *compact output methods* have been studied to decrease the time to output in [32, 49, 13]. To enumerate objects, they output only differences between the current outputting object and the one outputted just before. Though this may still require the same time per an object in the worst case, they succeed in decreasing the total size of outputs.

In our algorithm, the size of difference between an output x and next one y is bounded by n times the number of edges of the enumeration tree which the algorithm traces between outputting x and outputting y. As the algorithm traces the enumeration tree by the depth-first search manner, an edge is traced at most twice. From these analysis, we obtain the following theorem.

Theorem 6 *The algorithm* ENUM_EDGE_COLORING *enumerates all edge colorings of G in $O(nN)$ time and $O(\Delta(m+n))$ space.*

We have already shown the correctness of the algorithm and the time complexity. In each level of the recursive calls, the algorithm spends $\mathrm{O}(m+n)$ memory space for enumerating covering matchings. They are stored when a new covering matching is found and a recursive call occurs. Hence the memory complexity is bounded by $\mathrm{O}(\Delta(m+n))$ since the depth of the enumeration tree is at most Δ.∎

In the next subsection, we explain the technique for reducing the memory complexity to $\mathrm{O}(m+n)$.

4.2.3 A Technique for Reducing the Space Complexity

In this subsection, we show the technique for reducing the space complexity. In the previous section, we described an algorithm for finding all covering matchings which requires $\mathrm{O}(m+n)$ extra memory space in each level of the enumeration tree. Moreover, since our enumeration algorithm is composed of two nested enumeration algorithms which are for edge colorings and for covering matchings, the depth of the recursion may reach to $\Theta(\Delta m)$. These are the memory-consuming parts.

To reduce the memory complexity, we propose an algorithm for covering matchings without any recursion. When a covering matching M is found, the algorithm stops the process, and generates a recursive call with the graph obtained by removing M. After the recursive call terminates, the algorithm restarts the process. If the algorithm requires at most additional $\mathrm{O}(|M|)$ memory space to restart, the total memory complexity will be $\mathrm{O}(m+n)$.

To construct the algorithm, we consider indices for all edges of G. In each iteration, we find the minimum index edge among edges included in a covering matching and not included in the other covering matching, and partition the problem by using the edge. This algorithm can be constructed without any recursion.

To remove the recursion from the algorithm, we use a counter i. Let M be a covering matching of G. Let E_i be the set of edges whose indices are less than i. We define $\mathcal{M}(M,i)$ by the set of covering matchings M' satisfying that $M \cap E_i = M' \cap E_i$. At the beginning of the algorithm, we set the counter i to 1. The algorithm finds the minimum index edge e_j among all edges $e_k, k \geq i$ such that there are both covering matchings including e_k and not including e_k. The edges e_k are included in some alternating cycles or alternating paths, hence this process can be done by applying a strongly connected component decomposition algorithm to $\hat{D}(G,M)$. The decomposition takes $\mathrm{O}(m+n)$ time. If M does not include e_j, then we generate a covering matching including e_i by using an alternating cycle, and set M to the matching.

After finding the above minimum index edge e_j, we put a mark to e_j, and start enumerating all the covering matchings including e_j. Instead of generating a recursive call, we set i to $j+1$, and enumerate only covering matchings in $\mathcal{M}(M,i)$. To find only these matchings, we remove all arcs of $\hat{D}(G,M)$ which are corresponding to edges of E_i, and find a directed cycle.

Let i^* be the maximum index such that $e_{i^*} \in M$ is marked. In the above process, if we can find neither alternating cycle nor path, then there is no more covering matching in $\mathcal{M}(M,i)$. In this case, we delete the mark of e_{i^*}, and set M to a covering matching of $\mathcal{M}(M,i^*)$ not including e_{i^*}. Note that there is at least one matching of $\mathcal{M}(M,i^*)$ not including e_{i^*} from the rule of marking. After this, we start enumerating all the covering matching not including e_{i^*} by setting i to i^*+1.

Since the algorithm takes $O(m+n)$ time until marking an edge, the time complexity of the algorithm is $O(m+n)$ per a covering matching. Note that the process of marking edges corresponds to generating two recursive calls.

This algorithm requires memory space for G, M, i, and the marks of edges of M. When we find a new covering matching M', we generate a subproblem of enumerating edge colorings of $G \setminus M$. After enumerating them, we restore all those variables, G, M, i, and the marks of edges of M. The subproblem inputs $G \setminus M$, hence we can construct G by adding M to $G \setminus M$. The others take at most $O(|M|)$ memory space. Hence the total memory complexity required to enumerate all the edge colorings is $O(m+n)$.

Finally, we describe the details of our algorithm.

ALGORITHM: ENUM_COVERING_MATCHINGS_2 (G,M)

```
Step 1: Set $i$ to 1.
Step 2: Set $G'$ to the graph obtained by removing edges of $G$ with indices smaller than $i$.
Step 3: Construct $\hat{D}(G',M)$.
Step 4: Find the minimum index edge $e_j$ of $G'$ such that $e_j$ is included in an
        alternating cycle $C$.
Step 5: If there is such an edge $e_j$, then do Step 5a and Step 5b.
Step 5a:  Find the covering matching including $e_j$ by using $C$. Set $M$ to the matching.
Step 5b:  Put a mark to $e_j$. Set $i$ to $j+1$. Go to Step 2.
Step 6: Find the maximum index marked edge $e_{i^*}$, and delete the mark.
        If there is no marked edge, stop.
Step 7: Set $G'$ to the graph obtained by removing all edges of $E_{i^*}$ from $G$.
Step 8: Generate a covering matching not including $e_{i^*}$ with an alternating cycle of $G'$.
Step 9: Set $i$ to $i^*+1$. Go to Step 2.
```

Theorem 7 *The algorithm* ENUM_EDGE_COLORING *with* ENUM_COVERING_MATCHINGS_2 *enumerates all edge colorings of G in $O(nN)$ time and $O(m+n)$ space.*

∎

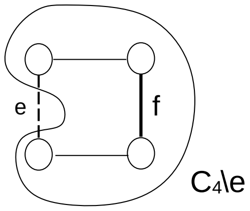

Figure 4.6: A maximal matching $\{f\}$ in $C_4^-(e)$ is not maximal in C_4.

4.3 Enumerating Maximal Matchings in Bipartite Graphs

For a given bipartite graph G, a *matching* is an edge set satisfying that no pair of its edges share their endpoints. A matching is called *maximal* if it is included in no other matching properly. In this section, we consider the problem of enumerating all maximal matchings in a given bipartite graph. In 1977, Tsukiyama et al. [52] proposed an algorithm for enumerating all maximal stable sets in a general graph. Since a matching in a graph is a stable set in the line graph of the graph, maximal matching can be enumerated by their algorithm. The algorithm runs in $\mathrm{O}(nm)$ time per a stable set, thus we can enumerate maximal matchings in $\mathrm{O}(nm^2)$ time per a maximal matching.

Our algorithm for maximal matchings is based on binary partition. It partitions the problem into two subproblems. Both subproblems are composed of some numbers of problems of enumerating maximal matchings in some subgraphs. To construct all these subproblems, we use an enumeration algorithm for maximum matchings in bipartite graphs. The algorithm is described in the later chapter. Our algorithm in this section can enumerate only bipartite maximal matchings, while the algorithm of Tsukiyama et al. works in general graphs. However our algorithm runs in the quite shorter time than theirs. For maximal matchings in general graphs, we also have an improved version of their algorithm described in the later chapter.

For the enumeration problem of perfect matchings, a simple binary partition algorithm works well. Because we can easily generate the subproblems of enumerating all perfect matchings including an edge e, and also those not including e. However we can hardly obtain naive binary partition algorithms for maximal matchings. We can easily construct the subproblem of enumerating all maximal matchings including e by using the subgraph $G^+(e)$, although we cannot enumerate those not including e so easily by using $G \setminus e$. We show a simple example. Let us consider the graph C_4 which is a cycle with the length 4. Let e be an edge of C_4 and f be an edge not adjacent to e. We can easily enumerate maximal matchings including e by the subproblem with $C_4^+(e)$, but cannot enumerate maximal matchings not including e by the subproblem with $C_4 \setminus e$. In $C_4 \setminus e$, $\{f\}$ is a maximal matching of the graph, but not in C_4 (see Figure 4.6). Therefore we consider the other way of partitioning the set of maximal matchings.

Let us consider two types of maximal matchings, on which covers a vertex v and the others which does not. All maximal matchings covering v include exactly one edge incident to v. All maximal matchings not covering v do not include these edges, and cover all vertices

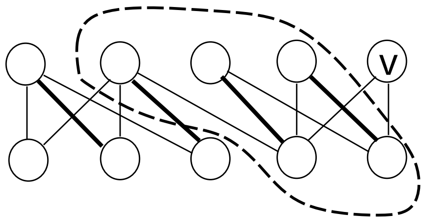

Figure 4.7: A maximal matching and a vertex v. The subgraph circled by dotted line is $\hat{G}$.

adjacent to v from the maximality. Enumerating all maximal matching covering v is easy. It is done by enumerating all maximal matchings in $G^+(e)$ for all edges e incident to v. Since a maximal matching in $G^+(e)$ plus e forms a maximal matching in G, and conversely all maximal matchings including e contain maximal matchings of $G^+(e)$. For matchings not covering v, we consider $\hat{G}$ which is the subgraph composed of all edges incident to vertices adjacent to v, except for edges incident to v (see Figure 4.7). Any maximal matching not covering v includes a matching of $\hat{G}$ with the cardinality $d(v)$. Here $d(v)$ denotes the degree of v, which is the number of vertices adjacent to v. Conversely, for any matching $\hat{M}$ of $\hat{G}$ with the cardinality $d(v)$, some maximal matchings in G include it. Thus, by finding all matchings with the cardinality $d(v)$ and enumerating maximal matchings including them, we can enumerate all maximal matchings not covering v. To enumerate maximal matchings including $\hat{M}$, we remove all edges adjacent to edges of $\hat{M}$ from G, and enumerate all maximal matchings in the obtained graph.

Now we can enumerate all maximal matchings by finding all maximal matchings covering v and not. In the following, we describe the details of the algorithm.

```
ALGORITHM: ENUM_BIPARTITE_MAXIMAL_MATCHINGS ($G$)
Step 1: If the degrees of all vertices in $G$ are 0 or 1,
        output the unique maximal matching of $G$ and stop.
Step 2: Choose a vertex $v$ with the degree at least 2.
Step 3: For each edge $e$ in $G$ incident to $v$, enumerate all maximal matchings including $e$
        by recursive calls with $G^+(e)$.
Step 4: Find a maximum matching $M$ in $\hat{G}$. If $|M| = d(v)$, then enumerate all maximum
        matchings in $\hat{G}$.
Step 5: For each maximum matching, enumerate all maximal matchings including it by
        a recursive call.
```

The algorithm spends $O(n)$ time in Step 1 and 2, and $O(n)$ time in Step 3 and 5 per a recursive call. We spend $O(d(v)m)$ time to find a maximum matching of $\hat{G}$ in Step 4. After finding a maximum matching, the enumeration of all the other maximum matchings can be done in $O(n)$ time per a maximum matching. We will show the algorithm in the later chapter. To generate a subproblem by a maximum matching M, we construct a subgraph G' by removing all edges adjacent to edges of M. We may spend $O(m)$ time to construct G', but we can reduce the time by constructing G' from the graph which the subproblem generated just before input. We denote the maximum matching generating the subproblem by M'.

The time to construct G' is now depend on the number d of the vertices which are covered exactly one matching among M and M'. We take time proportional to $d \times n$ time. The enumeration of maximum matchings is done by finding alternating cycles and paths. Since we find just one alternating path or cycle in each iteration, the total number of alternating paths and cycles found by the enumeration algorithm does not exceed the twice the number of maximum matchings. By exchanging the edges along an alternating cycle or path, at most two vertices change from uncovered to covered or vice versa. Hence the total time to generate the graphs which subproblems input is $\mathrm{O}(n)$ per a subproblem.

From the above observation, the algorithm seems to spend $O(d(v)m)$ time per a maximal matching. We will show that it terminates in shorter time. To bound the time complexity, we distribute the computation time on each vertex to its children. For a vertex x of the enumeration tree, let G_x be the input graph of x, and $d(v_x)$ be the degree of the vertex chosen in Step 2. We call the children corresponding to the subproblems generated in Step 3 *type 1* and the others *type 2*. We suppose that the computation time on each vertex x is bounded by $Cd(v_x)|E(G_x)|$ and Cn per a type 2 child of x. We set T^* to $3Cn$, and distribute excess computation time. We distribute $C(|E(G_x)| + n)$ of excess time to all the type 1 children, and Cn time to all the type 2 children uniformly. Since each vertex x has $d(v_x)$ type 1 children, the excess time on x is distributed completely.

A child y of x receives at most $C(|E(G_y)| + n)$ computation time from x. We distribute the computation time to all type 1 children of y uniformly. Since x has at least two type 1 children, we have $C(d(v_x)|E(G_x)|+n) \geq 2C(|E(G_y)|+n)$. The ith vertex in the path from y to the root of the enumeration tree input a graph with at most $|E(G_y)| + (2i)n$ edges. Since $\sum_i C(|E(G_y)| + (2i)n)/2^i \leq 4C(|E(G_y)| + 4n)$, we have $T_p(y) \leq 2C(|E(G_y)| + 4n)$ for any vertex y of the enumeration tree. Thus, after the distribution, each leaf has at most $4Cn$ computation time on it. Now we have the following theorem.

Theorem 8 *Maximal matchings in a bipartite graph can be enumerated in $O(nN)$ time and $O(m + n)$ space.*

▮

Chapter 5

A Speeding Up Technique "Trimming and Balancing"

As we mentioned in the previous chapters, speeding up algorithms is an important and interesting part of the studies on enumeration problems. Speeding up is also important for general algorithms, thus numerous techniques for speeding up have been also studied for various algorithms. There also have been proposed several generalized techniques and frameworks for speeding up. In the certain sense, the history of improvements of algorithms can be considered as the history of speeding up, especially the worst case time complexity for their input sizes. In the studies on the enumeration, we have also proposed many algorithms and their improvements, especially for spanning trees and paths [44, 23, 32, 37, 48, 49, 33, 16, 14, 53, 38, 13], although neither generalized technique nor framework is proposed for these improvements. Almost all fast enumeration algorithms are improved by speeding up their iterations with some data structures. Hence if we can not speed up iterations, we may obtain no fast algorithm. In this chapter, which is the main of this thesis, we propose a new approach "trimming and balancing" for speeding up enumeration algorithms. This approach is not based on data structures, hence we can apply it to many enumeration algorithms which have not been improved in the existing studies. Some our algorithms with the worst case time complexity of an iteration larger than the original one often attain a better upper bound of the time complexity than the original.

Our approach in this chapter is simple. It is based on mainly some analysis and slight modifications of the enumerating methods. In the previous chapter, we have shown some enumeration algorithms improved by some techniques not depend on data structures. The technique of the analysis is simple and naive, but it can not be applied to all algorithms. In the case that the structure or the computation time on enumeration trees is biased, the analysis may fail. Let us consider the following two cases of such failures.

The first case (1) is that the computation time on a parent is quite larger than its children. If the children are leaves, then the computation time may be a lower bound of the time complexity per an output, even if the other vertices have quite few computation time on them. The second case (2) is that the structure of the enumeration tree is biased. Suppose that at least one child of an internal vertex is a leaf in the enumeration tree. Under the condition, all the internal vertices compose a path. The enumeration tree is also like a path.

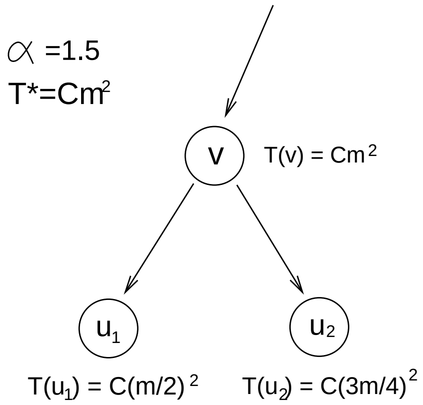

Figure 5.1: An vertex v satisfying the condition of Lemma 18 : The sum of computation time on both children is $C(m/2)^2+C(3m/4)^4 = C(13m/16) \leq 1.5Cm^2$ where m is the input size of v.

Suppose that the computation time on each internal vertex x is polynomial in the level of the vertex, i.e. it is given by the $d(x)^k$ where $d(x)$ is the distance from x to the farthest descendant, and k is a constant number. In the case, even if all the leaves have only O(1) computation time, the time complexity is O$(d(r)^k)$ per an iteration where r is the root vertex of the enumeration tree.

In these bad cases, the approach in the previous chapter will fail. The approach in this chapter avoids these bad cases by adding some modifications to the enumeration algorithms. We call the approach "trimming and balancing". The name comes from that the approach has a trimming phase and a balancing phase in each iteration.

The trimming phase avoids the bad case (1). We remove unnecessary parts from the inputs in the trimming phase. For example, let us consider the enumeration problem of all spanning trees in an undirected graph. If the graph contains some self loops, they are not included in any spanning tree. Thus they can be removed. If the graph contains some bridges, which are edges whose removals result disconnected, they are included in all spanning trees. Hence they can be contracted. The obtained graph is reduced. Therefore we can save unnecessary computation time for these edges. The obtained graph contains many spanning trees for its size. Hence if all the children of a vertex are leaves, then the computation time of the vertex may be small. Therefore we can avoid the bad case (1).

The balancing phase avoids the bad case (2). We generate subproblems including not so few outputting objects in the balancing phase. For example, we choose a partitioning element satisfying that both generated subproblems by the element have not so small sizes. Both subproblems may have not so few descendants in the enumeration tree, hence the enumeration tree will be balanced. Therefore we can avoid the bad case (2).

A trimming and balancing algorithm is constructed by the above way. We show the method for analyzing the time complexities of the algorithm. To analyze the time complexity, we consider the following distribution rule. Let $\mathcal{T}$ be an enumeration tree, and $D(x)$ be the number of descendants of a vertex x of $\mathcal{T}$. We recall that $T(x)$ is an upper bound of the time complexity of an iteration corresponding to x. Suppose that $\hat{T}$ is an upper bound of $\max_{x\in\mathcal{T}}\{T(x)/D(x)\}$. Our analysis also uses the parameter T^*, and a new parameter $\alpha > 1$. T^* must be larger than $(\alpha+1)\hat{T}$. The distribution is also done from a vertex to its children in the top-down manner. For a vertex x of the enumeration tree, let $T_p(x)$ be the computation time distributed by the parent of x to x. We distribute $T_p(x)+T(x)-T^*$ of the computation time to its children such that each child receives the computation time proportional to the number of its descendants. We distribute the computation time of each child recursively.

By this distribution rule, some vertices may receive much computation time. We define *excess vertices*, and stop the distribution on them. A vertex x is called excess if $T_p(x)+T(x) > \alpha\hat{T}D(x)$. The children of an excess vertex receive no computation time from their parent. The distribution rule is also applied to the descendants of excess vertices. By this new rule, $T_p(x)$ is bounded by $\alpha\hat{T}D(x)$ for any vertex x, since the computation time distributed from a parent to its child is proportional to the number of descendants of the child.

After the distribution, no vertex except excess vertices has more than $O(T^*)$ on it. Next, we distribute the computation time on each excess vertex x to all its descendants uniformly. Since the excess time $T_p(x)+T(x)-T^*$ is bounded by $(\alpha+1)\hat{T}D(x)$, each descendant receives at most $(\alpha+1)\hat{T}$ time from an excess ancestor. Let X^* be an upper bound of the maximum number of the excess vertices on a path from the root to a leaf. By the above analysis, we obtain an upper bound $O(T^* + \hat{T}X^*)$ of the time complexity per an iteration. From these facts, we obtain the following theorem.

Theorem 9 *An enumeration algorithm terminates in $O(T^* + \hat{T}X^*)$ time per an iteration.*

∎

Our analysis requrires $\hat{T}$ and X^*. To obtain a good upper bound of the time complexity, we have to set X^* and $\hat{T}$ to sufficiently good values. As a candidate of $\hat{T}$, we can utilize $\max_{x\in\mathcal{T}} T(x)/\bar{D}(x)$ where $\bar{D}(x)$ is a lower bound of $D(x)$. In the enumeration tree, it is hard to identify excess vertices, although we can obtain an efficient upper bound X^*. Let x and y be excess vertices such that y is an ancestor of x and no other excess vertex is in the path P_{yx} from y to x in the enumeration tree. Note that P_{yx} has at least one internal vertex.

Lemma 18 *At least one vertex w of $P_{yx}\setminus y$ satisfies the condition that $T(w)$ is larger than the sum of $\frac{\alpha}{\alpha+1}T(u)$ over all children u of w.*

Proof: If $P_{yx}\setminus y$ includes no such vertex, then all vertices w of $P_{yx}\setminus y$ satisfy the condition that $T_p(w) \leq \alpha\hat{T}D(w)$. It contradicts to the assumption of the statement. We prove it by induction. Any child of y satisfies the condition since y is an excess vertex. Suppose that a vertex w of $P_{yx}\setminus y$ holds $T_p(w') \leq \alpha T(w')$, where w' is the parent of w. Then $T_p(w) \leq (\alpha+1)T(w')D(w)/D(w')$. From the assumption, $T(w')$ is not greater than the sum of $\frac{\alpha}{\alpha+1}\hat{T}D(u)$ over all children u of w'. Since the sum of $D(u)$ is not greater than $D(w')$, we have $T(w') \leq \frac{\alpha}{\alpha+1}\hat{T}D(w')$. Therefore we have $T_p(w) \leq \frac{\alpha(\alpha+1)}{\alpha+1}\hat{T}D(w')D(w)/D(w') = \alpha\hat{T}D(w)$. ∎

From this lemma, we can obtain X^* by estimating an upper bound of the number of vertices satisfying this condition in any path from the root to a leaf in the enumeration tree. Similarly, we can obtain the following corollary.

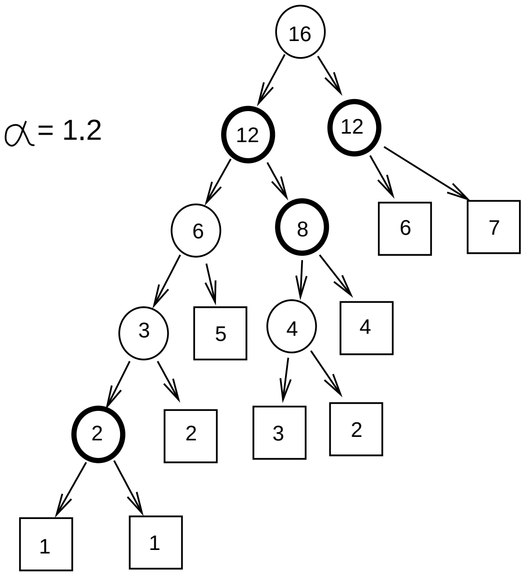

Figure 5.2: The enumeration tree and computation time on each vertex: The vertices satisfying the condition of Lemma 18 are drawn by emphasized circles, and all leaves are drawn by rectangles. In this tree, we can set $\hat{T}$ to 7, and X^* to 2.

Corollary 1 *If* $\hat{T} = \max_{x \in \mathcal{T}}\{T(x)/\bar{D}(x)\}$, *we have that at least one vertex* w *of* $P_{yx} \setminus y$ *satisfies that* $\bar{D}(w)$ *is larger than the sum of* $\frac{\alpha}{\alpha+1}\bar{D}(u)$ *over all children* u *of* w.

Proof: If $P_{yx} \setminus y$ includes no such vertex, then all vertices w of $P_{yx} \setminus y$ satisfy that $T_p(w) \leq \alpha\hat{T}D(w)$. We also prove it by induction. Any child of y satisfies the condition since y is an excess vertex. For any vertex w in $P_{yx} \setminus y$ and its parent w', if $T_p(w') \leq \alpha T(w')$ holds, then $T_p(w) \leq (\alpha+1)T(w')D(w)/D(w')$. Since (1) $T(w')$ is not greater than the sum of $\frac{\alpha}{\alpha+1}\hat{T}\bar{D}(u)$ over all children u of w', and (2) the sum of $\bar{D}(u)$ is not greater than $D(w')$, we have $T(w') \leq \frac{\alpha}{\alpha+1}\hat{T}D(w')$. Therefore we have $T_p(w) \leq \frac{\alpha(\alpha+1)}{\alpha+1}\hat{T}D(w')D(w)/D(w') = \alpha\hat{T}D(w)$. ▮

These conditions can be easily checked, and are often sufficient to obtain a good upper bound of the time complexity.

In this chapter, we have some enumeration algorithms obtained by the trimming and balancing approach. The first one is for directed spanning trees. The algorithm is based on binary partition. Its time complexity of an iteration is $O((m+n)\log n)$ where m and n are the number of edges and vertices, respectively. To analyze, we set T^* to $O(1)$ and $\hat{T}$ to $O(\log m)$. For an iteration x inputting m' edges, we also have $\bar{D}(x) = m'/2$ and $T(x) = O(m' \log m)$. X^* is bounded by $\log m$, and we have that the time complexity is $O(\log^2 m)$ per a directed spanning tree. The current fastest algorithm takes $O(n^{1/2})$ time per a directed spanning tree [53].

The second one is described in Section 5.2. The algorithm is for enumerating all bases of matroids. It is based on binary partition method. We have an upper bound of X^* is $O(1)$, and $\bar{D}(x) = (m-n)n$ for an iteration x where m and n are the size of the grand set and the rank of the matroid. Under the condition that $T(x) = (m-n)nT$ for some T, the time complexity of the algorithm is $O(T)$ per an output.

A trimming and balancing algorithm for perfect matchings is described in Section 5.3. In the existing studies, we are proposed an enumeration algorithm running in $\mathrm{O}(m+n)$ time per a perfect matching where m and n are the number of edges and vertices in the input graph [18]. To analyze our algorithm, we set $T^* = \mathrm{O}(n)$ and $\hat{T} = O(1)$. For an iteration x with m' edges, we also have $\bar{D}(x) = m' - n$ and $T(x) = \mathrm{O}(m'+n)$. X^* is bounded by n, and we reduce the time complexity from $\mathrm{O}(m+n)$ to $\mathrm{O}(n)$.

Section 5.4 describes some algorithms for the problems of enumerating some kinds of covering matchings. The problems are to enumerate all the covering matchings, maximum cardinality covering matchings, minimum weight matchings, and minimum weight maximum cardinality matchings. They utilize the algorithm for prefect matchings described in Section 5.3. All the time complexities of these algorithms are $\mathrm{O}(n)$ per a matching.

In Section 5.5, we propose an algorithm for maximal matchings in general graphs. In the existing studies, there is an enumeration algorithm running in $\mathrm{O}(nm^2)$ time per a maximal matching where n and m are the numbers of vertices and edges in the input graph [52]. Our algorithm is based on reverse search. Since the enumeration tree of the algorithm often has many vertices with only one child, we cannot apply the above analysis to the algorithm directly. However, by using the idea of balancing phase and data structures, we can reduce the time complexity from $\mathrm{O}(nm^2)$ to $\mathrm{O}(n \log\log n)$ per a maximal matching.

We describe the details of these algorithms in the rest of this chapter.

5.1 Enumerating Directed Spanning Trees

In this section, we consider an enumeration algorithm for directed spanning trees. For a given directed graph $G = (V, A)$, a directed spanning tree of G is a spanning tree such that no pair of its arcs share their heads. Since a spanning tree has $|V| - 1$ arcs, there is a vertex r which is a head of no arc. We call the vertex the *root* of the tree. The problem is to enumerate all directed spanning trees with the root vertex r. Here we simply call directed spanning trees with the root vertex r directed spanning trees. Suppose that we are given a directed graph including at least one directed spanning tree.

On the enumeration problem of directed spanning trees, a simple binary partition algorithm works efficiently. In 1978, H. N. Gabow and E. W. Myers [23] proposed an algorithm which runs in $O(|A| + |V| + |A|N)$ time and $O(|A| + |V|)$ space. Here N denotes the number of directed spanning trees in G. The algorithm chooses a partitioning arc a^* satisfying that there are both directed spanning trees of G including a^* and those not including a^*. The algorithm divides the problem into two subproblems which are to enumerate all directed spanning trees including a^* and those not including a^*. From the choosing rule of a^*, both subproblems are not empty. These two subproblems can be solved simply by two recursive calls with two subgraphs of G. One of the subgraphs is obtained by removing all arcs from G sharing their heads with a^*, and the other is obtained by removing a^* from G. Any directed spanning tree in the former graph is a directed spanning of G including a^* and vice versa, and any directed spanning tree in the latter graph is a directed spanning of G not including a^* and vice versa. These subgraphs can be easily constructed in $O(|V|)$ time. Since we can find the partitioning arc a^* in $O(|A|)$ time, the running time of this simple algorithm is $O(|A|)$ per a directed spanning tree.

In 1992, H. N. Kapoor and H. Ramesh [32] reduced the time complexity to $O(|A| + |V| + |V|N)$. They utilize some data structures which give us a partitioning arc in $O(|V|)$ time. Their improvement does not lose the optimality of the memory complexity. Since a directed spanning tree requires $O(|V|)$ outputting time, their algorithm is optimal in the sense of both time and space complexities if we have to output all directed spanning trees explicitly. If explicit outputting is not required, we can enumerate them in shorter time. For general enumeration algorithms, an outputting method called *compact output method* has been studied [32, 49]. It can shorten the output size of an object by outputting the local changes from the previous outputted objects. Also for directed spanning trees, an algorithm with a compact output method is proposed in [53]. The algorithm terminates in $O(|A| + |V| + |V|^{1/2}N)$ time, and reduces the output size from $O(|V|N)$ to $O(N)$. The details of the algorithm are described in Section 3.2. In our algorithm, we also use a kind of compact outputting method. When we generate a subproblem, we output arcs included in all directed spanning trees of the subproblem. By this, the size of output does not exceed the time complexity of the algorithm.

Our trimming and balancing algorithm for the problem is based on binary partition. For the simple binary partition algorithm, we add a trimming phase and a balancing phase. In the trimming phase, we remove all the unnecessary arcs. Unnecessary arcs are characterized by arcs included in no directed spanning tree or included in all directed spanning trees. We also remove multiple arcs and self loops. By the trimming algorithm, for any arc a, there are some directed spanning trees including a, and those not including a. Hence we can choose any arc to partition the problem. This partitioning arc is chosen in the balancing phase. The arc satisfies that the input sizes of both generated subproblems are not smaller than the one-fourth of the original problem. By these modifications, the number of excess vertices on any path is bounded by $X^* = \log n$. The time complexity of the algorithm is reduced to $O(\log^2 |V|)$ time per a directed spanning tree. In the following subsections, we show the details of the trimming phase and the balancing phase, and the analysis of the time complexity.

5.1.1 A Trimming Algorithm

In this subsection, we describe our trimming algorithm. The trimming algorithm requires some good necessary and sufficient conditions for unnecessary arcs. Firstly, we see a trimming method for self loops and multiple arcs. Any self loop is included in no directed spanning tree, hence we remove all self loops from the input graph. If the graph contains multiple arcs, any directed spanning tree includes at most one of them. Moreover, if a directed spanning tree includes an arc a of them, there are directed spanning trees obtained by replacing a by the other arcs of them. Therefore these arcs except a can be deleted. By putting marks of deleted arcs to a, we can obtain directed spanning trees including the deleted arcs when a directed spanning tree including a is outputted.

Next we see the trimming method for arcs which all directed spanning trees include or do not include them. All such arcs can be contracted or removed from the graph. Our trimming algorithm finds all such arcs, and contract or remove them from the graph. To find such arcs, we utilize the following properties.

Property 3 *For an arc (u, v), there is a directed spanning tree including (u, v) if and only if there is a simple directed path P from r to u not including v.*

Proof : The "only if" part of the statement is obviously. Suppose that there is a directed path P from r to u not including v. Let T be a directed spanning tree. We consider the case that T does not include (u, v). By adding P to T and deleting the arcs of $T \setminus P$ sharing their heads with arcs of P, we can obtain a directed spanning tree including (u, v). ▮

From this property, we can see that arcs which have their heads on r are included in no directed spanning tree.

Property 4 *For an arc (u, v), there is a directed spanning tree not including (u, v) if and only if there is a simple directed path P from r to v not including (u, v).*

Proof : The "only if" part of the statement is obviously. Suppose that there is a directed path P from r to v not including (u, v). Let T be a directed spanning tree. We consider the case that T includes (u, v). By adding P to T and deleting the arcs of $T \setminus P$ sharing

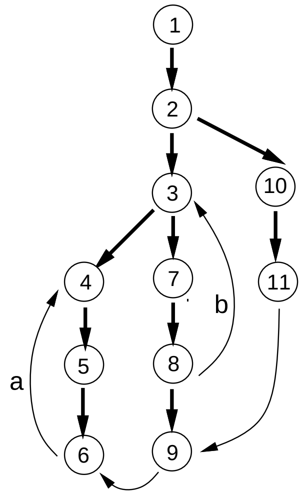

Figure 5.3: The back arc a is included in a directed spanning tree since $i(h(6,6)) = 2$. The back arc b is included in no directed spanning tree since $i(h(7,7)) = 3$ and $i(h(8,8)) = 7$.

their heads with some arcs of the path, we can obtain a directed spanning tree not including (u,v). ▮

From this property, we can see that for an arc whose tail is r, there is some directed spanning trees not including the arc only if there is another arc whose tail is r.

These conditions can be checked in $O(|A|)$ time for an arc by some graph search algorithms. Thus we can obtain a simple trimming algorithm running in $O(|A|^2)$ time. Since the time complexity is not small enough, we can not obtain a fast enumeration algorithm by this trimming algorithm. Hence we consider the other fast trimming algorithm. From Property 4, if all arcs not satisfying the former condition are removed from the graph, then an arc is not included in a directed spanning tree if and only if it shares its head with the other arc. Hence the latter condition can be checked in $O(|A|)$ time for all arcs. In the following, we show the other way for checking the former condition for all arcs in short time.

Let T be a depth-first search tree of G with the root r. For a non-back arc a of T, the directed path from r to the tail of a in T does not include the head of a. Thus the former condition holds for any non-back arc. To check the former condition for back arcs, we introduce some notations, and state some lemmas. Let us put indices to all vertices in the order of visiting of the depth-first search. The index of a vertex v is denoted by $i(v)$. We denote the unique path in T from a vertex u to its descendant v by P_{uv}. For an index i, vertices v and u, we call a directed path from u to v an *i-bypass* if all the internal vertices of the path have indices larger than or equal to i. For a vertex v and an index $i \leq i(v)$, let $h(v,i)$ be the minimum index vertex among vertices satisfying that there are some i-bypasses from the vertices to v. Since any directed path to v from a vertex with an index smaller than v includes a common ancestor of them, $h(v,i)$ is an ancestor of v. Note that an ancestor has an index smaller than any its descendant. By using these notations, we state the following lemmas (see Figure 5.3).

Lemma 19 *For a back arc (u,v), there is a directed path from r to u not including v if and* *only if $i(h(w, i(w))) < i(v)$ holds for a vertex w in $P_{vu} \setminus v$.*

Proof: If $i(h(w, i(w))) < i(v)$ holds for a vertex $w \in P_{vu} \setminus v$, there is a directed path from r to v via $i(w)$-bypass from $h(w, i(w))$ to u. Thus the "if" part of the lemma is obviously. Suppose that there is a simple directed path P from r to u not including v. Then P includes some vertices of $P_{vu} \setminus v$. Let w be the first vertex of them to appear in the path. The subpath of P from r to w includes some ancestors of v. Let w' be the last vertex of them to appear in the subpath. Note that the subpath from w' to w of P includes no vertex of $P_{w'w}$ in its internal vertices. Moreover, the subpath includes no vertex u with $i(u) < i(v)$ because any path from u to v includes some common ancestors of u and v. It contradicts the choosing rule of w'. Hence all internal vertices of the subpath have indices larger than $i(v)$. We have $i(h(w, i(w))) < i(v)$. ▮

From this lemma, we can identify unnecessary arcs by using h. Our algorithm firstly obtains $h(v, i(v))$ for the vertex v with $i(v) = n$. In each iteration, we decrease i one by one, and obtain all $h(v, i)$ for all v with $i(v) \geq i$. All $h(v, i)$ are obtained by all $h(v, i+1)$. The updating method of h is based on the following properties.

Lemma 20 *Suppose that a vertex u has the index $i-1$.*
(1) $h(u, i-1)$ is the minimum index vertex among all v and $h(v, i)$ satisfying that there are arcs (v, u).
(2) If $h(v, i) \neq h(v, i-1)$ holds for a vertex v, then v is a descendant of u and holds $h(v, i-1) = h(u, i-1)$.

Proof: (1) Let P be an $i-1$-bypass from $h(u, i-1)$ to u. If P has no internal vertex, there is an arc from $h(u, i-1)$ to u. Otherwise, the vertex v next to u satisfies $h(v, i) = h(u, i-1)$. Thus the condition holds. (2) If $h(v, i-1) \neq h(v, i)$, then u is included in any $i-1$-bypass from $h(v, i-1)$ to v. Any $i-1$-bypass from $h(u, i-1)$ to u does not include v, since if v is included in the bypass, then the subpath of the bypass from $h(u, i-1)$ to v is an i-bypass from $h(u, i-1)$ to v. Hence we have that $h(v, i-1) = h(u, i-1)$. Since (1) a directed path from u to a vertex with an index larger than $i-1$ includes at least one common ancestor of them, and (2) any ancestor of u has an index smaller than $i-1$, v is a descendant of u. Therefore $h(v, i-1) \neq h(v, i)$ holds only for descendants of u. ▮

From the lemma, we can see that $i(h(v, j)) \leq i(h(v, i))$ for any $j < i$. We can also see the following lemma.

Lemma 21 *Let u be a vertex satisfying that $i(u) > i$, and v be a descendant of u. If we have $h(u, i) = h(v, i)$, then $h(u, j) = h(v, j)$ holds for any $j < i$.*

Proof: Since u is an ancestor of v, we have $i(h(u, j)) \geq i(h(v, j))$ for any j. Suppose that there is an index j satisfying $i(h(u, j)) > i(h(v, j))$. We assume that j is the maximum index among indices satisfying the condition. From this assumption, there is a j-bypass from $h(v, j)$ to v not including u. Note that the bypass includes the vertex v' whose index is j. Since T is a depth-first search tree, any directed path from v' to v includes some common ancestors of v and v'. These ancestors are also ancestors of u, thus we can obtain a j-bypass from $h(v,j)$ to u by merging the j-bypass from $h(v,j)$ to v and $P_{v'u}$. It contradicts the assumption. ∎

From the lemma, if we have $h(u,i) = h(v,i)$ for a vertex u and its child v, the equation holds for all $j < i$. Hence, in the graph obtained by contracting u and v, h is preserved for any vertex and any index smaller than i. Thus we contract them if such a vertex and a child exist. For a vertex u, if a descendant v of u satisfies that $i(h(v,i(u)+1)) > i(h(v,i(u)))$, the child v' of u included in the directed path from u to v satisfies the condition $i(h(v',i(u)+1)) > i(h(u,i(u)))$. Hence v' and u satisfy that $i(h(v',i(u))) = i(h(u,i(u)))$. Therefore, in each iteration with the index i, we find the child v' of the vertex u with $i(u) = i$ which maximizes $i(h(v',i))$ among all children of u. If v' satisfies $i(h(v',i)) < i(h(u,i))$, all descendants v of u satisfies $i(h(v,i)) < i(h(u,i))$. Otherwise, we have that $h(v',i) = h(u,i)$, hence we contract u and v'. We do this operation until there is no child v satisfying $i(h(v,i)) = i(h(u,i))$. After contracting all these vertices, no descendant v of u satisfies that $h(u,i) \leq h(v,i+1)$. Therefore no vertex v satisfies $h(v,i) \neq h(v,i+1)$ in the contracted graph. Hence we can update all h by this contracting operation.

The details of the trimming algorithm is as follows.

```
ALGORITHM: TRIM_DIRECTED_SPANNING_TREES($G$ )
Step 1: Find a depth-first search tree $T$ in $G$.
Step 2: Put indices to all vertices of $T$ in the order of visiting. Set $i$ to $|V|$.
Step 3: Compute $h(u,i)$ for the vertex $u$ with $i(u) = i$.
Step 4: Find a vertex $v$ maximizes $i(h(v,i+1))$ among all children of $u$.
Step 5: If $i(h(v,i+1)) > i(h(u,i))$, then contract $u$ and $v$. Go to Step 4.
Step 6: Set $i$ to $i-1$. If $i \geq 1$, go to Step 3.
Step 7: Recover the graph $G$.
Step 8: For each back arc $(u,v)$, find the vertex $w$ of $P_{vu} \setminus v$ minimizing $i(h(w,i(w)))$.
        If $i(h(w,i(w))) \geq i(v)$, then remove the arc from $G$.
Step 9: Remove all self loops.
Step 10: Contract arcs which share their heads with no other arc.
Step 11: Remove all multiple arcs except one arc of them.
```

The time complexity of the algorithm is as follows. The total time spent for Step 1, 2, 6, 7, 9, 10 and 11 is $O(|A|)$. Step 3 takes the time proportional to the number of arcs whose heads are u. Thus the total time spent for Step 3 through the algorithm is $O(|A|)$. Step 4 may take time proportional to the number of children of u. To reduce the computation time, we have a heap on each vertex v composed of all children of v. The heap gives us the child w of v maximizing $i(h(w,i))$ in $O(\log |V|)$ time. The contraction of two vertices u and v in Step 5 requires the merge of two heaps and the merge of two edge adjacency lists on u and v. By utilizing some binary trees to store the heaps and edge adjacency lists, the contraction takes only $O(\log |V|)$ time. By this, the time complexity of an operation for an edge increases from $O(1)$ to $O(\log |V|)$. Hence Step 1, 2, 6, 7, 9, 10 and 11 now take $O(|A| \log |V|)$ time. Since the contractions of vertices occur at most $|V|$ times through all the iterations, the total required time for Step 4 and 5 is $O(|V| \log |V|)$. In Step 8, we have to find the vertex w minimizing $i(h(w, i(w)))$ in a path from the head to tail for each back arc. It seems to take $O(|V||A|)$ time, although the dynamic tree algorithm [47] reduces the computation time. The algorithm spends $O(|V|)$ preprocessing time and finds the vertex v minimizing $i(h(w, i(w)))$ among all vertices of any path of T in $O(\log |V|)$ time. Therefore, the total time complexity of the trimming algorithm is $O(|A| \log |V|)$.

5.1.2 A Balancing Algorithm

In this subsection, we consider an algorithm for the balancing phase. For a given trimmed directed graph G and the root r, the algorithm finds an arc a^* satisfying that both generated subproblems input graphs including at least $|A|/4$ arcs such that each of them is included in a directed spanning tree and not included in the other directed spanning tree.

For an arc a, let G' be the graph obtained by deleting all arcs sharing their heads with a except for a. Under the condition that G is a trimmed graph, the following lemma holds. Suppose that a directed path from a vertex to an arc is a simple directed path from the vertex to the head of the arc which includes the arc.

Lemma 22 *Let P be a directed path from r to a. If an arc e is included in all directed spanning trees or no directed spanning tree of G', then the head of e is on P.*

Proof : We state the lemma by proving its contraposition. For an arc e whose head is not on P, there exists a directed path Q in G from r to e, because of Property 3. If Q includes no arc of $G \setminus G'$, then it is also included in G'. Thus e is included in some directed spanning trees of G'. Otherwise, we consider the subgraph given by $Q \cup P$. The subgraph includes a directed path of G' from r to e. Since the head of e is not on P, the path is always simple. Thus, only arcs which have their heads on P may be included in no directed spanning tree of G'.

Since G is trimmed, an arc e whose head is not on P shares its head with some other arcs. The arcs are also included in some directed spanning trees of G'. Hence they are also not included in a directed spanning tree of G'. Therefore any arc included in all directed spanning trees of G' or no directed spanning tree of G' has its head on P. ▮

Let a' be an arc sharing its head with a, and P' be a directed path from r to a'.

Lemma 23 *If an arc e is included in all directed spanning trees or no directed spanning tree of $G \setminus a$, then its head is on P'. In the case that at least two arcs share their heads with a, for any arc e of the arcs, there are both directed spanning trees including e and not including e.*

Proof : $G \setminus a$ includes the graph obtained by deleting all arcs sharing their heads with a' except for a'. Thus, for any arc whose head is not on P', there are both directed spanning trees including the arc and those not including the arc in $G \setminus a$ from Lemma 22. Therefore the first assertion is proved.

For any arc of $G \setminus a$ sharing its head with a', a directed path from r to it does not include a. Hence, from Property 3, the arc is included in some directed spanning trees of G'. Except for

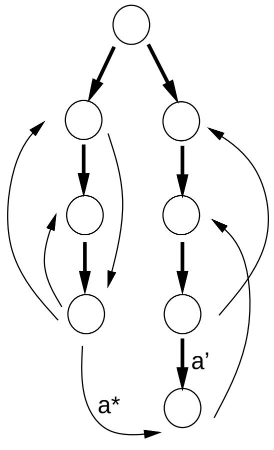

Figure 5.4: An instance of the arc a^* and a'.

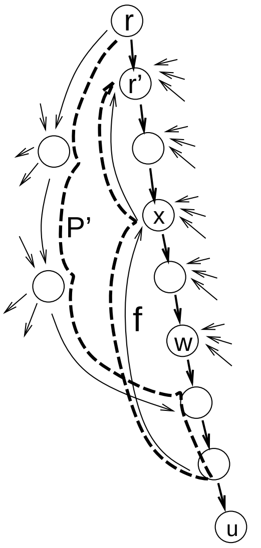

Figure 5.5: Vertices r, r', u, x and w. The paths are P_{ru} and P'.

the case that only two arcs a and a' have their heads on v, there are some directed spanning trees not including the arc in $G \setminus a$. ▮

By using these lemmas, we select a partitioning arc as follows. Let the weight of a directed path be the number of arcs whose heads are on the path. The weight of a directed path P is denoted by $w(P)$. In the balancing phase, we will find an arc a^* satisfying the conditions that (a) the weight of a directed path from r to a^* is at most $3|A|/4$, and (b) for an arc a' sharing its head with a^*, the weight of a directed path from r to the tail of a' is at most $|A|/2$. If an arc a^* satisfies these conditions, $G \setminus a^*$ includes at most $|A|/2 + 2$ arcs which are included in all directed spanning trees or no directed spanning tree (see Figure 5.4). Under the condition that $|A| \geq 8$, $|A|/2 + 2 \leq 3|A|/4$. The graph obtained by removing all arcs sharing whose heads with a^* also includes at most $3|A|/4$ such arcs. We consider the method for finding such an arc a^*.

Let T be a directed spanning tree of G. To select an arc, we consider the following three cases. (1) If there is a non-back arc (u, v) of T such that $w(P_{ru}) \leq |A|/2$ and $w(P_{rv}) \leq |A|/2$, then the arc of T whose head is v satisfies the above conditions. (2) In the other case, let u be the vertex maximizing $w(P_{ru})$ among all vertices. We denote the vertex next to r in P_{ru} by r'. Let P' be a simple directed path from r to r' not including the arc (r, r'). P' always exists since G is a trimmed graph. Let w be the vertex minimizing $w(P_{rw})$ among all vertices v of P_{ru} satisfying that $w(P_{rv}) > |A|/2$. We suppose that x denotes the first vertex to appear in P' among vertices v of P_{ru} with $w(P_{rv}) \leq |A|/2$. Since any vertex except r is the head of at least two arcs, at least $2|V| - 4$ arcs of G have their heads not on w. On the other hand, at most $|V| - 1$ arcs have their heads on w, thus the number of arcs whose heads are w does not exceed $|A|/2$. Therefore the weight of r' is at most $|A|/2$, and x always exists in P'. Let f be the arc of P' whose head is x. We show an example of these vertices and arcs in Figure 5.5. If the subpath P'' of P' from r to f satisfies $w(P'') \leq |A|/2$, then the arc of T sharing its head with f satisfies the conditions (a) and (b).

(3) If f does not satisfies these conditions, $w(P'') > |A|/2$. From the definition of x, the vertices included in both P'' and P_{rw} are at most r and w. Hence P'' includes w since more than $|A|/2$ arcs have their heads on P_{rw}. Suppose that P_1 denotes the path with the smaller weight among P_{uw} and the subpath of P'' from r to w. We also denote the other path by P_2. Let d denote the number of arcs whose heads are w. Since $P_1 \cap P_2 = \{r, w\}$, we have $w(P_1) \leq (|A| - d)/2 + d = |A|/2 + d/2 \leq 3|A|/4$, and $w(P_2 \setminus w) \leq |A|/2$.

From the above observations, there is an arc satisfying the above two conditions for any trimmed directed graph. To find such an arc, what we have to do is only to find a directed spanning tree and some directed paths. They can be done in $O(|A| + |V|)$ time. We now describe the details of the balancing algorithm.

```
ALGORITHM: BALANCE_DIRECTED_SPANNING_TREES($G$ )
Step 1: Find a directed spanning tree $T$ of $G$.
Step 2: Compute $w(P_{rv})$ for all vertices $v$.
Step 3: Find a non-back arc $a$ whose both endpoints have weights at most $|A|/2$. If $a$
        exists, output the arc of $T$ sharing its head with $a$. Stop.
Step 4: Find a vertex $u$ maximizing $w(P_{ru})$.
Step 5: Find the vertex $w$, a directed path $P'$, and the arc $f$.
Step 6: If the weights of two paths from $r$ to endpoints of $f$ is less than or
        equal to $|A|/2$, then output $f$ and stop.
Step 7: Compute the weights of the paths $P_1$ and $P_2$.
Step 8: Output the end arc of the path with the smaller weight among $P_1$ and $P_2$.
```

By utilizing the trimming algorithm and the balancing algorithm, we obtain the following algorithm.

```
ALGORITHM: T&B_FOR_DIRECTED_SPANNING_TREES($G$ )
Step 1: Find a partitioning arc $a^*$ by BALANCE_DIRECTED_SPANNING_TREES.
Step 2: Remove all arcs sharing their heads with $a^*$,
        and trim the graph by TRIMMING_DIRECTED_SPANNING_TREES.
```

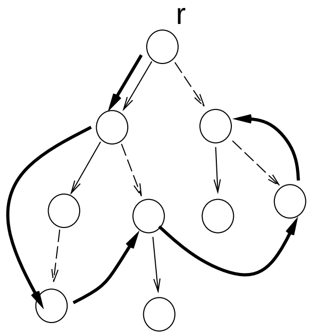

Figure 5.6: Generating the other directed spanning tree by adding a path to a directed spanning tree. Arcs drawn by dotted lines are eliminated.

```
         Call T&B_for_Directed_Spanning_Trees recursively.
Step 3: Remove $a^*$ from $G$, and trim the graph by
         Trimming_Directed_Spanning_Trees.
         Call T&B_for_Directed_Spanning_Trees recursively.
```

The algorithm takes $O(|A| \log |V|)$ time for an iteration in the worst case. In the following subsection, we bound the time complexity more tightly by making a distributing rule of computation time.

5.1.3 Bounding the Total Time Complexity by Analyzing the Enumeration Tree

In this subsection, we estimate an amortized time complexity of our algorithm. We firstly consider the enumeration tree of the algorithm. Each vertex of the tree corresponds to each recursive call, and each edge connects two vertices if the recursive call corresponding to one of the vertices occurs in the other. There is a one-to-one correspondence between all leaves and all directed spanning trees. Any internal vertex has exactly two children. G_x denotes the input graph of the recursive call corresponding to a vertex x.

We estimate a lower bound of the number of descendants of a vertex of the enumeration tree by the following lemma.

Lemma 24 *If any arc of G is included in a directed spanning tree and not included in the other directed spanning tree, G includes at least $|A|/2$ distinct directed spanning trees.*

Proof : Let T be a directed spanning tree of G. From Property 3, for any non-tree arc a of T, there is a simple directed path P from r to a. We can construct the other directed spanning tree T' by adding P to T and deleting arcs of $T \setminus P$ which share their heads with arcs of P (see Figure 5.6). All arcs of $T' \setminus T$ are included in P. If a directed spanning tree T' is generated by two distinct non-tree arcs a and a', then we have a directed path from r to a including a' and a directed path from r to a' including a in T'. This implies that T includes a directed cycle. Hence, for any pair of arcs, we obtain distinct directed spanning trees. Since there are at least $|A|/2$ non-tree arcs in G, G includes at least $|A|/2$ directed spanning trees. ∎

From the lemma, we obtain a lower bound $\bar{D}(x) = |E(G_x)|/2$ of $D(x)$. Here $D(x)$ is the number of descendants of a vertex x of the enumeration tree, and $E(G_x)$ denotes the arc set of G_x. The computation time on a vertex x is $\mathrm{O}(|E(G_x)| \log |V|)$ time, hence we set $\hat{T} = \mathrm{O}(\log |V|)$, and $T^* = \mathrm{O}(\log |V|)$.

By the balancing algorithm, we can obtain a good upper bound of excess vertices. From Corollary 1, we can bound the number of excess vertices by the number of vertices satisfying that $\bar{D}(x) > \frac{\alpha}{\alpha+1}(\bar{D}(x_1)+\bar{D}(x_2))$ where x_1 and x_2 are the children of x. If the condition holds, we have that $|E(G_x)|/2 > \frac{\alpha}{\alpha+1}(|E(G_{x_1})|/2 + |E(G_{x_2})|/2)$, thus $|E(G_{x_i})| \leq \frac{\alpha+1}{\alpha}|E(G_x)| - |E(G_x)|/4$ for each child x_i of x. By setting $\alpha = 8$, we can estimate an upper bound of $|E(G_{x_i})|$ by $(7/8)|E(G_x)|$. Therefore any path from the root to a leaf has at most $\log_{8/7} |A|$ excess vertices in the enumeration tree. We obtain an upper bound $X^* = \log_{8/7} |A|$. From Theorem 9, an upper bound of the time complexity is given by $\mathrm{O}(T^* + \hat{T}X^*)$ per a vertex of the enumeration tree. It is $\mathrm{O}((\log |V| + \log |V|) \log |A|)$, hence the trimming and balancing algorithm spends $\mathrm{O}(\log^2 |V|)$ time per a directed spanning tree.

Theorem 10 *All directed spanning trees in a directed graph can be enumerated in $O(|A| \log |V| + |V| + N \log^2 |V|)$ time and $O(|A| + |V|)$ space by a trimming and balancing algorithm where N is the number of directed spanning trees.*

∎

5.2 Enumerating Bases of Matroids

In this section, we describe a trimming and balancing algorithm for enumerating all bases of a given matroid. The algorithm is based on the binary partition scheme. Since matroids have some good structures, our algorithm is quite simple. The analysis of the trimming and balancing works well for the algorithm.

Before explaining the algorithms, we show the definition and properties of matroids. To see the details of the matroid theory, refer [43]. A matroid $\mathcal{M}$ is composed of a correction E of elements and a subset family $\mathcal{I}$ of 2^E. The former set is called the *grand set* and the latter is called the *independent set family*. An element of the independent set family is called an *independent set*. If the independent set family satisfies some conditions, we say that $\mathcal{M}$ is a matroid. To define matroids, we have some kinds of axioms. For example, we show the independent set axiom as follows.
(I1) $\emptyset \in \mathcal{I}$.
(I2) If $I \in \mathcal{I}$ and $I' \subseteq I$, then $I' \in \mathcal{I}$.
(I3) If $I_1, I_2 \in \mathcal{I}$ satisfies $|I_1| < |I_2|$, then there is an element $e \in I_2 \setminus I_1$ such that $I_1 \cup e \in \mathcal{I}$.
A subset family $\mathcal{I}$ satisfying these three conditions is an independent set family. There are also some other axioms to define the matroids.

A subset of E is called a *dependent set* if it is not an independent set. A minimal dependent set is called a *circuit*. A maximal cardinality independent set is called a *base* of the matroid. It is known that all bases have the same cardinality. The cardinality of bases is called the *rank* of the matroid. For the given matroid, we denote the cardinality of the grand set by m, and its rank by n.

Bases of a matroid satisfy some good conditions. One of them is the following.

Property 5 *For two bases $B \neq B'$ and an element $e \in B \setminus B'$, there is an element $f \in B'$ satisfying that $B \setminus e \cup f$ is a base.*

∎

This property is very useful since if there are two bases in a matroid, then we can always obtain the other base from a base by exchanging only a pair of elements.

If a set obtained by adding an element to an independent set is dependent, then the set is known to include a unique circuit. Since any base B is maximal, there is a unique circuit in $B \cup e$ for any element e not in B. The circuit is called the *elementary circuit* of e on B, and denoted by $Cir(B, e)$. Since the elementary circuit is the unique circuit in $B \cup e$, we can obtain the other base by deleting an element of $Cir(B, e)$ from $B \cup e$. For an element $e \in B$, the *elementary cut* of e on B is a set of elements f such that $B \setminus e \cup f$ is a base. It is denoted by $Cut(B, e)$.

For an element e of a matroid $\mathcal{M} = (E, \mathcal{I})$, we consider a subset family $\mathcal{I}'$ which is the correction of all independent sets obtained by deleting e from the independent sets of $\mathcal{I}$ including e. It is known that $\mathcal{I}'$ is also an independent set family, and $(E \setminus e, \mathcal{I}')$ is a matroid [43]. We say that the matroid is obtained by *contracting* e, and denote it by $\mathcal{M}/e$. The correction of the independent sets of $\mathcal{I}$ not including e is also known to be an independent set family. We say that the matroid is obtained by *deleting* e, and denote it by $\mathcal{M} \setminus e$.

Generally, the size of the independent set family of a matroid is quite huge. Hence we often give a matroid by its grand set and some oracle algorithms. The oracle algorithms answer some questions. "A given subset of E is independent or not?" is such a question. In this section, we deal two major oracle algorithms. One of them answers whether a given set is independent or not, and the other outputs an elementary circuit for a given base and an element. The former is called an *independent set oracle algorithm*, and the latter is called an *elementary circuit oracle algorithm*. We denote the time complexity of these oracle algorithms by $T_{ind}(m,n)$ and $T_{cir}(m,n)$. Our algorithm can be constructed with either them.

We utilize some kinds of other matroid subroutines. These subroutines are utilized for contracting and deleting of an element. We denote the time complexities of them by $T_{cont}(m,n)$ and $T_{rmov}(m,n)$. We assume that the contraction and the deletion of k elements simultaneously also take only $T_{cont}(m,n)$ and $T_{rmov}(m,n)$ time.

By using these algorithms, we can obtain a simple binary partition algorithm. For a given matroid $\mathcal{M} = (E, \mathcal{I})$ and its base B, we find a partitioning element $e \in B$ and an element $f \notin B$ such that $B \setminus e \cup f$ forms a base. If there is another base in $\mathcal{M}$, such a pair of elements always exist. We spend $\mathrm{O}((m-n)T_{cir}(m,n))$ or $\mathrm{O}((m-n)nT_{ind}(m,n))$ time to find them. By using e, we divide the problem into two subproblems of enumerating all bases including e and all those not including e. All bases including e correspond to bases of $\mathcal{M}/e$. All bases not including e also correspond to bases of $\mathcal{M} \setminus e$. Hence they can be enumerated recursively by solving the subproblems with $\mathcal{M}/e$ and $\mathcal{M} \setminus e$. Thus we obtain an enumeration algorithm by using only algorithms for constructing these two matroids. Hence the simple algorithm runs in $\mathrm{O}((m-n)\min\{T_{cir}(m,n), nT_{ind}(m,n)\} + T_{cont}(m,n) + T_{rmov}(m,n))$ time. It is a very simple bound of the time complexity, hence we reduce it by adding a trimming algorithm and a balancing algorithm in the next subsection.

5.2.1 A Trimming and Balancing Algorithm

In this subsection, we describe a trimming algorithm and a balancing algorithm. Our trimming algorithm is quite simple. For a given matroid, if an element of the grand set forms a circuit, then it is included in no base. We call the element a *loop*. In the matroid obtained by removing all loops, the set of bases is preserved. If the elementary cut of an element is composed of only itself, it is included in all bases. These elements can be also contracted. The trimming algorithm contracts or deletes these unnecessary elements. The check of these conditions for all elements is done by finding the elementary circuits for all elements outside a base. It can be done in $\mathrm{O}((m-n)\min\{nT_{ind}(m,n), T_{cir}(m,n)\})$. For any element e, a given trimmed matroid includes bases including e and those not including e. Hence we take only $\mathrm{O}(\min\{nT_{ind}(m,n), T_{cir}(m,n)\})$ time to find a partitioning element and the other base for any trimmed matroid.

We next consider the balancing algorithm. By partitioning the given matroid, we obtain two matroid $\mathcal{M}/e$ and $\mathcal{M} \setminus e$. After trimming these matroids, if one of the matroid has a sufficiently small size, the enumeration tree of the algorithm may be biased. Hence we consider the case that one of the subproblems loses some elements by the trimming algorithm. To obtain the balancing algorithm, we show the following properties of $\mathcal{M}/e$ and $\mathcal{M} \setminus e$ for a base B of $\mathcal{M}$.

Property 6 *Suppose that $e \in B$ and B' is the base of $\mathcal{M}/e$ obtained by $B \setminus e$. The following conditions hold for $\mathcal{M}/e$ and B'.*
(1) For any element $f \notin B$, the difference between $Cir(B,f)$ and $Cir(B',f)$ is e or nothing.
(2) For any element $f \in B \setminus e$, $Cut(B,f) = Cut(B',f)$.

Proof : For an element f, if $Cir(B,f)$ does not include e, then $Cir(B,f) = Cir(B',f)$. In the other case, $Cir(B,f) \setminus e = Cir(B',f)$. Therefore, for any element f in $Cut(B,e')$, f is also included in $Cut(B',e')$. ▮

Property 7 *The following conditions hold for $e \notin B$.*
(1) For any element $f \notin B$, the elementary circuits of f on B in $\mathcal{M}$ and that in $\mathcal{M} \setminus e$ are equal.
(2) For any element $f \in B$, the difference between the elementary cuts $Cut(B,f)$ in $\mathcal{M}$ and that in $\mathcal{M} \setminus e$ is nothing or e.

Proof : Since $Cir(B,f)$ in $\mathcal{M}$ does not include e, it is also an elementary circuit of f on B in $\mathcal{M} \setminus e$. Hence $Cir(B,f)$ is preserved by deleting e from $\mathcal{M}$. For any element e' and $f \neq e$ in $Cut(B,e')$, $Cir(f,B)$ in $\mathcal{M} \setminus e$ includes e'. Therefore the difference between $Cut(B,f)$ in $\mathcal{M}$ and that in $\mathcal{M} \setminus e$ is nothing or e. ▮

By using these properties, we construct the balancing algorithm. Firstly we see the case that some elements of E are loops in $\mathcal{M}/e$. In the case, we partition the original problem in the other way. The partitioning method is based on the following properties.

Property 8 *For two elements e and e' of E such that $\{e,e'\}$ is a circuit, B is a base including e if and only if $B \setminus e \cup e'$ is a base.*

Proof : Since $\{e,e'\}$ is a circuit, there is no base including both e and e'. For any base including e and not including e', we can obtain the other base by adding e' and deleting an element from the base. Since $\{e,e'\}$ is included in the dependent set obtained by adding e to the base, the unique circuit in the dependent set is $\{e,e'\}$. Hence the deleted element is always e. Similarly, for any base B' including e', we can also see that $B' \setminus e' \cup e$ is a base. ▮

Since $\mathcal{M}$ is trimmed, any loop $\{e'\}$ of $\mathcal{M}/e$ composes a circuit of $\mathcal{M}$ with e. Thus we can see that the enumeration of all bases of $\mathcal{M}$ including a loop e' of $\mathcal{M}/e$ can be done by enumerating all bases including e. All bases including e' are constructed from the bases including e by only replacing e by e'.

Suppose that we consider the case that $e_2, ..., e_k$ are loops in $\mathcal{M}/e_1$. From the above lemmas, we divide the problem into the subproblems enumerating all bases including each e_i, and those not including any e_i. Since any matroid $\mathcal{M}/e_i$ can be constructed from $\mathcal{M}/e_1$ by renaming e_1 to e_i, the time complexity of an iteration does not increase.

Next we consider the case that some elements of $\mathcal{M} \setminus e$ have elementary cuts composed of only themselves. Similar to the above partitioning method, we can partition the problem in the other way in the case. The partitioning method is based on the following lemma.

Lemma 25 *For a trimmed matroid $\mathcal{M}$, let $e_1, ..., e_k$ are elements satisfying that any base not including e_1 always includes $e_2, ..., e_k$. All bases include at least $k-1$ elements of them.*

Proof: Suppose that there is a base not including e_i and e_j. From the assumption, e_i and e_j are not e_1. Since $\mathcal{M}$ is trimmed, by removing e_1 and adding an element to the base, we obtain a base not including e_1 and e_j, or not including e_1 and e_i. It contradicts the assumption. ■

Lemma 26 *Let $e_1, ..., e_k$ be elements satisfying that any base not including e_1 always includes $e_2, ..., e_k$. For any base not including e_1, there is a base obtained by exchanging e_1 and e_i. Conversely, for any i and any base including e_1, there is a base obtained by exchanging e_1 and e_i.*

Proof : To prove the lemma, we show that any circuit includes all e_i, or includes no e_i. Suppose that a circuit C includes e_i and does not include e_j. By removing e_i from C, we can obtain an independent set. Let B be a base including the independent set. From the above lemma, B includes e_j. Since the matroid is trimmed, there is a base not including e_j. Hence there is an element e' whose elementary circuit includes e_j. Since $Cir(B, e_i)$ is C, e' is not e_i. By exchanging e' and e_j, we obtain a base including neither e_i nor e_j. This contradicts the above lemma.

We now have that any circuit includes all e_i or no e_i. Hence for any base not including e_1, we have a base obtained by exchanging e_1 and e_i. For any i and any base including e_1, we also have a base obtained by exchanging e_1 and e_i. ■

Let us consider the case that $e_2, ..., e_k$ are included in any base of $\mathcal{M} \setminus e_1$. In this case, we partition the problem into subproblems of enumerating all bases not including each e_i, and those including all e_i. Since any matroid $\mathcal{M} \setminus e_i$ can be constructed from $\mathcal{M} \setminus e_i$ by renaming e_1 to e_i, each additional subproblem is constructed in O(1) time.

From the above properties, the balancing algorithm can be constructed by using the trimming algorithm. By using the trimming algorithm and the balancing algorithm, we obtain an enumeration algorithm for bases of a matroid. We now describe the details of the algorithm as follows.

```
ALGORITHM: T&B_for_Matroid_Bases ($\mathcal{M}$)
Step 1: Trim $\mathcal{M}$.
Step 2: Find a base $B$ of $\mathcal{M}$.
Step 3: Call T&B_for_Matroid_Bases_iter ($B$, $\mathcal{M}$).
```

```
ALGORITHM: T&B_for_Matroid_Bases_iter ($\mathcal{M}$)
Step 1: Choose an element $e \in B$.
Step 2: Find a base $B'$ not including $e$.
Step 3: Generate and trim $\mathcal{M}/e$ and $\mathcal{M} \setminus e$.
Step 4: If one of the subproblems loses some elements, partition the problem again
        by the balancing algorithm.
Step 5: Solve the subproblems by recursive calls.
```

The algorithm T&B_FOR_MATROID_BASES spends

$$O(T_{cont}(m,n) + T_{delt}(m,n) + \min\{(m-n)nT_{ind}(m,n), (m-n)T_{cir}(m,n)\})$$

time. An iteration of the algorithm T&B_FOR_MATROID_BASES_ITER may also spend this computation time, although the total computation time will be bounded by a smaller order. In the next subsection, we estimate an upper bound of the amortized time complexity by the analysis of the trimming and balancing approach.

5.2.2 Bounding the Amortized Time Complexity

The algorithm in the above subsection is quite simple and natural. It may spend much computation time in an iteration in the worst case, but the analysis of the trimming and balancing approach works efficiently on the algorithm. We can reduce the time complexity to $1/(m-n)n$ of the original computation time per an iteration.

To bound the total time complexity, we consider the enumeration tree of the algorithm. Each vertex of the enumeration tree corresponds to each recursive call, and each edge connects two vertices if the recursive call corresponding to one of the vertices occurs in the other. There is a one-to-one correspondence between all leaves and all bases. Any internal vertex has at least two children. Each recursive call corresponding to a vertex x inputs a matroid. We denote the input matroid by $\mathcal{M}_x$. We also denote the size of the grand set of $\mathcal{M}_x$ by m_x and the rank by n_x.

Firstly we establish a lower bound $\bar{D}(x)$ of the number of descendants of x for a vertex x in the enumeration tree. It is given by the lower bound of the number of bases in a trimmed matroid.

Lemma 27 *A trimmed matroid $\mathcal{M}$ includes at least $(m-n)n$ bases.*

Proof: We prove the lemma by induction. Let B be a base of $\mathcal{M}$. For each element e in B, there is an element not in B whose elementary circuit includes e. By exchanging e and the element, we obtain the other base. The generated bases are distinct, hence we have at least n other bases in the matroid. For each element e' not in B, we have an elementary circuit including at least one element of B. By exchanging e' and an element of the circuit, we can obtain $m-n$ distinct bases. Hence we have at least $m-n$ bases in the matroid. Therefore the condition holds if $n=1$ or $m-n=1$.

Suppose that the condition holds if $m-n<k$ and $n \le k'$ hold, or $m-n \le k$ and $n<k'$ hold. Under this assumption, we prove that the condition holds for the matroid $\mathcal{M}$ with $m-n=k>1$ and $n=k'>1$. Let e be an element of the grand set of $\mathcal{M}$. If the matroids $\mathcal{M}/e$ and $\mathcal{M} \setminus e$ lose no element by the trimming algorithm, $\mathcal{M}$ includes at least $(m-n-1)n + (m-n)(n-1) = 2(m-n)n - m$ bases. Since $m-n, n > 1$, we have $2(m-n)n - m > (m-n)n$.

Let us consider the other cases. If $\mathcal{M}/e$ loses $k, 1 \le k \le m-n$ elements by the trimming algorithm, then the bases of $\mathcal{M}$ can be partitioned to bases of at least $k+1$ trimmed matroids with $m-k-1$ elements and the rank $n-1$ by the above balancing algorithm. Since they have at least $(m-k-1-(n-1))(n-1)$ bases from the assumption, $\mathcal{M}$ has at least $(k+1)(m-k-n)(n-1) \geq (m-n)n$ bases.

In the case that $\mathcal{M} \setminus e$ loses $k, 1 \leq k \leq n$ elements, the bases of $\mathcal{M}$ are partitioned to bases of $k+1$ matroids with $m-k-1$ elements and the rank $n-k$. Since they all have at least $(m-k-1-(n-k))(n-k)$ bases from the assumption, $\mathcal{M}$ has at least $(k+1)(m-n-1)(n-k) \geq (m-n)n$ bases. ▮

Next we estimate an upper bound X^* of excess vertices. We set α to 30. From Corollary 1, we can set X^* to an upper bound of the number of vertices x in path from the root to a leaf satisfying the condition (a) that $\bar{D}(x)$ is less than the sum of $\frac{\alpha}{\alpha+1}\bar{D}(u)$ over all children u of x. We show which vertices of the enumeration tree satisfy the condition. We firstly consider the case that $m_x - n_x = 1$ or $n_x = 1$. In the case, the balancing algorithm generates $m_x - n_x$ or n_x subproblems with only one element in the grand set. It may be an excess vertex, although the number of this type of excess vertices in the path from the root to a leaf is at most 1.

Next we consider the case that $m_x - n_x, n_x > 1$. If both generated subproblems of x lose no element by the trimming algorithm, the number of bases included in $\mathcal{M}/e$ or $\mathcal{M} \setminus e$ is at least $(m_x - n_x - 1)n_x + (m_x - n_x)(n_x - 1) = 2(m_x - n_x)n_x - m_x$. For a fixed m_x, $(m_x - n_x)n_x$ gets the minimum value $2m_x - 2$ for $n_x = 2$ and $n_x = m_x - 2$. Since $2m_x - 2 \geq m_x/30$ for any $m_x > 2$, we have $(m_x - n_x)n_x \leq \frac{30}{31}(2(m_x - n_x)n_x - m_x)$. Hence x does not satisfies the condition (a). Suppose that x is an excess vertex. Then, a subproblem of x loses k elements by the trimming algorithm. Thus we have $\frac{31}{30}(m_x - n_x)n_x > (k+1)(m_x - k - n_x)(n_x - 1)$ or $\frac{31}{30}(m_x - n_x)n_x > (k+1)(m_x - n_x - 1)(n_x - k)$ from the proof of the above lemma. Hence we have

$$\frac{1.04 n_x}{n_x - 1} > \frac{(k+1)(m_x - k - n_x)}{(m_x - n_x)}, \text{or}$$

$$\frac{1.04(m_x - n_x)}{m_x - n_x - 1} > \frac{(k+1)(n_x - k)}{n_x}.$$

The former condition holds for only $k > m_x - n_x - 4$, and the latter condition holds for only $k > n_x - 4$. Thus any child y of x satisfies $m - n \leq 4$ or $n \leq 4$. The rank and the cardinality of the grand set decrease strictly in the children, hence there are at most 8 excess vertices in a path from the root to a leaf. At last, we consider the case with $n_x = 1$ or $m_x - n_x = 1$. In the case, the balancing algorithm generates $m_x - n_x$ or n_x subproblems with only one element in the grand set. Hence the balancing algorithm generates at most 9 excess vertices on the path from the root to any leaf in the enumeration tree. Therefore we can set X^* to 9.

Let $T(m,n)$ be $[T_{cont}(m,n) + T_{delt}(m,n) + (m-n)\min\{T_{ind}(m,n), T_{cir}(m,n)/n\}]/(m-n)n$. An iteration of the algorithm takes $\mathrm{O}((m_x - n_x)n_x T(m_x, n_x))$ time, thus we set $T(x)$ to $\mathrm{O}((m_x - n_x)n_x T(m_x, n_x))$. We also set T^* to $\mathrm{O}(1)$. By using $\bar{D}(x)$, we set $\hat{T}$ to $\max_x\{\mathrm{O}((m_x - n_x)n_x T(m_x, n_x)/\bar{D}(x))\} = \mathrm{O}(T(m,n))$. Hence we have that $\mathrm{O}(T^* + \hat{T}X^*) = \mathrm{O}(T)$. Since the number of iterations does not exceed twice the number of bases, we obtain the following theorem from Theorem 9.

Theorem 11 *All bases of a matroid can be enumerated in* $O(T_{cont}(m,n) + T_{delt}(m,n) + (m-$

$n)\min\{nT_{ind}(m,n), T_{cir}(m,n)\}$) preprocessing time, and $O([T_{cont}(m,n)+T_{delt}(m,n)]/n(m-n)+\min\{T_{ind}(m,n), T_{cir}(m,n)/n\})$ time per a base, if the time complexities of oracle algorithms are depend on the input size for contracted matroids and deleted matroids.

∎

5.2.3 On Graphic Matroids

In the following subsections, we show some practical cases of enumeration algorithms for matroid bases. Firstly, we see the case of graphic matroids. The grand set of a graphic matroid is given by the edge set of an undirected graph, and its independent set is given by a forest of the graph. It is known that the set of all forests satisfies the axiom of the independent set family [43]. If the graph is connected, all bases of the graphic matroid correspond to all spanning trees. Hence the problem is that of enumerating all spanning trees (or spanning forests in the case of disconnected) of given undirected graphs. There are many studies for this problem. Numerous algorithms have been proposed and improved [23, 32, 37, 44, 48, 49]. We also have some studies on the finding k-best problem in the weighted case [14, 33]. The time complexities of the enumeration algorithms have been reduced from $O(m+n)$ to $O(n)$, and to $O(1)$ per a spanning tree. The space complexity has also been reduced from $O(nm)$ to $O(m+n)$. Here m and n denotes the number of edges and the size of a spanning tree, respectively. These improving methods are relied on some advanced data structures, although our algorithm in this subsection requires neither complicated algorithm nor advanced data structure.

Suppose that $\mathcal{M}$ is a graphic matroid given by an undirected connected graph $G=(V,E)$. Since any spanning tree of G has $|V|-1$ edges, we have $|E|=m$ and its rank n is $|V|-1$. A base of a graphic matroid can be found by a graph search algorithm in $O(m+n)$ time. The contraction and the deletion of the matroid are also easy. $\mathcal{M}/e$ is given by the graph obtained by contracting the edge e, and $\mathcal{M}\setminus e$ is given by the graph obtained by deleting e. Since any independent set of the matroid includes no cycle, a circuit of the matroid is a cycle in G. Thus an elementary circuit oracle algorithm can be realized with $O(n)$ running time.

From these, we have $T_{cont}(m,n)=O(n)$, $T_{delt}(m,n)=O(m-n)$, and $T_{cir}(m,n)=O(n)$. By using these algorithms, the time complexity of an iteration of the trimming and balancing algorithm is bounded by $O((m-n)n)$. Since all loops are detected in $O(m)$ time and all elements with elementary cuts composed of only themselves can be found by the 2-connected component decomposition in $O(m+n)$ time [50], the time complexity of the trimming algorithm can be reduced to $O(m+n)$ time. Therefore the trimming and balancing algorithm takes $O(m+n)$ time for preprocessing and $O(\frac{O((m-n)n)}{(m-n)n})=O(1)$ time per a spanning tree. It requires only $O(m+n)$ memory. These complexities are equal to an optimal algorithm of Shioura et al.[49]. Here we have the following theorem.

Theorem 12 *All bases of a graphic matroid can be enumerated in $O(m+n+N)$ time and $O(m+n)$ memory by a trimming and balancing algorithm where N denotes the number of bases.*

∎

5.2.4 On Linear Matroids

This subsection describes a trimming and balancing algorithm for linear matroids. Linear matroids are the origin of matroids. To generalize the properties of the independence of vectors, the matroids were considered. For an m by n matrix M with $m > n$, the grand set of the linear matroid is given by the set of column vectors, and an independent set is given by a set of independent column vectors. The independent set family is given by the correction of all the independent sets. The family is known to satisfy the axiom of the independent set family [43]. If the given matrix is not degenerate, the bases of the linear matroid are all submatrices composed of n independent column vectors. For a given non-singular m by n matrix, we consider the problem of enumerating all these n by n non-singular submatrices, which are composed of n independent column vectors.

A base of the linear matroid can be found easily. By some LDL or LU decomposition algorithms, we can obtain a non-singular submatrix B in at most $\mathrm{O}(n^2m)$ time. Our algorithm for linear matroids utilizes an elementary circuit oracle algorithm. To obtain a fast elementary circuit oracle algorithm, we consider a linear matroid equivalent to $\mathcal{M}$. The matroid is given by a matrix $U(M, B)$ which is generated from M as follows. All ith column vectors of B is replaced by the unit vector e_i such that whose ith element is 1 and the other elements are 0. A column vector $x \notin B$ is replaced by the vector y satisfying $By = x$. Since a set of column vectors of M is independent if and only if the set of column vectors corresponding to them is independent in $U(M, B)$. Hence the linear matroid given by $U(M, B)$ is equivalent to $\mathcal{M}$. We enumerate all bases of the matroid instead of $\mathcal{M}$.

Since all column vectors of B are unit vectors, the elementary circuit of a column vector x can be easily found. The ith column vector of B is in $Cir(B, x)$ if and only if the ith element of x is not 0. In each iteration, we exchange ith column vector of B and the other column vector x, and obtain the other base B'. Generally, x is not a unit vector, thus we have to obtain $U(M, B')$ in each iteration. Since only x is not a unit vector in B', for any column vector z of M, we can obtain a vector y satisfying $B'y = z$ in $\mathrm{O}(n^2)$ time. Therefore, we take $\mathrm{O}(n^2(m - n))$ time to obtain $U(M, B')$.

We next show the contracting and the deleting operation of a linear matroid. The deleting operation is quite simple, since it is done by deleting a column vector. Since we delete a column vector outside the base in each iteration, we are not in need to obtain $U(M, B')$. To contract the matroid by the ith column vector x_i, we fix x_i in the base B, and consider only the bases including x_i. Since x_i is included in any base, we can restrict the rest column vectors to the linear subspace orthogonal to x_i. Hence we consider the matrix obtained by deleting ith element from all column vectors of M. Since the linear matroid given by the matrix is equivalent to $\mathcal{M}/x_i$, we enumerate all bases of the matroid instead of $\mathcal{M}/x_i$.

Each the above operations takes at most $\mathrm{O}(n^2(m - n))$ time, thus the time complexity of an iteration is $\mathrm{O}(n_x^2(m_x - n_x))$ for a vertex x inputting an m_x by n_x matrix. Since $n^2(m - n)/n(m - n) = n$, we obtain the following theorem by applying the analysis of trimming and balancing approach.

Theorem 13 *All bases of the linear matroid given by an m by n matrix are enumerated in $O(n^2m)$ preprocessing time and $O(n)$ time per a base.*

▮

5.2.5 On Matching Matroids

Matching matroids are given by bipartite graphs. For a bipartite graph $G = (V_1 \cup V_2, E)$, the grand set of a matching matroid is given by V_1, and an independent set of the matroid is given by a set of vertices of V_1 which is covered by a matching of G. Note that a matching of a graph is an edge set satisfying that no two edges of it share their endpoints. A base B of a matching matroid is obtained by finding a maximum cardinality matching M of G. For a vertex $v \in V_1 \setminus B$, $u \in B$ is in $Cir(B, v)$ if and only if there is an alternating path from v to u in G. An alternating path is a path of G satisfying that edges of M and the other edges appear in the path alternatively. By exchanging edges of M and the other edges along the alternating path from v to u, we obtain the other maximum matching M' which covers v and does not cover u. To obtain the elementary circuit of v, we find the vertices of the base satisfying that there are alternating paths from v to the vertices. We can find them in $\mathrm{O}(m+n)$ time by using a graph search algorithm [27, 18].

To delete a vertex from a matching matroid is easy. It is done by deleting the vertex from G. The contracted matroid is not obtained by reforming of G. We mark the vertex to contract, and never output marked vertices when we find elementary circuit. By this modification of the elementary circuit oracle algorithm, we can obtain the contracted matching matroid.

In each iteration x, we have to spend $\mathrm{O}((|E_x| - |V_x|)(|E| + |V|))$ time where E_x and V_x denote the edge set and vertex set of the input graph of x, respectively. Note that E is the edge set of the graph of original problem. By the analysis of the trimming and balancing approach, the time complexity of the algorithm is $\frac{\mathrm{O}((|E_x|-|V_x|)(|E|+|V|))}{|V_x|(|E_x|-|V_x|)} = \mathrm{O}(|E| + |V|)$ per an iteration. Therefore we have the following theorem.

Theorem 14 *All bases of a matching matroid given by a bipartite graph $G = (V_1 \cup V_2, E)$ can be enumerated in $O(|E| + |V|)$ time per a base.*

▮

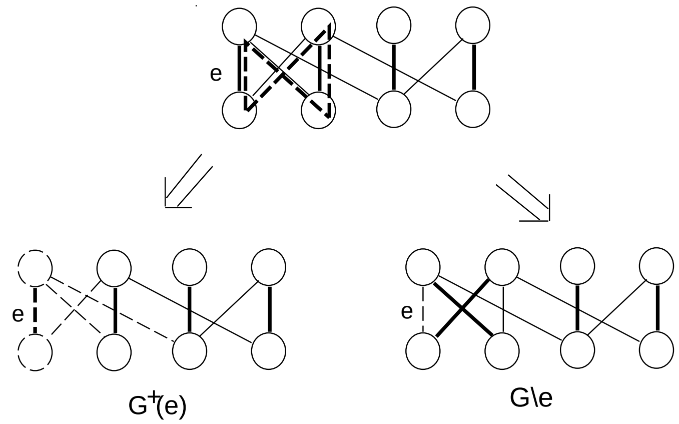

Figure 5.7: A perfect matching M' is generated from M by exchanging edges along an alternating cycle. Two subproblems with the graph $G^+(e)$ and M, and $G \setminus e$ and M' are generated. Dotted lines are deleted edges from the graph G.

5.3 Enumerating Perfect Matchings in Bipartite Graphs

In this section, we consider the problem of enumerating all perfect matchings in a bipartite graph $G = (V_1 \cup V_2, E)$, and propose a trimming and balancing algorithm for the problem. Here m and n denote the numbers of edges and vertices, respectively. In this thesis, we propose enumeration algorithms for some kinds of matchings. Among these algorithms, the algorithm for perfect matching is the most fundamental one. The algorithm will be modified to obtain algorithms for covering matchings in the next section. The algorithm is also utilized to enumerate all maximal matchings in bipartite graphs in the previous chapter.

Matchings are very fundamental objects in the graph theory and the optimization theory. There are numerous studies on properties of matchings. Many efficient algorithms have been proposed for a variety of matching problems. For example, for the problem of finding a maximum cardinality matching, an algorithm running in $\mathrm{O}(n^{1/2}m)$ time is proposed in [27].

In 1993, K. Fukuda and T. Matsui proposed a binary partition algorithm for enumerating perfect matchings [18]. The running time of the algorithm is $\mathrm{O}(m+n)$ per a perfect matching. This algorithm is the base of our algorithm. We apply the trimming and balancing approach to it for speeding up. In the trimming phase of each iteration, we remove edges which are included in all perfect matchings or no perfect matching. By some graph search algorithms, it can be done in $\mathrm{O}(m+n)$ time. In the balancing phase, we find an edge which partitions the problem such that both generated subproblems are not so small. The edge satisfies that one of the subproblem includes at least $m/2 - n$ perfect matchings. By these algorithms, we can reduce the time complexity of the algorithm from $\mathrm{O}(m+n)$ to $\mathrm{O}(n)$ per a perfect matching.

In the following subsections, we show these algorithms and some properties utilized by them. We also show the time complexities of these algorithms, and analyze the total time complexity of the enumeration algorithm.

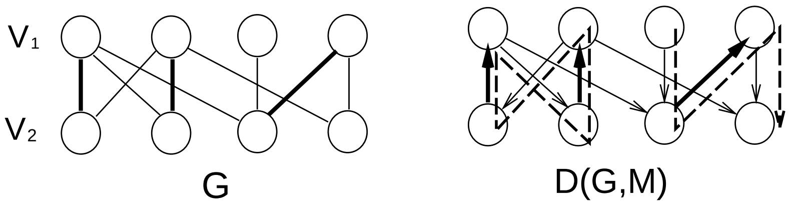

Figure 5.8: The graph $D(G, M)$ and a directed cycle and a directed path corresponding an alternating cycle and an alternating path in G.

5.3.1 An Algorithm for Perfect Matchings

In this subsection, we describe an algorithm for enumerating perfect matchings in a given bipartite graph. Our algorithm is based on the binary partition algorithm proposed by Fukuda and Matsui, hence we firstly explain their algorithm.

For a given bipartite graph, their algorithm firstly finds a perfect matching M of the graph. We spend $O(n^{1/2}m)$ time to do this [27]. After finding M, the algorithm starts enumerating all perfect matchings by checking whether there is another perfect matching or not. If there is no other perfect matching, the algorithm stops. Otherwise, it finds a new perfect matching M'. The algorithm selects an edge e included in just one matching among M and M'. By using the edge, the algorithm partitions the problem into two subproblems. One of them is to enumerate all perfect matchings including e, and the other is to enumerate all those not including e. For these subproblems, the algorithm constructs two subgraphs $G^+(e)$ and $G \setminus e$ of G. $G^+(e)$ is the graph obtained by deleting e, both endpoints and all edges adjacent to e. Any perfect matching of $G \setminus e$ is a perfect matching of G not including e, and vice versa. Any perfect matching of $G^+(e)$ plus e is a perfect matching of G including e, and vice versa. The algorithm solves the subproblems by two recursive calls with these subgraphs. One of them inputs a graph $G^+(e)$ and a perfect matching $M \setminus e$ of $G^+(e)$, and the other inputs $G \setminus e$ and M' (see Figure 5.7). In the subproblems, we are not in need to find perfect matchings of the input graphs, since $M \setminus e$ and M' are perfect matchings of those graphs.

An important part of the algorithm is how to find the other perfect matching. The symmetric difference between M and the other matching is composed of some cycles and paths. In these cycles and paths, the edges of M and the other edges appear alternatively. The ends of the paths are not adjacent to edges of M. We call these cycles and paths *alternating cycles* and *alternating paths* (see Figure 5.8). A matching generated by an alternating cycle has the cardinality equal to M. Since a perfect matching covers all vertices, the symmetric difference between two perfect matchings is composed only of alternating cycles. Hence the necessary and sufficient condition of the existence of another perfect matching is that there exists an alternating cycle. On the other hand, since the two ends of any path are matching edges if M is perfect, any alternating path generates a non-perfect matching.

To find alternating cycles and paths, we utilize a directed graph $D(G, M)$ defined for a graph G and a matching M. The vertex set of $D(G, M)$ is given by V. The arc set of $D(G, M)$ is given by orienting edges of M from V_1 to V_2, and the other edges of G in the opposite direction. For any directed cycle or directed path in the graph $D(G, M)$, edges of M and the other arcs appear alternatively in the corresponding cycle or path in G. Hence we can find the other perfect matching by finding a directed cycle of $D(G, M)$.

We spend $\mathrm{O}(m+n)$ time to find a directed cycle, and $\mathrm{O}(n)$ time for the other operations in an iteration. A new perfect matching is found in each iteration, thus the time complexity of their algorithm is $\mathrm{O}(m+n)$ per a perfect matching. The memory complexity is as follows. In each recursive call, we store the given graph. This requires $\mathrm{O}(m+n)$ space, but by storing only deleted edges when a recursive call occurs, we can reduce the size of the required memory space. Since the number of these deleted edges does not exceed m, the accumulating storing space is bounded by $\mathrm{O}(m+n)$. In the next subsection, we add some modifications to the algorithm. We analyze its time complexity in detail, and bound it by $\mathrm{O}(n)$ per a perfect matching.

5.3.2 Properties for Bounding the Time Complexity

In this subsection, we describe our trimming algorithm and balancing algorithm. Our trimming algorithm removes all unnecessary edges before generating a subproblem. Unnecessary edges are characterized by edges included in no alternating cycle. Since the algorithm treats only edges included in alternating cycles, we can delete these unnecessary edges. These arcs can be found in $\mathrm{O}(m+n)$ time by applying a strongly connected component decomposition algorithm to $D(G, M)$. By this, we can save the unnecessary time for operating those edges.

The balancing algorithm selects an edge e included in M and not included in the other perfect matching by finding an alternating cycle. The edge e satisfies that $G^{+}(e)$ includes at least $m/6$ perfect matchings if $m > 6n$. To obtain e, we find an edge such that at least $m/2 - n$ edges are included in some alternating cycles not including e. Since the graph is trimmed, e always exists, and $G^{+}(e)$ includes at least $m/2 - 2n$ perfect matchings. By this, we can bound the total amount of the time spent by all the iterations.

To analyze the time complexity, we introduce the *enumeration tree* of the algorithm, which captures the movement of the algorithm. It is defined for an algorithm and an input. The enumeration tree of our algorithm is defined for input graphs. For a given graph, each vertex of an enumeration tree $\mathcal{T}$ corresponds to a recursive call of the algorithm, and if a recursive call occurs in another recursive call, an edge connects the two corresponding vertices. The root vertex of the tree corresponds to the start of the algorithm. Each recursive call corresponding to a vertex x of $\mathcal{T}$ inputs a graph. We denote the input graph by G_x. We show some properties such that the enumeration tree for any input satisfies them.

In the case that the input graph G_x has at least two perfect matchings, the recursive call corresponding to x generates exactly two recursive calls. Otherwise, it terminates immediately, and corresponds to a leaf of $\mathcal{T}$. Therefore all internal vertices of $\mathcal{T}$ have exactly two children, and all the leaves are corresponding to all the perfect matchings.

By using the enumeration tree, the time complexity of the algorithm is given by the sum of $|E(G_x)| + |V(G_x)|$ over all vertices of $\mathcal{T}$, where $E(G_x)$ and $V(G_x)$ are the edge set and the vertex set of the graph G_x. In [18], they claim that the sum of computation time of their algorithm is bounded by $\mathrm{O}((m+n)N)$ where N is the number of perfect matchings. The

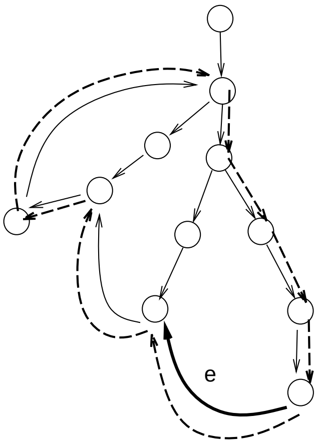

Figure 5.9: Dotted lines compose the generated cycle for an arc $e \notin T$ by using the tree T.

time complexity of our algorithm can be bounded by a better bound. In the following, we estimate some statements to analyze the time complexity with the use of the distributing rule. We assume that G contains a perfect matching M, and all arcs in $D(G, M)$ are included in some directed cycles through this subsection since we remove unnecessary edges in each iteration,

Lemma 28 *G contains at least $m - n + 2$ perfect matchings.*

Proof: Let H be a strongly connected component of $D(G, M)$, and T be a depth-first search tree of H. We assume that the search traverses an arc from its tail to its head. Since H is strongly connected, T spans all vertices of H. We put indices to all vertices of H by the post-ordering of the depth-first search. In the ordering, a vertex has an index larger than any its descendant, and for any non-back arc of H, the index of its tail is larger than its head. A non-back arc is an arc not in T whose head is not an ancestor of its tail. Conversely, we call an arc not in T such that its head is an ancestor of its tail a back arc. Note that a back arc has an index on its tail smaller than its head.

Let the index of an arc in T be zero, and the index of an arc not in T be the index of its tail. We will show that there are distinct directed cycles for each arc in $H \setminus T$. For each back arc e of T in H, we can generate its own directed cycle by adding e to T, so that e is the maximum index arc in the cycle. For each non-back arc e of T in H, let P be a directed path from its head to its tail. Since T is a depth-first search tree, P includes some ancestors of the tail of e (see Figure 5.9) .

To see the reason, we claim the following fact.

Claim 1 *Suppose that three vertices v, w and u satisfy that the index of v is less than or equal to w, and the index of w is less than u. Then, any directed path from v to u includes a common ancestor of u and w.*

Proof: Suppose that there exists a simple directed path P' from v to u not including common ancestors. Let c be the ancestor of v which is the child of the nearest common ancestor of v

and u. Note that the common ancestor is an ancestor of w. By connecting the directed path from c to v on T and P', we obtain a directed path from c to u without any ancestor of c. This contradicts that T is a depth-first search tree. ▮

From the claim, P includes at least one ancestor of the tail of e. Let v be the first vertex in P to appear among these ancestors. Note that v is also an ancestor of the head of e from the claim. We can make a directed cycle by merging the subpath P' of P from the head of e to v, the directed path in T from v to the tail of e, and e. In this cycle, e is the maximum index arc. If not, then P' includes a vertex whose index is larger than e. Note that the vertex is not a descendant of v. From the claim, P' includes a common ancestor of the vertex and v, hence it contradicts the definition of v. Since all arcs in a directed cycle have distinct tails, the maximum index arc is unique. Therefore all the generated directed cycles are distinct.

Now we have distinct directed cycles for each arc not in T. The number of arcs not in T is $|E(H)| - |V(H)| + 1$ for any strongly connected components H in $D(G, M)$. Therefore we have $|E(G)| - |V(G)| + 2 = m - n + 2$ perfect matchings in G. ▮

Next, we show that there exists an edge e such that at least $m/2 - n$ edges of $G^+(e)$ are included in some alternating cycles. Let $D'(G, M)$ be the graph obtained by contracting all arcs of $D(G, M)$ corresponding to the edges of M. $D'(G, M)$ has $m - (n/2)$ arcs and $n/2$ vertices. Each vertex of $D'(G, M)$ corresponds to each matching edge. $D'(G^+(e), M \setminus e)$ is equal to the graph obtained by removing the vertex corresponding to e from $D'(G, M)$. Without loss of generality, we assume that $D'(G, M)$ is strongly connected.

Lemma 29 *There is a vertex v^* such that at least $m/2 - n$ arcs of $D'(G, M)$ are included in directed cycles not including v^*.*

Proof: To obtain the vertex, we find a depth-first search tree T in $D'(G, M)$ with the root vertex r. Suppose that v denotes the vertex last visited by the search. v is a leaf of T. Indices are assigned to all vertices by the post-ordering of the search. Let us consider the graph obtained by removing v from $D'(G, M)$. Let e be an arc not incident to v and satisfying that at least one of its endpoint is not on the directed path from r to v in T. If e is a back arc, it is included in some directed cycles after removing v. If e is not a back arc, then there is a directed path P from the head of e to v since $D'(G, M)$ is strongly connected. From Claim 1, P must include a common ancestor w of v and the head of e. By merging e, the subpath of P from the head of e to w, and the directed path from w to the tail of e in T, we obtain a directed cycle including e. Since these paths do not include v, the cycle does not include v. Therefore, if an arc not in T is included in only directed cycles including v, then it is a non-back arc connecting two ancestors of v. We denote the set of these non-back arcs connecting two ancestors of v by A.

If at least $\lceil m/2 - n \rceil$ arcs are included in directed cycles after removing v, v satisfies the condition of v^*. Hence let us consider the other case, which is that at least $\lceil m - n/2 - (m/2 - n) \rceil = \lceil m/2 + n/2 \rceil$ arcs are included in only directed cycles containing v. In this case, A includes at least $\lceil m/2 \rceil$ arcs. Let u_1 be the vertex with the largest index among ancestors of v which are heads of some arcs of A. Let P be a directed path from v to r, and u_2 be the ancestor of v with the largest index in P except for r. We suppose that u denotes the parent of the vertex with the largest index among u_1 and u_2.

Let us consider how many arcs of A are included in directed cycles in the graph obtained by removing u. Let a be an arc of A which is not incident to u. If u is r, then u_2 is a child of r since there is no non-back arc from r to a child of r. By jointing the subpath of P from v to u_2 and the directed path from u_2 to v in T, we can obtain a directed cycle including v and u_2. Thus we can obtain a directed cycle including a by adding a to the cycle. Let us consider the other case. Let D be the directed graph obtained by deleting arcs incident to u from the union of P and the directed path from r to v in T. The graph is composed of a non-simple directed cycle including all the vertices of the path from u_2 to v in T, an arc from a descendant of u_2 to r, the directed path from r to the parent of u, and the directed path $\hat{P}$ from the child of u to u_2. Hence if the tail of an arc a of A is not in $\hat{P} \setminus u_2$, then we can obtain a directed cycle including a and not including u by adding a to D. If the tail of a is in $\hat{P} \setminus u_2$, then u_1 is a child of u. Hence there is a non-back arc connecting a proper ancestor of u and u_1. By adding the arc and a to D, we can obtain a directed cycle including a without u.

Since at most $n/2$ arcs can be incident to u, at least $\lceil (m-n)/2 \rceil$ arcs are included in some directed cycles after removing u from $D'(G,M)$. Therefore u satisfies the condition of v^*. ∎

From the lemma, we can construct an algorithm for finding such a vertex. We describe the algorithm as follows.

ALGORITHM: FIND_PARTITION_EDGE

```
Step 1: Find a depth-first search tree $T$ in $D'(G,M)$ with the root vertex $r$.
Step 2: Apply a strongly connected component decomposition algorithm to the graph
        obtained by removing the last visited vertex $v$. If $\lceil m/2 - n \rceil$ edges are
        included in some directed cycles, output $v$ and stop.
Step 3: Find $u_1$ and $u_2$.
Step 4: Output the vertex nearest from $r$ among the parents of $u_1$ and $u_2$.
```

Since each step takes $O(m+n)$ time, the time complexity of this balancing algorithm is $O(m+n)$. By adding the trimming algorithm and the balancing algorithm, we can obtain a trimming and balancing algorithm. We describe the whole algorithm as follows.

ALGORITHM ENUM_PERFECT_MATCHINGS (G)

```
Step 1: Find a perfect matching $M$ of $G$. If $M$ is not found, stop.
Step 2: Remove all unnecessary edges from $G$ by using a strongly connected component
        decomposition algorithm.
Step 3: Call ENUM_PERFECT_MATCHINGS_ITER $(G,M)$.
```

ALGORITHM ENUM_PERFECT_MATCHINGS_ITER (G,M)

```
Step 1: If $G$ has no edge, output $M$ and stop.
Step 2: Find a partitioning edge $e$ by FIND_PARTITION_EDGE.
Step 3: Find a directed cycle including $e$ by a depth-first search algorithm.
Step 4: Generate a perfect matching $M'$ by exchanging edges along the cycle.
Step 5: Trim unnecessary edges from $G^+(e)$.
Step 6: Enumerate all perfect matchings including $e$ by ENUM_PERFECT_MATCHINGS_ITER
```

with the obtained graph and M.

Step 7: Trim unnecessary edges from $G \setminus e$.

Step 8: Enumerate all perfect matchings not including e by ENUM_PERFECT_MATCHINGS_ITER with the obtained graph and M'.

Theorem 15 *Perfect matchings in a bipartite graph can be enumerated in $O(n^{1/2}m + nN)$ time and $O(m + n)$ space where N is the number of perfect matchings in the input graph.*

Proof: To bound the time complexity, we consider a distributing rule of computation time. For an internal vertex x of $\mathcal{T}$, if $|E(G_x)| \geq 6|V(G_x)|$ holds, then we distribute its computation time to all the descendants of the child of x inputting $G_x^+(e)$ where e is the edge used to partition the problem in the iteration x. From Lemma 29, at least $|E(G_x)|/3$ edges of $G_x^+(e)$ are included in some directed cycles. Hence, from Lemma 28, $G_x^+(e)$ contains at least $|E(G_x)|/3 - |V(G_x)| \geq |E(G_x)|/6$ perfect matchings. Therefore the child inputting $G_x^+(e)$ has at least $|E(G_x)|/6$ descendants. Since the computation time on x is distributed uniformly, a vertex receives O(1) computation time from an ancestor. Since we distribute the computation time to only descendants of the child inputting $G_x^+(e)$, a vertex inputs a perfect matching M receives computation time from at most $|M|$ ancestors. Therefore, each vertex is distributed at most O(n) computation time. ∎

5.4 Enumeration Algorithms for Some Matching Problems

For a given bipartite graph $G = (V, E)$ and a vertex subset $\hat{V} \subseteq V$, we define a covering matching by a matching such that any vertex of $\hat{V}$ is incident to an edge of the matching. This section deals two problems of enumerating covering matchings, and proposes two algorithms for solving them. The first one is to enumerate all the covering matchings for a given bipartite graph and a vertex set. The second one is to enumerate all the covering matchings with the maximum cardinality for a given bipartite graph and a vertex set. We also show some their applications. Covering matchings are utilized in some graph algorithms. For example, an algorithm for finding a minimum edge coloring of a bipartite graph utilizes matchings covering all the vertices with the maximum degree. If $\hat{V} = V$, then a covering matching is a perfect matching.

A covering matching can be found in $\mathrm{O}(nm)$ time. The symmetric difference between a matching M and a covering matching M' is composed of alternating paths and cycles. If M is not a covering matching, then the symmetric difference always includes some alternating paths one of whose endpoint is in $\hat{V}$ and not covered by M. Hence, by exchanging edges along the alternating path, we can obtain a matching covering at least one more vertex of $\hat{V}$ than M. For a matching, these alternating paths can be detected in $\mathrm{O}(m + n)$ time by some graph search algorithms. Therefore we can find a converting matching in $\mathrm{O}(nm)$ time if it exists.

To enumerate covering matchings, we use binary partition scheme similar to the algorithm in the previous section. For a covering matching M, we find the other covering matching M'. We choose an edge e which is included in only one matching among M and M', and partition the problem into two subproblems. The subproblems are to enumerate all covering matchings including e, and to enumerate those not including e. These two subproblems are solved recursively with the graph $G^+(e)$ and $G \setminus e$. The set of all the covering matchings not including e is the set of all the covering matchings in $G \setminus e$, and the set of all the covering matchings including e minus e is the set of all the matchings in $G^+(e)$ covering $\hat{V}$ minus the endpoints of e.

The symmetric difference between M and the other covering matching is composed of alternating cycles and paths whose endpoints are not in $\hat{V}$. Thus, to obtain the other covering matching, we find such alternating cycles and paths. These cycles and paths can be found in $\mathrm{O}(m + n)$ time by applying some graph search algorithm to $D(G, M)$, hence we obtain a simple algorithm running in $\mathrm{O}(m + n)$ time per a covering matching.

The time complexity of this simple algorithm can be also reduced by using the technique in the previous section. Let us consider a directed graph $\hat{D}(G, M)$ obtained by the following operation. We contract all the uncovered vertices of V_1 to s_1. If V_1 has no uncovered vertex, we make a new vertex s_1. We also contract all the uncovered vertices of V_2 to s_2. If V_2 has no uncovered vertex, we make a new vertex s_2. We make an arc $\hat{e}$ from s_1 to s_2. We also make arcs from each vertex in $\hat{V} \cap V_2$ to s_1, and from s_2 to each vertex in $\hat{V} \cap V_1$. Any directed cycle in $D(G, M)$ is a directed cycle of $\hat{D}(G, M)$. A directed path connecting two

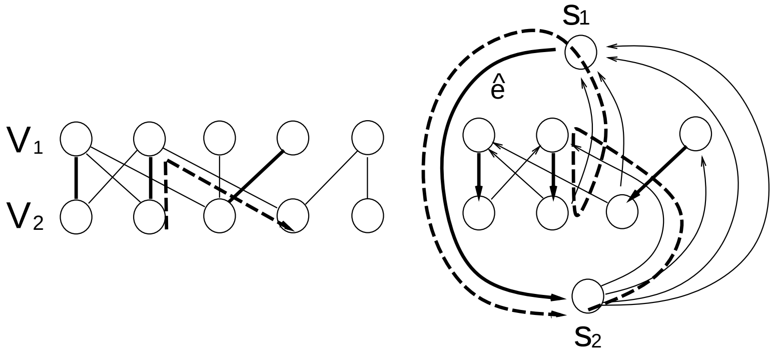

Figure 5.10: $\hat{D}(G,M)$ and an alternating path.

vertices not in $\hat{V}$ also gives a directed cycle of $\hat{D}(G,M)$ including $\hat{e}$. The cycle is obtained by connecting its endpoints with $\hat{e}$. Conversely, any directed cycle of $\hat{D}(G,M)$ is a directed cycle of $D(G,M)$ if it does not include $\hat{e}$, and otherwise includes a directed path connecting two vertices not in $\hat{V}$. Therefore we can find the other covering matching by finding a directed cycle of $\hat{D}(G,M)$. From the above observation, we can remove all arcs of $\hat{D}(G,M)$ included in no directed cycle when we find the other covering matchings. In each iteration, we remove these arcs from $\hat{D}(G,M)$. From Lemma 28, the obtained graph includes at least $|E(\hat{D}(G,M))| - |V(\hat{D}(G,M))| + 1$ directed cycles. Since $\hat{D}(G,M)$ includes at least $m+1$ arcs and at most $n+2$ vertices, G includes at least $m-n$ covering matchings.

To select an edge to partition the problem, we also use the algorithm Find_Partition_Edge. Let $\hat{m}$ be the number of arcs in $\hat{D}(G,M)$. Suppose that $\hat{e}$ is treated as an edge of M. We find an arc e^* such that at least $\hat{m}/2-(n+2)$ arcs are included in some directed cycles of $\hat{D}(G,M)^+(e^*)$. If e^* corresponds to an edge of G, then the problem will be partitioned exactly. Otherwise, e^* is $\hat{e}$ since e^* is a matching edge. In this case, we can not partition the problem. Thus we apply Find_Partition_Edge to $\hat{D}(G,M)^+(\hat{e})$, and obtain the other edge e^* which always corresponds to an edge of G. Since at least $\hat{m}/2-(n+2)$ arcs of $\hat{D}(G,M)$ are included in directed cycles without $\hat{e}$, e^* satisfies that at least $(\hat{m}/2-(n+2))/2-n = \hat{m}/4-3n/2-2$ arcs of $\hat{D}(G,M)^+(e^*)$ are included in some directed cycles.

From the above, $G^+(e)$ includes at least $m/12$ perfect matchings if $m > 12n$ holds. Thus for an iteration inputting a graph G with $m > 12n$, its child inputting $G^+(e)$ has at least $m/12$ descendants in the enumeration tree. By using the above method, for an iteration inputting a graph G with $m > 12n$, the child of the iteration inputting $G^+(e)$ has at least $m/12$ descendants in the enumeration tree. Hence, by using the analysis in the previous section, the time complexity is also bounded by O(n) per a covering matching.

Theorem 16 *All covering matchings for a vertex set in a bipartite graph can be enumerated in O(nm) preprocessing time and O(n) time per a covering matching.*

▮

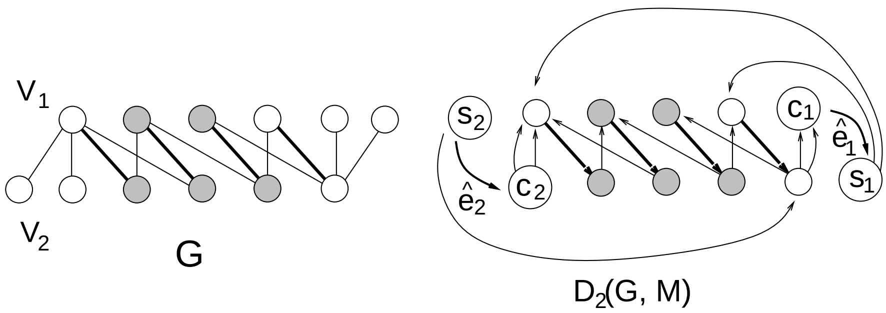

Figure 5.11: An instance of $D_2(G, M)$. Gray vertices are in $\hat{V}$

5.4.1 Enumerating Maximum Cardinality Covering Matchings

In the previous subsection, we considered the problem of enumerating all the covering matchings. This subsection describes an algorithm for a modification of the problem, which is to enumerate all the covering matchings with the maximum cardinality. If covering matchings exist in G, then a covering matching M with the maximum cardinality among all the covering matchings has the same number of edges as a maximum cardinality matching of G. Since if M is not a maximum matching, then the symmetric difference between M and a maximum matching includes an alternating path both whose end-edges are not in M. By exchanging edges along the path, we can obtain a covering matching whose cardinality is strictly larger than M. By using this property, we can find a maximum cardinality covering matching in O(nm) time if a covering matching exists.

Maximum cardinality covering matchings can be also enumerated by a simple binary partition algorithm. Any maximum cardinality covering matching including an edge e^* is a maximum matching in $G^+(e^*)$ covering all vertices of $\hat{V}$ except both endpoints of e^*, and vice versa. Any maximum cardinality covering matching not including e^* is a maximum cardinality covering matching in $G \setminus e^*$, and vice versa. Hence, similar to the algorithm of the previous subsection, we can divide the problem into two subproblems with an edge e^*. The edge e^* has to be included in a maximum cardinality covering matching and not included in the other maximum cardinality covering matching. For a maximum cardinality matching M, the edge is given by an edge in the symmetric difference between M and the other maximum cardinality covering matching.

To find the other maximum cardinality covering matching, we find alternating cycles or even length alternating paths whose endpoints are not in $\hat{V}$. We utilize a directed graph $D_2(G, M)$ to find them. $D_2(G, M)$ is constructed by (1) contracting all the uncovered vertices of V_1 to a vertex c_1, and all the uncovered vertices of V_2 to a vertex c_2, (2) adding a vertex s_1 to V_2, and a vertex s_2 to V_1, (3) making an arc $\hat{e}_1$ from c_1 to s_1, and an arc $\hat{e}_2$ from s_2 to c_2, and (4) making arcs from s_1 to all the covered vertices in $V_1 \setminus \hat{V}$, and arcs from all the covered vertices in $V_2 \setminus \hat{V}$ to s_2. Any alternating cycle of M corresponds to a directed cycle including neither $\hat{e}_1$ nor $\hat{e}_2$. Any even length alternating path whose endpoints are in V_1 corresponds to a directed cycle including $\hat{e}_1$. Any even length alternating path whose endpoints are in V_2 corresponds to a directed cycle including $\hat{e}_2$. There is no directed cycle including both $\hat{e}_1$

and $\hat{e}_2$, since such a directed cycle includes an alternating path whose both end edges are not included in M. It contradicts that M is a maximum matching. Therefore we can find the other maximum cardinality covering matching by finding a directed cycle of $D_2(G,M)$.

To reduce the time to find those cycles, we also use the technique in the previous section. In each iteration, we remove all arcs of $D_2(G,M)$ included in no directed cycle. By this, we can estimate a lower bound $m_2 - n - 2$ of the number of maximum cardinality covering matchings in G. Here m_2 denotes the number of edges in $D_2(G,M)$.

To partition the problem, we also apply the method in the previous section to $D_2(G,M)$. We apply Find_Partition_Edge to $D_2(G,M)$ repeatedly until we obtain an edge of G. In the worst case, we obtain e_1 and e_2 by the method, although the obtained edge e^* satisfies that $G^+(e^*)$ includes at least $m/8 - 7n/8 - 7$ edges. Hence $G^+(e^*)$ includes at least $m/32$ maximum cardinality covering matchings if $m > 32n$ and $n > 8$.

By using the above method to select the partitioning arc e in each iteration, an iteration inputting a graph G has at least $m/24$ descendants which are also descendants of the child with $G^+(e)$ in the enumeration tree. Hence, by using the analysis in the above section, the time complexity of the algorithm is also bounded by $O(n)$ time per a maximum cardinality covering matching. The memory complexity of the algorithm is $O(m+n)$.

Theorem 17 *All the maximum cardinality covering matchings for a vertex set in a bipartite graph can be enumerated in $O(nm)$ preprocessing time and $O(n)$ time per one with $O(m+n)$ memory space.*

∎

5.4.2 Enumerating Minimum Weight Matchings

The algorithms in the previous subsections can be utilized to solve the other enumeration problems. In this subsection, we show two applications of those algorithms.

Let $w : E \rightarrow R$ be an edge weight function. The weight of a matching is given by the sum of weights over all its edges. The weighted version of matchings have been studied. Some efficient algorithms are proposed for some optimization problems. For example, an algorithm running in $O(nm)$ time was proposed for finding a minimum weight matching [36]. We treat these weighted matchings in this subsection, and consider some problems of enumerating minimum weight matchings. For the problem of minimum weight perfect matchings, an algorithm was proposed by K. Fukuda and T. Matsui [17]. The algorithm takes $O(nm)$ time per one matching.

To enumerate these matchings, we can utilize the algorithms in the previous subsections. We can transform an enumeration problems of these matchings to a problem solved by those algorithms. They take only $O(n)$ time per a matching, hence we can enumerate these matchings in $O(n)$ time per a matching. Since perfect matchings are maximum matchings, our algorithm can also enumerate all the minimum weight perfect matchings in $O(n)$ time per one.

Firstly, we explain the case of minimum weight matchings. For a matching M, we consider the reduced weight w' and $\hat{w}$ for all edges of G. The weight $w'(e)$ of an edge e is given by

$-w(e)$ if $e \in M$, and $w(e)$ otherwise. We also define the weight $w'(a)$ of an arc a of $D(G, M)$ corresponding to an edge e of G by $w'(e)$. By using w', the matching obtained by exchanging edges along an alternating cycle C is given by $w(M)+w'(C)$ where $w(M)$ and $w'(C)$ are the sum of edge weights in M and C, respectively. Hence a zero weight alternating cycle or path for the weight w' generates the other matching whose weight is equal to M. To enumerate minimum weight matchings, we use these zero weight alternating cycles and paths.

Suppose that M has the minimum weight among all matchings with the same cardinality to M. From this assumption, no directed cycle of $D(G, M)$ has a negative weight. Let V' be the set of uncovered vertices of V_1 and covered vertices of V_2. In the following, we consider a directed graph $\bar{D}(G, M)$ obtained by adding a super source $\bar{s}$ and arcs from $\bar{s}$ to all vertices of V'. An alternating cycle of M corresponds to a directed cycle of $\bar{D}(G, M)$. Since one of the endpoints of any alternating path is in V', an alternating path of M also corresponds to a directed path of $\bar{D}(G, M)$ starting from $\bar{s}$ to a vertex of V'. For any arc a from $\bar{s}$ to the other vertex, we set its reduced weight $w'(a)$ to zero.

For a vertex v, let $d(v)$ be the shortest path length from $\bar{s}$ to v in $\bar{D}(G, M)$ with the reduced weight w'. For an arc $a = (u, v)$, we define a weight $\hat{w}(a)$ by $w'(a) - d(v) + d(u)$. Since d is given by the shortest path length, we have $w'(a) \geq d(v) - d(u)$ hence $\hat{w}(a) \geq 0$. For a directed cycle $C = \{v_1, v_2, ..., v_k, v_1\}$, we have

$$\begin{aligned}
\hat{w}(C) &= \hat{w}((v_1, v_2)) + ... + \hat{w}((v_k, v_1)) \\
&= w'((v_1, v_2)) - d(v_2) + d(v_1) + w'((v_2, v_3)) - d(v_3) + d(v_2) +, ..., \\
&\quad +w'((v_k, v_1)) - d(v_1) + d(v_k) \\
&= w'((v_1, v_2)) + w'((v_2, v_3)) +, ..., +w'((v_k, v_1)).
\end{aligned}$$

Therefore any directed cycle C satisfies $w'(C) = \hat{w}(C)$. Moreover, since $\hat{w}(a) \geq 0$ for any arc a, all the arcs a of a zero weight directed cycle have $\hat{w}(a) = 0$.

Let $\bar{D}'$ be the graph composed of all arcs a of $\bar{D}(G, M)$ satisfying $\hat{w}(a) = 0$. Let $\bar{V}$ be the set of vertices v with $d(v) = 0$. Any directed cycle in $\bar{D}'$ corresponds to a zero weight alternating cycle. Any directed path from $\bar{s}$ to a vertex of $\bar{V} \cap V'$ corresponds to a zero weight alternating path.

Now we suppose that M has the minimum weight among all matchings of G. The symmetric difference between M and the other minimum weight matching is composed of some zero weight alternating cycles and paths. Since their weights for w' are zero, these cycles and paths correspond to directed cycles of $\bar{D}'$ and directed paths of $\bar{D}'$ from $\bar{s}$ to vertices of $\bar{V} \setminus V'$. Let $\bar{M}$ be the set of edges of M not corresponding to arcs of $\bar{D}'$. Let G' be the graph whose edge set is composed of all edges corresponding to arcs of $\bar{D}'$ not adjacent to edges of $\bar{M}$. Let M' be a matching of G' covering all the vertices not in $\bar{V}$. From the above observation, the matching obtained by adding edges of M not in G' to M' is a minimum weight matching of G, since the symmetric difference between M and M' is composed of alternating cycles of G' and alternating paths of G' whose endpoints are in $\bar{V}$. Therefore, by enumerating all the covering matchings of G', we can enumerate all the minimum weight matchings. It takes only $\mathrm{O}(nm)$ preprocessing time and $\mathrm{O}(n)$ time per one minimum weight matching.

Secondly we consider the problem of enumerating minimum weight maximum matchings, which are maximum matchings with the minimum weight among all the maximum matchings. The symmetric difference between a minimum weight maximum matching M and the other minimum weight maximum matching M' is composed of zero weight alternating cycles and paths with even lengths. Hence they are included in G'. On the other hand, the edges of M included in G' compose a maximum matching of G'. If not, then we can generate a matching with a larger cardinality than M. Conversely, the symmetric difference between M and a maximum matching of G' covering all vertices not in $\bar{V}$ is composed of some zero weight alternating cycles and paths. Hence the covering matching has the cardinality and the weight equal to M. Therefore, by enumerating all the maximum cardinality covering matchings in the graph G', we can enumerate all the minimum weight maximum matchings. It also takes O(nm) preprocessing time and O(n) time per one minimum weight maximum matching.

Theorem 18 *All minimum weight matchings and all minimum weight maximum matchings in a bipartite graph can be enumerated in $O(nm)$ preprocessing time and $O(n)$ time per a matching.*

∎

5.5 Enumerating Maximal Matchings in General Graphs

Let $G = (V, E)$ be a general undirected graph with a vertex set V and an edge set E. We denote the numbers of vertices and edges in G by n and m, respectively. The components of E are denoted by $e_1, ..., e_m$. A *matching* M of the graph G is an edge set such that no two edges of M share their endpoints. Edges of a matching are called *matching edges*. For a matching, we say that a vertex of G is *covered* by the matching if it is incident to a matching edge, and otherwise *uncovered*. We call a matching which is contained in no other matching a *maximal matching*.

Matchings are one of the most fundamental objects in the graph theory and the optimization theory. There are a lot of studies on properties of matchings. Many efficient algorithms have been also proposed for a variety of matching problems. For example, an $\mathrm{O}(n^{1/2}m)$ time algorithm is proposed for the problem of finding a matching with the maximum cardinality [27]. On the other hand, no efficient algorithm has been developed for some difficult problems. In these problems, we can hardly find an optimal solution. For example, the problem of finding a minimum cardinality maximal matching is known to be NP-hard. The enumeration is a very naive approach for solving those hard problems. Therefore speeding up of enumeration algorithms is practical for these complicated problems.

For a graph G, a *stable set* in G is a vertex set such that no pair of its vertices are connected by an edge. A matching problem in a graph G is equivalent to a stable set problem in the line graph of G, as each vertex of the line graph corresponds to each edge of G. Hence we can transform a matching problem in G to a stable set problem in its line graph. Thus we may regard a matching problem as a special case of stable set problems. Since matchings satisfy good properties which stable sets do not, many matching problems can be solved easier than stable set problems. For example, we can find a maximum matching of a graph in polynomial time [27, 12], but the maximum stable set problem is known to be belong to the class of NP-hard problems. Of course, no polynomial time algorithm has been proposed for the latter problem.

In this section, we consider the problem of enumerating all maximal matchings in a given undirected general graph G. The problem can be also transformed to the problem of stable sets in the line graph. In 1977, S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa [52] proposed an efficient enumeration algorithm for the problem of maximal stable sets. Their algorithm enumerates all maximal stable sets of G in $\mathrm{O}(nmN)$ time and $\mathrm{O}(m+n)$ space where N is the number of maximal stable sets in G. In 1988, D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou [30] proposed the other algorithm for the problem. The time complexity of their algorithm is same as the algorithm of Tsukiyama et al., but their algorithm outputs all maximal sable sets in the lexicographical order with possibly using of the exponential memory space. Since the line graph of G has m vertices and $\mathrm{O}(nm)$ edges, these algorithms take $\mathrm{O}(nm^2N)$ time for enumerating maximal matchings. Although a simple modification can be considered to these algorithms, they still take $\mathrm{O}(n^2mN)$ time.

Our algorithm in this section can enumerate only maximal matchings, although its time complexity is quite better than theirs. Our algorithm is obtained by transforming the algorithm of Tsukiyama et al. to a reverse search algorithm. By restricting the problem to only matchings, we can utilize some good properties of matchings which Tsukiyama et al. did not use for stable sets. We add some improvements using those good properties to their algorithm, and reduce the time complexity to $O((nH(n))N)$. Here $H(n)$ denotes the time to maintain a heap with at most n elements each of which has a value ranging from 1 through n. We have $H(n) = O(\log n)$ by using a simple heap algorithm. If we use a more sophisticated heap algorithm [29] with some extra memory space, we can reduce $H(n)$ to $O(\log\log n)$ time.

5.5.1 A Modified Reverse Search Algorithm for Maximal Matchings

For constructing enumeration algorithms, we have a good scheme called *reverse search* [3]. Our enumeration algorithm is based on the scheme. Reveres search is a scheme for enumerating all elements of a specified set. It utilizes a parent-child relationship among all elements of the set. The parent-child relationship has to satisfy the following two conditions. The first one is that all elements except an element have their own unique parents. The second one is that they are not proper ancestors of themselves. The graph expression of the relationship, whose vertices correspond to all elements and edges connect children and their parents, forms a tree under these conditions. The tree is called an *enumeration tree*. Reverse search traverses all vertices of the tree in the depth-first search manner, and outputs all elements in the order of visiting. A feature of reverse search is that its memory complexity is not depend on the number of output.

Reverse search does not store the whole enumeration tree, but only the current traversing vertex of the tree. Reverse search lists all children of the vertex in each iteration, and continues traversing by moving to its children in the depth-first search manner. Therefore, to obtain an enumeration algorithm, we are in need to construct only an algorithm for finding all children of the current traversing vertex. An important part of a reverse search algorithm is to list all children in efficiently short time.

Now let us see our reverse search for maximal matchings arising from the method of Tsukiyama et al. For an edge sequence $\{e_1, ..., e_m\}$, we define a sequence composed of subgraphs $\{G_1, ..., G_m\}$ by $G_m = G$ and $G_{i-1} = G_i \setminus e_i$. Our parent-child relationship is defined among all maximal matchings in all subgraphs G_i. Since two maximal matchings of G_i and G_j may have the same edge set, we denote a maximal matching M of a subgraph G_i by (M, G_i) to identify them.

We now introduce the parent-child relationship. The unique maximal matching in G_1 has no parent in our relationship. The parent of the other maximal matching (M, G_i) is defined by (M, G_{i-1}) if M does not contain e_i. Otherwise, it is given by the maximal matching obtained by adding the minimum index edge among edges not adjacent to edges of M to $M \setminus e_i$ one by one until the matching will be maximal. The parents are defined uniquely, and no maximal matching is a proper ancestor of itself. Hence we obtain the enumeration tree whose vertices correspond to all maximal matchings in all subgraphs of the sequence. Leaves of the tree correspond to all maximal matchings in G (see Figure 5.12).

Next we explain how to find all children of a maximal matching (M, G_i). If e_{i+1} is adjacent

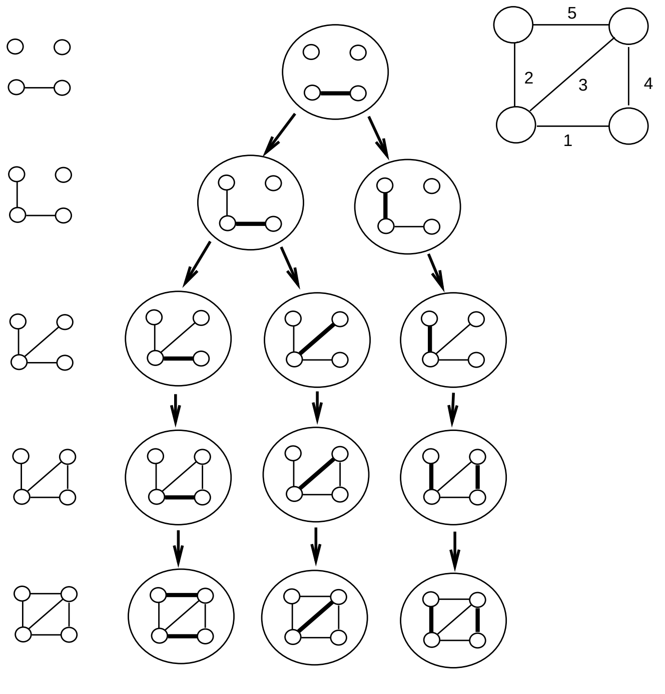

Figure 5.12: A graph G and the enumeration tree with an ordering of edge.

to some edges of M, (M, G_{i+1}) is a child of (M, G_i). In the other case, since $M \cup e_{i+1}$ is a maximal matching of G_{i+1}, $(M \cup e_{i+1}, G_{i+1})$ is a child of (M, G_i). We call this child *type 1*. Any maximal matching (M, G_i) has exactly one type 1 child if $i < m$. A type 1 child can be obtained by adding e_{i+1} if e_{i+1} is adjacent to no edge of M. It can be done in O(1) time.

Next we consider the other kind of children (M', G_{i+1}) whose parent (M, G_i) is obtained by removing e_i and adding some edges to M'. This kind of child is obtained from (M, G_i) by adding e_{i+1} to M, and deleting all edges adjacent to e_{i+1} from M. Hence the parent (M, G_i) has at most one such child. We call this kind of child a *type 2 child*. For a maximal matching (M, G_i), let M' be a matching obtained by adding e_{i+1} and deleting all edges adjacent to e_{i+1} from M. If M' is maximal in G_{i+1} and the parent of (M', G_{i+1}) is (M, G_i), (M', G_{i+1}) is a type 2 child of (M, G_i). M' is maximal in G_{i+1} if and only if no edge connects an uncovered vertex and an endpoint of edges in $M \setminus M'$. (M, G_{i+1}) is the parent of (M', G_{i+1}) if and only if for each endpoint of e_{i+1}, an edge of $M \setminus M'$ has the minimum index among edges connecting uncovered vertices and the endpoint. Thus, by checking these conditions, we can check whether a maximal matching has a type 2 child or not. To check the conditions in short time, we have a heap on each vertex which gives the minimum index edge among edges connecting the vertex and uncovered vertices. By using the heaps, we can check the conditions in $O(H(n))$ time.

Next we see updating time of heaps when we generate a child (M', G_{i+1}) from (M, G_i). The update of heaps are done by inserting or deleting some edges. Since the symmetric difference S between M' and M has a constant number of edges, the number of these edges is constant. The total updating time for all heaps are $O(nH(n))$. By using the number

d which is the number of edges adjacent to edges of S, the updating time is bounded by $O(dH(n))$. Here we write our reverse search algorithm.

```
ALGORITHM: ENUM_MAXIMAL_MATCHINGS ($M$, $G_i$)
Step 1: Generate the type 1 child of $(M, G_i)$, update heaps and call
        ENUM_MAXIMAL_MATCHINGS recursively.
Step 2: Check the existence of the type 2 child by generating a matching $M'$.
Step 3: If $(M', G_{i+1})$ is a type 2 child, update heaps and call
        ENUM_MAXIMAL_MATCHINGS recursively.
```

Let us consider the time complexity of the algorithm by using the enumeration tree. If an internal vertex of the enumeration tree has a type 2 child, the degree of the vertex is three. Hence the number of internal vertices with type 2 children is at most N, and the sum of time spent for generating type 2 children and updating heaps for them is bounded by $O(nH(n)N)$.

To bound the time for generating the type 1 children and checking the existence of type 2 children, let us delete all edges of the enumeration tree connecting type 2 children and their parents. Their removal is composed of some paths. We call these paths *type 1 child paths*. Since any internal vertex has type 1 child, one of the endpoints of these type 1 paths are leaves. The number of type 1 child paths is at most N. The computation time to generate the type 1 child on a vertex is given by the sum of degrees of both endpoints of the edge in the symmetric difference between the child and its parent. Hence, the total computation time to generate type 1 children on all vertices in a type 1 child path is bounded by the sum of degrees over all edges of a maximal matching. Therefore it is bounded by $O((m+n)H(n)N)$.

By using some heap algorithms, the update of a heap is done in $O(\log n)$ time. Some other heap algorithms with extra memory space can reduce the time complexity to $O(\log \log n)$ time [29]. Their heap deals only values from 1 through n, and reduces the time complexity by using n^{1+p} space for an element, where p is a small positive constant.

In the next subsection we will reduce the time complexity by introducing an ordering of edges.

5.5.2 Bounding the Time Complexity by a Better Bound

In the previous subsection, we described an algorithm but we does not achieve the time complexity of our results. In this subsection, we prove the following property to bound the time complexity.

Lemma 30 *With a certain ordering of edges, the sum of computation time to find and generate type 1 children is bounded by $O(nH(n)N)$.*

Proof : To obtain such an order, we partition the graph G into some blocks, which are disjoint edge subsets $\{B_1, ..., B_k\}$ of E by the following algorithm.

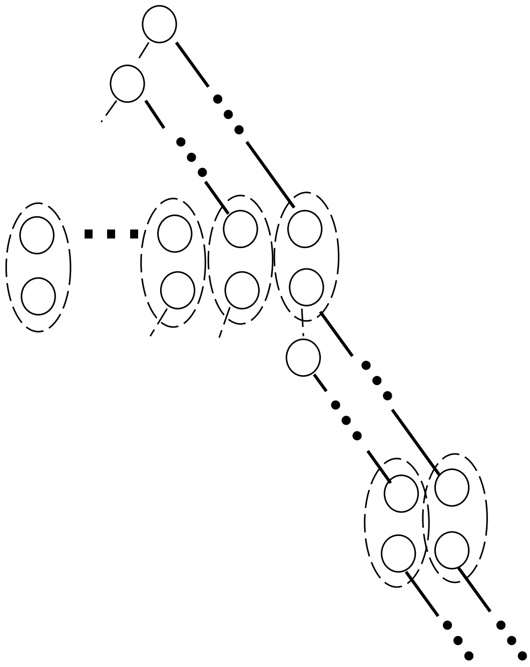

Figure 5.13: Splitting the enumeration tree. Dotted lines are the edges connecting type 2 children and their parents. Dotted circles include two vertices which are generated by splitting the vertices (M, G_{K_i}).

```
ALGORITHM: PARTITION_TO_BLOCKS ($G$)
Step 1: Choose a pair of edges $b$ and $b'$ adjacent to each other.
Step 2: If no such a pair exists, set $B_1$ to all the edges of $G$, and set $i$ to 1. Go to Step 8.
Step 3: Remove $b$ and $b'$, and all edges adjacent to $b$ or $b'$ from $G$.
Step 4: $i :=$ PARTITION_TO_BLOCKS ($G$).
Step 5: Set $b_i$ to $b$ and $b'_i$ to $b'$.
Step 6: Set $B_i$ to all the edges which were removed in Step 3.
Step 7: Add all edges of $B_i$ to $G$.
Step 8: Return $i$.
```

Since any edge of G is deleted only once by the algorithm, this algorithm spends $O(m+n)$ time to partition the graph to blocks. For any i, we have that $|B_i| < 3n$ and b_i and b'_i are adjacent to no edge of B_j for any $j < i$. By using this partition, we put indices for all edges of G such that the index of an edge $e \in B_i$ is smaller than any edge $e' \in B_{i'}$ if $i < i'$. By using the ordering of these indices, we state the following properties. Let K_i be $\sum_{j=1}^{i-1} |B_j|$. The indices of edges in B_i are ranged from $K_{i-1}+1$ through K_i.

Lemma 31 *(1) Let $P = \{(M_p, G_p), (M_{p+1}, G_{p+1}), ..., (M_q, G_q)\}$ be a subpath of a type 1 child path with $K_i < p < q = K_{i+1}$. The sum of computation time spent by vertices in P for generating type 1 children is $O(nH(n))$. (2) Any vertex (M_{K_i}, G_{K_i}) has at least two descendants $(M_{K_{i+1}}, G_{K_{i+1}})$ and $(M'_{K_{i+1}}, G_{K_{i+1}})$ if $i > 1$.*

Proof : (1) To generate type 1 child (M_j, G_j), the algorithm takes $O(H(n))$ time if e_j is not included in M_j, and $O(dH(n))$ time otherwise. Here d is the number of edges adjacent to e_j in G_j. Therefore the total time spent by the algorithm to generate type 1 children $(M_{p+1}, G_{p+1}), ..., (M_q, G_q)$ is $O((|B_i| + \hat{d})H(n))$ where $\hat{d}$ is the sum of degrees in G_q over all endpoints of edges in $M_q \setminus M_p$. If $i = 1$, then B_i forms a matching, hence $\hat{d} \leq n$. Otherwise, since all edges of B_i are incident to at least one endpoint of b_i or b'_i, we have $|M_q \setminus M_p| \leq 3$. Note that $M_q \setminus M_p \subseteq B_i$. Hence we obtain $O(\hat{d}H(n)) = O(nH(n))$. (2) There exist at least two maximal matching $M_{K_{i+1}}$ and $M'_{K_{i+1}}$ including M_{K_i} and b_i, and M_{K_i} and b'_i. Both $(M_{K_{i+1}}, G_{K_{i+1}})$ and $(M'_{K_{i+1}}, G_{K_{i+1}})$ are descendants of (M_{K_i}, G_{K_i}). ▮

By using these two properties, we bound the time complexity. Let $\mathcal{P}$ be the path set obtained by splitting each type 1 child path on the vertices (M_{K_j}, G_{K_j}) for all possible j. From the above lemmas, the sum of computation time over all vertices of a path of $\mathcal{P}$ is $O(nH(n))$. Let us see that there are at most $2N$ paths in $\mathcal{P}$. To see it, we show an assigning rule of type 2 children to all paths in $\mathcal{P}$ such that any type 2 child is assigned to at most two paths. Note that the enumeration tree includes just $N-1$ type 2 children. Let the head of a path in $\mathcal{P}$ be the nearest endpoint of the path to the root. For each path whose head is a type 2 child, we assign its head to the path. For the other path, its head is (M_{K_i}, G_{K_i}) for some i. By the above lemma 4, at least one vertex of the path has a type 2 child. We assign one of them to the path. By this, all paths are assigned type 2 children, and any type 2 child is assigned to at most two paths. Thus we have $|\mathcal{P}| \leq 2N$. Therefore the total time spent by the algorithm is $O(nH(n)N)$. ▮

Now we describe the preprocessing phase of our enumeration algorithm.

ALGORITHM: ENUM_MAXIMAL_MATCHINGS_2 (G)
Step 1: Call PARTITION_TO_BLOCKS (G, T).
Step 2: Put indices to all edges of each B_i.
Step 3: Call ENUM_MAXIMAL_MATCHINGS (e_1, G_1).

Theorem 19 *The algorithm* ENUM_MAXIMAL_MATCHINGS_2 *enumerates all maximal matchings of a general graph in $O((n \log n)N)$ time and $O(n+m)$ space, or $O((n \log\log n)N)$ time and $O(n^{1+p} + m)$ space where p is a small constant with $p > 0$. Here N is the number of maximal matchings in the graph.*

Proof : We have seen that the algorithm runs correctly, and takes $O(nH(n))$ time per a maximal matching. Thus, we see the computation time on preprocessing. The preprocessing takes $O(m + n)$ time. From the above lemmas, we can see that any graph contains at least $2^{m/3n-1}$ maximal matchings, hence we have $O(m + n) = O((n \log\log n)N)$. ▮

Bibliography

[1] R. K. Ahuja, T. L. Magnanti and J. B. Orlin, "Network Flows: Theory, Algorithms, and Applications," Prentice-Hall International Editions (1993).

[2] E. A. Akkoyunlu, "The Enumeration of Maximal Cliques of Large Graphs," SIAM J. Comp. **2**, pp.1-6 (1973).

[3] D. Avis and K. Fukuda, "A Pivoting Algorithm for Convex Hulls and Vertex Enumeration of Arrangements and Polyhedra," Discrete Comp. Geom. **8**, pp.295-313, (1992).

[4] D. Avis and K. Fukuda, "Reverse Search for Enumeration," Discrete Appl. Math. **65**, pp.21-46 (1996).

[5] J. A. Bondy and U. S. R. Murty, "Graph Theory with Applications," North-Holland (1976).

[6] R. M. Brady, "Optimization Strategies Gleaned from Biological Evolution," Nature **317**, pp.804-806 (1985).

[7] C. R. Chegireddy and H. W. Hamacher, "Algorithms for Finding K-best Perfect Matchings," Discrete Appl. Math. **18**, pp.155-165 (1987).

[8] R. Cole and J. Hopcroft, "On Edge Coloring Bipartite Graphs," SIAM J. Comp. **11**, pp.540-546 (1982).

[9] T. H. Cormen, C. E. Leiserson and R. L. Rivest, "Introduction To Algorithms," The MIT Press.

[10] M. A. H. Dempster, "Two Algorithms for the Time-Table Problem," Combinatorial Mathematics and its Applications(ed. D.J.A.Welsh). Academic Press New York, pp.63-85 (1971).

[11] M. T. Dickerson, R. L. S. Drysdale and J. R. Sack, "Simple Algorithms for Enumerating Interpoint Distances and Finding k Nearest Neighbors," J. CGA. **2**, pp.221-239 (1992).

[12] J. Edmonds, "Paths, Trees and Flowers," Canadian J. Math. **17**, pp.1-13 (1965).

[13] D. Eppstein, "Finding the k Shortest Paths," 35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, 20-22 November, pp.154-165 (1994).

[14] D. Eppstein, “Finding the k Smallest Spanning Trees,” SWAT ’90 Lec. Notes Comp. Sci. **447**, Springer Verlag, pp.38-47 (1990).

[15] D. Eppstein, Z. Galil, G. F. Italiano and A. Nissenzweig, “Sparsification - A Technique for Speeding up Dynamic Graph Algorithms, ” FOCS **33**, pp.60-69 (1992).

[16] G. N. Frederickson, “Data Structures for On-Line Updating of Minimum Spanning Trees With Applications,” SIAM J. Comp. **14**, No. 4, pp.781-798 (1985).

[17] K. Fukuda and T. Matsui, “Finding All the Minimum Cost Perfect Matchings in Bipartite Graphs,” Networks **22**, pp.461-468 (1992).

[18] K. Fukuda and T. Matsui, “Finding All the Perfect Matchings in Bipartite Graphs,” Appl. Math. Lett. **7**, No. 1, pp.15-18 (1994).

[19] K. Fukuda and M. Namiki, “Finding All Common Bases in Two Matroids,” Working Paper, Dept. of Social and International Relations, University of Tokyo, Tokyo, 32 (1993).

[20] K. Fukuda and V. Rosta, “Combinatorial Face Enumeration in Convex Polytopes,” (1993).

[21] K. Fukuda, S. Saito and A. Tamura, “Combinatorial Face Enumeration in Arrangements and Oriented Matoroids,” Discrete Appl. Math. **31**, pp.141-149 (1991).

[22] H. N. Gabow, “Using Euler Partitions to Edge Color Bipartite Multigraphs,” Information J. Comp. and Info. Sci. **5**, pp.345-355 (1976).

[23] H. N. Gabow and E. W. Myers, “Finding All Spanning Trees of Directed and Undirected Graphs,” SIAM J. Comp. **7**, pp.280-287 (1978).

[24] H. N. Gabow and O. Kariv, “Algorithms for Edge Coloring Bipartite Graphs and Multigraphs,” SIAM J. Comp. **11**, pp.117-129 (1982).

[25] T. Gonzalez and S. Sahni, “Open Shop Scheduling to Minimize Finish Time,” J. ACM. **23**, pp.665-679 (1976).

[26] H. W. Hamacher, J. C. Picard and M. Queyranne, “On Finding the K best Cuts in a Network,” Operations Research Lett. **2**, pp.303-305 (1984).

[27] J. E. Hopcroft and R. M. Karp, “An $n^{5/2}$ Algorithm for Maximum Matching in Bipartite Graphs,” SIAM J. Comp. **2**, pp.225-231 (1973).

[28] D. B. Johnson, “Finding All the Elementary Circuits of a Directed Graph,” SIAM J. Comp. **4**, pp.77-84 (1975).

[29] D. B. Johnson, “A Priority Queue in Which Initialization and Queue Operations Take O(log log D) Time,” Math. Systems Theory **15**, pp.295-309 (1982).

[30] D. S. Johnson, M. Yannakakis, and C. H. Papadimitriou, "On Generating All Maximal Independent Sets," Info. Processing Lett. **27**, pp.119-123 (1988).

[31] D. König, "Über Graphen und ihre Anwendung auf Determinantentheorie und Mengenlhere," Math. Ann. **77**, pp.453-465 (1916).

[32] H. N. Kapoor and H. Ramesh, "Algorithms for Generating All Spanning Trees of Undirected, Directed and Weighted Graphs," Lec. Notes Comp. Sci. **519**, Springer-Verlag, pp.461-472 (1992).

[33] N. Katou, T. Ibaraki and H. Mine, "An Algorithm for Finding K Minimum Spanning Trees," SIAM J. Comp. **10**, pp.247-255 (1981).

[34] T. Kashiwabara, S. Masuda, K. Nakajima and T. Fujisawa, "Generation of Maximum Independent Sets of a Bipartite Graph and Maximum Cliques of a Circular-Arc Graph," J. Algo. **13**, pp.161-174 (1992).

[35] D. B. Khang and O. Fujiwara, "A New Algorithm to Find All Vertices of a Polytope," Operations Research Lett. **8**, pp.261-264 (1989).

[36] Lovasz and Plummer, "Matching Theory", Ann. of Discrete Math. **29**, North-Holland (1986).

[37] T. Matsui, "A New Algorithms for Finding All the Spanning Trees in Undirected Graphs," Research Report, Dept. of Math. Engineering and Info. Physics, University of Tokyo, Tokyo (1993).

[38] N. Meggido, A. Tamir, E. Zemel and R. Chandrasekaran, "An $O(n \log^2 n)$ Algorithm for the Kth Longest Path in a Tree with Applications to Location Problems," SIAM J. Comp. **10**, pp.328-337 (1981).

[39] E. F. Moore, "The Shortest Path Through a Maze," Proceedings of the International Symposium on the Theory of Switching, pp.285-292 (1959).

[40] H. Mühlenbein, M. Gorges-Schleuter and O. Krämer, "Evolution Algorithms in Combinatorial Optimization," Networks **5**, 237-252 (1975).

[41] K. G. Murty, "An Algorithm for Ranking All the Assignments in Order of Increasing Cost," Operations Research **16**, pp.682-687 (1968).

[42] G. L. Nemhauser and G. Sigismondi, "A Strong Cutting Plane Branch-and-Bound Algorithm for Node Packing," J. Oprational Research Society (1992).

[43] J. G. Oxley, "Matroid Theory," Oxford Science Publications (1992).

[44] R. C. Read and R. E. Tarjan, "Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees," Networks **5**, 237-252 (1975).

[45] D. Rotem and J. Urrustia, “Finding Maximum Cliques in Circle Graphs,” Networks **11**, pp.267-278 (1981).

[46] K. Sekine and H. Imai, “A Unified Approach via BDD to the Network Reliability and Path Numbers,” Technical Report 95-9, Dept. of Info. Sci., University of Tokyo (1995).

[47] D. D. Sleator and R. E. Tarjan, “A Data Structure for Dynamic Trees,” J. Comp. Sys. Sci. **26**, pp.362-391 (1983).

[48] A. Shioura and A. Tamura, “Efficiently Scanning All Spanning Trees of an Undirected Graph,” J. Operation Research Society Japan (1993).

[49] A. Shioura, A. Tamura and T. Uno, “An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs, ” SIAM J. Comp. **26**, No. 3, pp.678-692 (1997).

[50] R. E. Tarjan, “Depth-First Search and Linear Graph Algorithm,” SIAM J. Comp. **1**, pp.146-169 (1972).

[51] S. Tsukiyama, I. Shirakawa and H. Ozaki, “An Algorithm to Enumerate All Cutsets of a Graph in Linear Time per Cutset,” J. ACM. **27**, pp.619-632 (1980).

[52] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, “A New Algorithm for Generating All the Maximum Independent Sets,” SIAM J. Comp. **6**, pp.505-517 (1977).

[53] T. Uno, “An Algorithm for Enumerating All Directed Spanning Trees in a Directed Graph,” Lect. Notes in Comp. Sci. **1178**, Springer-Verlag, Algorithms and Computation, pp.166-173 (1996).

[54] Y. Matsui and T. Uno, “A Simple and Fast Algorithm for Enumerating all Edge Colorings of a Bipartite Graph,” Research Report, Dept. Math. and Comp., Tokyo Institute of Technology, Japan (1996).

[55] T. Uno, “Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs,” Lecture Note in Computer Science **1350**, Springer-Verlag, Algorithms and Computation (Proceeding of ISAAC97), pp.92-101 (1997).

[56] T. Uno, “A Fast Algorithm for Enumeration of Maximal Matchings in General Graphs,” Research Report, Dept. Math. and Comp., Tokyo Institute of Technology, Japan (1997).

[57] T. Uno and M. Yagiura “Fast Algorithms to Enumerate All Common Intervals of Two Permutations,” Submitted to Algorithmica (1996).

[58] M. Yagiura, H. Nagamochi and T. Ibaraki, “Two Comments on the Subtour Exchange Crossover Operator,” Journal of Japanese Society for Artificial Intelligence **10**, pp.464-467 (1995) (in Japanese).

[59] Y. Yoshida, Matsui and T. Matsui, “Finding All the Edge Colorings in Bipartite Graphs, ” T.IEE. Japan 114-C **4**, pp.444-449 (1994) (in Japanese).

[60] Y. Matsui and T. Matsui, "Enumeration Algorithm for the Edge Coloring Problem on Bipartite Graphs," Lecture Notes in Computer Science, Springer-Verlag **1120**, pp.18-26 (1996).

[61] D. Werra, "On Some Combinatorial Problems Arising in Scheduling," INFOR. **8**, pp.165-175 (1970).