Vol.2014-AL-147 No.17
2014/3/4

Toward Constant Time Enumeration

Takeaki Uno[1,a)]

Abstract: In this paper, we propose a new amortized analysis for enumeration algorithms. We amortize the computation time of an iteration by charging to all its descendant iterations. We clarify sufficient conditions such that if an enumeration algorithm satisfies the condition, the amortized analysis works. By the amortization, we show that many elimination orderings, matchings in a graph, connected vertex induced subgraphs in a graph, and spanning trees can be enumerated in $O(1)$ time for each solution with simple algorithms and proofs.

Keywords: enumeration, amortization, matching, spanning tree, connected subgraph

1. Introduction

Enumeration algorithms have been one of the most recent hot topics in theoretical computer science. The number of papers on these has been increasing in journals and conferences. One of the reasons for this is the increase in applications. Thanks to the recent increase in computational power, we can enumerate all solutions in sufficiently practical time length, even when there are a large number of solutions. On the other hand, as the objectives of information processing have become complicated in recent research areas and applications, so that they can not be represented easily in well-written mathematical forms. A simple and beneficial approach in such cases is enumeration of candidate solutions. Many methods especially in data mining and data engineering have followed this approach. A typical example is pattern mining that finds all structures of the given database that have specified properties, such as appearing in the database frequently, or being similar to no other parts, etc.

Enumeration algorithms are often efficient in practice. For example, frequent itemset mining algorithms[*1] usually only take a constant time for each solution, even if they receive a large database as input[6]. As the theoretical bound for each solution is linear in the size of the input database, there is a large gap between theory and practice. The reason algorithms are fast arises from their recursive structures, which we call bottom-expanded.

Iterations of enumeration algorithms usually generate several, or many, recursive calls; thus, the number of iterations exponentially increases as one goes deep into the recursions. On the other hand, iterations usually give subproblems to their children, which are smaller than their input problems. Thus, we can observe that, near by the root of the recursion, there are small number of iterations that spend a long time near by the bottom of the recursions, and there are many iterations that spend very short time. This implies that the amortized computation time per iteration, or even per solution, is short. This mechanism is what we call bottom-expanded. As we can see this mechanism in many kinds of pattern mining algorithms, and also classical enumeration algorithms, they are efficient in practice.

This mechanism motivated us to develop good amortized analysis of enumeration algorithms. However, amortization is usually not easy, since it is hard to globally estimate the number of iterations and computation time. Thus, in many existing studies, the computation time is amortized between a parent and its children, and sometimes its grandchildren[2], [3], [4], [5], [7], [13]. This local structure is easier to analyze than the global structure. Extensions of this idea to more global structures are non-trivial. For example, if we want to amortize between iterations in different subtrees of the recursion, we have to understand the relation and the correspondence between all iterations in the different subtrees. This is usually a difficult task.

In this paper, we propose a new way of carrying out amortized analysis of the time complexity of enumeration algorithms, and propose new algorithms for enumeration of matchings, elimination orderings, and connected vertex induced subgraphs. We also show that the amortized analysis can prove the existing complexity results in very simple ways, for the enumerations of spanning trees perfect elimination orderings,and perfect sequences, while the existing algorithms often need sophisticated algorithms or data structures. We can say that our results can show deeper properties of enumeration algorithms that can never be shown by existing methods, and fill the gap between theory and

[1] National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda, Tokyo 101–8430, Japan
[a)] uno@nii.ac.jp

[*1] The purpose of frequent itemset mining is to find all sets of items that are included in at least σ records for a given database composed of records that are subsets of the itemset and a threshold, σ.

application; usually the computation time of implementation of enumeration algorithms usually take very short time compared to theoretical bounds in real-world problems. Our analysis can clarify the potential efficiency of enumeration algorithms that cannot be seen directly.

The amortization of an iteration basically uses all its descendants. For each iteration, we push out its computation time to its children so that the assigned time is proportional to their computation time. By applying this push-out from the root of the recursion to deeper levels, the long computation time near the root is diffused to deeper levels, that have shorter times on average. Since it is very hard to capture the structure of the recursion, we show some conditions that are sufficient to have a good upper bound obtained by the amortization. As the condition is given to the relation between each iteration and its children, it is usually easier than inter-subtree analysis. Thus, it has a large degree of utility and is easy to apply.

We demonstrate that several enumeration algorithms can be well analyzed by using amortization; we have efficient amortized time complexity even though the algorithm is quite simple. Several elimination orderings including perfect elimination orderings, perfect sequences can be enumerated in constant time for each ordering. Indeed algorithms for these problems have already been proposed that have the same time complexity[2], [8], our algorithms and proof are quite simple. We also propose enumeration algorithms for enumerating matchings, connected vertex induced subgraphs, and spanning trees in a graph. They take $O(1)$ time for each solution.

The organization of this paper is as follows. The next section introduces some notations, especially on the recursion of enumeration algorithms. Section 3 describes sufficient conditions and prove that our amortization works under these conditions. We present cases of elimination orderings, matchings, connected vertex induced subgraphs, and spanning trees in Section 4, 5, 6, and 7. We conclude the paper in Section 8.

2. Preliminaries

Let $\mathcal{A}$ be an enumeration algorithm. Suppose that $\mathcal{A}$ is a recursive type algorithm, i.e., a subroutine that recursively calls itself several times (or none). Thus, the recursion structure of the algorithm forms a tree. We call the subroutine, or the execution of the subroutine by an *iteration*. Note that an iteration does not include the computation done in the subroutines recursively called by the iteration. When the algorithm is composed of several kinds of subroutines and operations, and thus the recursion is a nest of several kind of subroutines, we consider a set of them as an iteration.

When an iteration X recursively calls an iteration Y, X is called the *parent* of Y, and Y is called a *child* of X. *Root iteration* is that with no parent. For non-root iterations, their parent is unique, and is denoted by $P(X)$. The set of the children of X is denoted by $C(X)$. The parent-child relation between iterations forms a tree structure called a *recursion tree*. An iteration is called a *leaf iteration* if it has no child, and an *inner iteration* otherwise.

Hereafter we suppose that $\mathcal{A}$ is executed by giving an instance, and thus $\mathcal{A}$ performs several recursive calls, and outputs several solutions. We denote the number of solutions by N, and let $\mathcal{I}$ be the set of iterations executed by $\mathcal{A}$. Here we suppose that an iteration does not include the operations executed in its recursive calls, thus no iteration is included in another iteration, and thus the iterations are disjoint. We remark that the term iteration means both a basic unit of an algorithm, and a set of executed operations that belong to $\mathcal{I}$.

For an iteration $X \in \mathcal{I}$, an upper bound of the execution time (the number of operations) of X is denoted by $T(X)$. Here we exclude the computation for the output process from the computation time. For example, when $T(X) = O(n^2)$, $T(X)$ is written as cn^2 for some constant c. T^* is the maximum $T(X)$ among all leaf iterations X. Here, T^* can be either constant, or a polynomial of the input size n. If X is an inner iteration, the sum of the computation time of the children of X is written as $\overline{T}(X)$, i.e., $\overline{T}(X) = \sum_{\text{child } Y \text{ of } X} T(Y)$.

We denote a graph of vertex V and edge set E by $G = (V, E)$. In this paper we assume that for computation a graph is stored in memory in a style of adjacency lists, i.e., for each vertex u, the set of vertices adjacent to u is stored by an array or a list. For a vertex subset U of a graph $G = (V, E)$, the *induced subgraph* of U is the graph whose vertex set is U, and whose edge set contains the edges of E connecting two vertices of U.

An edge f is said to be *parallel* to e if e and f have the same endpoints, and be *series* to e if e is a bridge in $G \backslash f$ and not in G. A graph is called *simple* if it has no edge parallel to another, and a multi-graph otherwise. The relations series and parallel are both symmetric, thus f is series/parallel to e if and only if e is series/parallel to f. The set of an edge e and edges series to e form a cycle like global structure of the graph.

For an edge (u, v) of G, the graph *contracted* by (u, v), denoted by $G/(u, v)$, is the graph obtained by unifying the vertices u and v into one. Note that $G/(u, v)$ of a simple graph G can have parallel edges but never if (u, v) is a bridge. For an edge set $F = \{e_1, \ldots, e_k\}$, G/F denotes the graph $G/e_1/e_2/\cdots/e_k$. Let $G//(u, v)$ be the graph obtained from $G/(u, v)$ by removing all selfloops and parallel edges generated by the contraction, i.e., the graph obtained from $G \setminus v$ by adding an edge (u, w) for each vertex w adjacent to v and not to u.

A graph is called a *tree* if the removal of any its edge results a disconnected graph. For a graph $G = (V, E)$, a *subtree* of G is its subgraph that is a tree.

3. Push Out Amortization

The size of input of each iteration for a recursive algorithm usually decreases as the depth of the recursion. Thus, iterations near the root iteration usually take a relatively

long time, and iterations near leaf iterations usually take a relatively short time. Motivated by this observation, we amortize the computation time by moving the computation time of each iteration to its children. We carry out this move from the top to the bottom, so that the computation time of ancestors is recursively diffused to their descendants. When we can obtain a short amortized computation time in this way, iterations with long computation times have many descendants proportional to their computation time; the average computation time per iteration will be long with only a few descendants. However, it is not easy to prove that any inner iteration has sufficiently many descendants. Instead of that, we use some local conditions, related to a parent and children. Suppose that we have two constants $\alpha > 1$ and $\beta \geq 0$. Let $c(X)$ be the sum of $|C(X)| + 1$ and the number of solutions output in X.

Push Out Condition (PO condition): $\overline{T}(X) \geq \alpha T(X) - \beta c(X)T^*$ where X is an iteration.

This condition intuitively implies that after assigning the computation time of $\beta c(X)T^*$ to children, the total computation time of one level of recursion increases as the depth; thus, each iteration has many descendants. Under this condition, we have the following theorem.

Theorem 1 If any inner iteration of an enumeration algorithm satisfies the PO condition, the amortized computation time of an iteration is $O(T^*)$.

Proof: To prove the lemma, we charge the computation time. We neither move the operations nor modify the algorithm, but just charge the computation time; the computation time can be considered as tokens, and we move the tokens so that each iteration has a small number of tokens. We charge the computation time from an iteration to its children, i.e., from the top of the recursion tree to the bottom. Thus, an iteration receives computation time from its parent. We charge (push out) its computation time and that received from its parent to its children. The computation time is charged to the children, in proportion of their individual computation time, using the following rule.

Push out rule: Suppose that iteration X receives a computation time of $S(X)$ from its parent, thus X has computation time of $S(X) + T(X)$ in total. Then, we charge $\frac{\beta}{\alpha-1}c(X)T^*$ of the computation time to X, and charge (push out) the remaining computation time of quantity $S(X) + T(X) - \frac{\beta}{\alpha-1}c(X)T^*$ to its children. Each child Z receives computation time proportional to $T(Z)$ due to the push out rule, i.e., Z receives computation time of

$$(S(X) + T(X) - \frac{\beta}{\alpha - 1}c(X)T^*)\frac{T(Z)}{\overline{T}(X)}.$$

According to this rule, we charge the computation time from the root iteration to leaf iterations, so that each inner iteration has $O(c(X)T^*)$ computation time. We prove the statement of the lemma by showing that each leaf iteration receives computation time of $O(T^*)$. To show that, we state the following claim.

Claim: if we charge computation time in the manner of the push out rule, each iteration X receives computation time of at most $T(X)/(\alpha - 1)$ from its parent, i.e., $S(X) \leq T(X)/(\alpha - 1)$

The root iteration satisfies this condition. Suppose that an iteration X satisfies this condition. Then, for any child Z of X, Z receives computation time of

$$\begin{aligned}
&(S(X) + T(X) - \frac{\beta}{\alpha - 1}c(X)T^*)\frac{T(Z)}{\overline{T}(X)} \\
\leq\ &(T(X)/(\alpha - 1) + T(X) - \frac{\beta}{\alpha - 1}c(X)T^*)\frac{T(Z)}{\overline{T}(X)} \\
=\ &\frac{\alpha T(X) - \beta c(X)T^*}{\alpha - 1} \times \frac{T(Z)}{\overline{T}(X)} \\
=\ &\frac{\alpha T(X) - \beta c(X)T^*}{\overline{T}(X)} \times \frac{T(Z)}{\alpha - 1}.
\end{aligned}$$

Since the PO condition is satisfied, we have $\overline{T}(X) \geq \alpha T(X) - \beta c(X)T^*$. Thus,

$$\frac{\alpha T(X) - \beta c(X)T^*}{\overline{T}(X)}\frac{T(Z)}{\alpha - 1} \leq \frac{T(Z)}{\alpha - 1}.$$

By induction, any iteration satisfies the condition in the claim.

We can see that any iteration has been charged time of at most $O(\beta c(X)T^*/(\alpha - 1))$. The summation of the number of children over all vertices in a rooted tree is equal to the number of edges in the tree. Therefore, we have the statement. ∎

Note that PO condition does not require for the iterations to have at least two children.

4. Enumeration of Elimination Ordering

Let $\mathcal{L}$ be a class of structures such as sets, graphs, sequences. Suppose that any structure $Z \in \mathcal{L}$ consists of a set of elements called an *element set*, that is denoted by $V(Z)$. Examples of element sets are the vertex set of a graph, the edge set of a graph, the cells of a matrix, and the letters of a string. The empty structure $\perp$ is the unique structure that has $V(\perp) = \emptyset$, and hereafter we consider only $\mathcal{L}$ including the empty structure. For each $Z \in \mathcal{L}, Z \neq \perp$, we define the set of *removable elements* $R(Z)$, such that for each removable element $e \in R(Z)$, the removal of e from Z results a structure $Z' \in \mathcal{L}, V(Z') = V(Z) \setminus \{e\}$. We denote the removal of e from Z by $Z \setminus e$, and we assume that no two different structures can be generated by the removal of e. By using removable elements, we define *elimination orderings*. An elimination ordering is an ordering $(z_1, \ldots, z_n)$ of elements in $V(Z)$ iteratively removed from Z until Z is $\perp$, i.e., any z_i is removable in the structure Z_i that is obtained by repeatedly removing z_1 to z_{i-1} from Z.

Example (a): connected component

$\mathcal{L}$ is a set of connected graphs, and $V(Z)$ is the vertex set

of graph Z. For a graph Z, $R(Z)$ is the set of vertices z such that the removal of z also results in a connected graph (non-cut vertex, thus $R(Z)$ is non-empty). An elimination ordering is a way of removing vertices one by one while keeping the graph connected.

Example (b): perfect elimination ordering[2]
For a graph, a vertex is called *simplicial* if the vertices adjacent to it form a clique. A graph is a *chordal graph* if it has no induced cycle (chordless cycle) of length more than three, and we define $\mathcal{L}$ by the set of chordal graphs. $V(Z)$ is the vertex set of Z. We define $R(Z)$ by the set of simplicial vertices of graph Z. It is known that the removal of a simplicial vertex from a chordal graph results a chordal graph, and any chordal graph has at least one simplicial vertex, thus $R(Z)$ is well defined and non-empty. An elimination ordering, called *perfect elimination ordering*, is a vertex sequence obtained by repeatedly removing simplicial vertices until the graph is empty. Note that a graph is chordal if and only if it has a perfect elimination ordering.

This section deals with the problem of enumerating elimination orderings of a given structure class. A simple algorithm for enumerating elimination orderings can be described as follows.

ALGORITHM Enum_Elim_Ordering (Z, S)
1. **if** $|V(Z)| = 1$, **output** $S + z$ where $V(Z) = \{z\}$; **return**
2. **for** each element $z \in V(Z)$ **do**
3. **if** $z \in R(Z)$, **call** Enum_Elim_Ordering $(Z \setminus z, S + z)$
4. **end for**

We have the following theorem by giving two conditions to the algorithm. Suppose that we are given a structure Z in a class $\mathcal{L}$ and removable element set R for element set $V(Z)$. We also suppose that for any $z \in V(Z)$, we can check $z \in R(Z)$ or not in $O(p(|V(Z)|)q(n))$ time, where $p(|V(Z)|)$ is a polynomial of $|V(Z)|$, and $q(n)$ is a function where n is an invariant of the input structure, such as the number of edges in the original graph.

Theorem 2 Elimination orderings of a class $\mathcal{L}$ can be enumerated in $O(q(n))$ time for each, if $|R(Z)| \geq 2$ holds for each $Z \in \mathcal{L}$ such that $|V(Z)|$ is larger than a constant number c.

Proof: We first bound the computation time for the processes not for output, that is, step 1 of Enum_Elim_Ordering. After that, we estimate the bound for the output process. Note that the computation time of an iteration is bounded by $O(|V(Z)|p(|V(Z)|)q(n))$.

First, we choose two constants $\delta > c$ and $\alpha > 1$ such that $p(i)i/2(i-1)p(i-1) < \alpha$ holds for any $i > \delta$. Since p is a polynomial function, such constants always exist. Let X be an iteration. When X inputs Z with $|V(Z)| \leq \delta$, the computation time is $q(n)$, except for the output process. Hence, we have $T^* = O(q(n))$. For the case $|V(X)| > \delta$, we have

$$\begin{aligned}\overline{T}(X) &\geq 2(|V(Z)|-1)p(|V(Z)|-1)q(n) \\ &> \alpha|V(Z)|p(|V(Z)|)q(n),\end{aligned}$$

since X has at least two children when its input has at most c elements. Thus, X satisfies the PO condition with any constant $\beta > 0$. From Theorem 1, except for the output process, the computation time is bounded by $O(q(n))$ time for each iteration. Since any inner iteration Y has exactly one child only if $|V(Y)| \leq c$, the number of inner iterations is bounded by the number of leaf iterations, multiplied by c. Therefore, the computation time for each elimination ordering can be bounded by $O(cq(n))$ time.

Next, let us consider the output process. Instead of explicitly outputting elimination orderings, we output each elimination ordering S by the difference from S' that is output just before S. We can output them compactly in this way. Although the difference can be large up to $|V(Z)|$, we can see that it is bounded by the number of operations done from the previous output process. Thus, the size of all output differences, except for the first one output in the usual way, can be bounded by the computation time. Therefore, the computation time for the output process is also bounded by $O(q(n))$ time for each. ∎

The next corollary immediately follows the theorem.

Corollary 1 For a given set class, elimination ordering can be enumerated by Enum_Elim_Ordering in $O(1)$ amortized time for each, if each inner iteration generates at least two recursive calls, and takes $O(p(|V(Z)|))$ time, where p is a polynomial of $|V(Z)|$. ∎

There are actually several elimination orderings to which this theorem can be applied, and they are listed below. For conciseness, we have described each by their structures and removable elements.

Example (a): connected graphs
$\mathcal{L}$ is the set of connected graphs, and $V(Z)$ is the vertex set of $Z \in \mathcal{L}$. $R(Z)$ is the set of vertices whose removal results connected graphs, i.e., the set of non-cut vertices. Any connected graph with at least two vertices has at least two vertices that are not cut. Since we can check whether a vertex is a cut or not in $O(|V(Z)|^2)$ time, we can enumerate all elimination orderings in $O(1)$ time for each.

Example (b): perfect elimination orderings of a chordal graph[2]
The definition is described above. It is known that any chordal graph Z has a structure called a clique tree, whose vertices are maximal cliques of Z. If Z is a clique, all vertices in Z are simplicial. If not, the clique tree consists at least two leaves. Since a leaf of a clique tree always has a vertex that is not included in the other maximal cliques, $|R(Z)| \geq 2$ always holds. Since we can check whether a vertex is simplicial or not in $(|V(X)|^2)$ time, we can enumerate all elimination orderings in $O(1)$ time for each. Note that although the algorithm in [2] already attained the same time complexity, our analysis yields a much simpler algorithm,

Example (c): surface points of convex hull
$\mathcal{L}$ is the class of point sets in a plane, and $V(Z) = Z$. For

$Z \in \mathcal{L}$, point $z \in Z$ is removable if z is on the surface of the convex hull of Z. The convex hull of Z can be computed in $O(|V(Z)| \log |V(Z)|)$ time, and any convex hull has at least two points if $|V(Z)| \geq 2$, we can enumerate all elimination orderings in $O(1)$ time for each.

Example (d): leaf of a tree
$\mathcal{L}$ is the class of graphs that are trees, and $V(Z), Z \in \mathcal{L}$ is the vertex set of Z. A vertex of Z is removable if z is a leaf of Z, and the removal of v is a tree. A tree has at least two leaves if it has more than one vertex, thus we can enumerate all elimination orderings in $O(1)$ time for each.

Example (e): small degree vertex of a simple planar graph
$\mathcal{L}$ is the class of simple planar graphs, and $V(Z), Z \in \mathcal{L}$ is the vertex set of Z. A vertex of Z is removable if its degree is at most six, and the removal of v is a planar graph. According to Euler's polyhedron theorem, any planar graph of at least two vertices has at least two vertices of degree at most six, we can enumerate all elimination orderings in $O(1)$ time for each.

Example (f): end vertex of a DAG
$\mathcal{L}$ is the class of directed acyclic graphs, and $V(Z), Z \in \mathcal{L}$ is the vertex set of Z. A vertex of Z is removable if it has no out-going arcs or has no in-coming arc, and the removal of v is a directed acyclic graph. Since any $Z \in \mathcal{L}$ has at least one vertex having no out-going arc and at least one vertex having no in-coming arc, we can enumerate all elimination orderings in $O(1)$ time for each.

Example (g): perfect sequence[8]
$\mathcal{L}$ is the class of chordal graphs Z, and $V(Z)$ is the set of maximal cliques in Z. A maximal clique is removable if it is a leaf of some clique trees of Z, and the removal of a maximal clique z from Z is the removal of all vertices of z that do not belong to another maximal clique. The removal of the vertices results in the graph that includes remaining maximal cliques, and no new maximal clique appears in the graph. Note that a clique tree has at least two leaves if it has more than one vertex, thus $|R(Z)| \geq 2$. An elimination ordering is called a *perfect sequence*, and is enumerated in $O(1)$ time for each.

If we forget about $V(Z)$ and consider just repetitive removals of substructures, the following problems can be considered as the issues that can be solved by Enum_Elim_Ordering.

Example (h): bipartite edge color sequence
$\mathcal{L}$ consists of bipartite graphs with n vertices, and we generate an elimination ordering by iteratively removing a matching covering all maximum degree vertices. According to Koenig's theorem, such matchings always exist, and their number is at least Δ, where Δ is the maximum degree. After removing Δ matchings, the graph becomes empty. We can enumerate all such matchings in $O(n\Delta)$ time for each, thus the enumeration of elimination orderings is done in $O(n)$ time for each. We remind that the algorithms in [9], [10] enumerate all sets of matchings that compose edge colorings. The algorithm in [10] attains $O(|V|)$ time for each solution.

There are several classes of elimination orderings and similar structures for that Enum_Elim_Ordering do not satisfy the conditions of the theorem. A topological ordering is an elimination ordering that is obtained by removing vertices one by one from a directed acyclic graph such that the vertex removed is a vertex having no out-going arc. Since there are several graphs that have only one removable element, some inner iterations will have only one recursive call. The followings are removals sequences, but the lengths of sequences may not be the same, thus the computation time for each iteration is not bounded as condition (a) of the theorem.

- (cycle decomposition) $\mathcal{L}$ is the set of graphs all of whose vertices have even degrees, and removable elements are its cycles
- (ear decomposition) $\mathcal{L}$ is the set of graphs, and removable elements are its paths whose end points are of degrees more than two, and whose internal vertices have degrees of two.

5. Enumeration of Matchings

A *matching* of a graph is an edge subset of a graph $G = (V, E)$ such that no two edges are adjacent, i.e., no vertex is incident to two of the edge subset. The matchings are enumerated by the following algorithm. In this section, we assume that the input graph has no isolated vertex, which has no edge incident to it.

ALGORITHM Enum_Matching $(G = (V, E), M)$
1: **if** $E = \emptyset$ **then output** M; **return**
2: choose an edge e from E
3: **call** Enum_Matching $(G \setminus e, M)$
4: **call** Enum_Matching $(G^+(e), M \cup \{e\})$

The time complexity of an iteration of Enum_Matching is $O(|V|)$. Since each inner iteration generates two children, the amortized computation time for each matching is $O(|V|)$, and no better algorithm has been proposed in the literature. The leaf iteration takes $O(1)$ time, thus $T^* = O(1)$. However, the PO condition may not hold for some iterations. Thus, in a straightforward way, this cannot be better than $O(|V|)$.

The PO condition actually does not hold when many edges are adjacent to e, i.e., the endpoints of e have large degrees compared to $|E|$. In such cases, $G^+(e)$ may have small numbers of edges, thus the computation time of the subproblem inputting $G^+(e)$ takes short time so that the PO condition does not hold. To avoid this problem, we modify the way of recursion so that in such cases the iteration has many children whereby the PO condition always holds. The pseudo code is described as follows.

ALGORITHM Enum_Matching2 $(G = (V, E), M)$

1: **if** $E = \emptyset$ **then output** M; **return**
2: choose a vertex v having the maximum degree in G
3: **call** Enum_Matching2 $(G \setminus v, M)$
4: **for** each edge e adjacent to v
call Enum_Matching2 $(G^+(e), M \cup \{e\})$

Let $u_1, \ldots, u_k$ be the vertices adjacent to v, and $e_i = (v, u_i)$. We partition the matchings to be enumerated into

- matchings including e_1
- matchings including e_2
- $\cdots$
- matchings including e_k
- matchings including no edge incident to v.

We can see that any matching belongs to exactly one of these groups. To recur, we derive $G^+(e_1), \ldots, G^+(e_k)$ and $G \setminus v$. $G \setminus v$ and $G^+(e_1)$ can be derived in $O(|E|)$ time. To shorten the computation time for $G^+(e_i)$ for $i \geq 2$, we construct $G^+(e_i)$ from $G^+(e_{i-1})$. We add all edges of G adjacent to u_{i-1} to $G^+(e_{i-1})$, and remove all edges adjacent to u_i, and obtain $G^+(e_i)$. This can be done in $O(d(u_{i-1}) + d(u_i))$ time. To construct $G^+(e_i)$ for all $i = 2, \ldots, k$, we need

$$O(\ (d(u_1) + d(u_2))\ +\ (d(u_2) + d(u_3))\ +\ \cdots\ + (d(u_{k-1}) + d(u_k))\ )\ =\ O(|E|)$$

time. Thus, the computation time of an iteration is bounded by $c|E|$ with a constant c.

Theorem 3 All matchings in a graph can be enumerated in $O(1)$ time for each, with $O(|E| + |V|)$ space.

Proof: Let us consider an inner iteration X. In the iteration X, if $d(v) \geq |E|/4$, we generate at least $|E|/4$ recursive calls, thus we have $|C(X)| = \Omega(|E|)$ and the PO condition is satisfied by choosing sufficiently large β. If $d(v) < |E|/4$, the subproblems of $G \setminus v$ take at least $\Theta(3c|E|/4)$ time, and the subproblems of $G^+(e_1)$ take at least $c|E|/2$ time. Hence, by setting $\alpha = 1.25$, we have

$$\overline{T}(X) \geq 3c|E|/4 + c|E|/2 = 5c|E|/4 \geq \alpha T(X) - \beta c(X)T^*$$

thereby the PO condition holds. Since any inner iteration satisfies the PO condition and $T^* = O(1)$, the statement holds. We remind that we assumed that there is no isolated vertex in the input graph, and thus the number of matchings in the graph is greater than the number of vertices, and the number of edges.

In each iteration X, graph G' given to a child iteration is constructed by removing edges from its input graph G. To reconstruct G from G', we store the removed edges in memory. Since any edge is removed in at most one iteration among these iterations, the accumulation of the stored edges is bounded by $O(|E|)$. This concludes the proof. ∎

6. Enumeration of Connected Vertex Induced Subgraphs

We consider the problem of enumerating all connected vertex induced subgraphs of the given graph $G = (V, E)$. Here we enumerate all vertex subsets which induces connected subgraphs. In literature, an algorithm is proposed that runs in $O(|V|)$ time for each connected vertex induced subgraphs[1]. For a vertex r of G, the connected vertex induced subgraphs are partitioned into two groups, one is of those including r and the other is of those not including r. The connected vertex induced subgraphs included in the latter group can be enumerated by recursively solving the problem of $G \setminus r$. Hence, we consider the former problem. For a vertex v adjacent to r, the connected vertex induced subgraphs including r are partitioned into two groups; those including v and those not including v. The former group is the set of connected vertex induced subgraphs in $\underline{G/(r, v)}$ and the latter is those in $G \setminus v$. According to this partition, we can solve the enumeration problem with the following algorithm. We will prove that this algorithm satisfies the PO condition.

ALGORITHM Enum_Connect $(G = (V, E), S, r)$
1: **output** S
2: **if** $d(r) = 0$ **then return**
3: choose a vertex v adjacent to r
4: **call** Enum_Connect $(\underline{G/(r, v)}, S \cup \{v\}, r)$
5: **call** Enum_Connect $(G \setminus v, S, r)$

Theorem 4 All connected vertex induced subgraphs in a graph can be enumerated in $O(1)$ time for each, with $O(|E| + |V|)$ space.

Proof: The correctness of the algorithm and the bound for memory usage are clear. Since each inner iteration generates exactly two recursive calls, the number of iterations is linearly bounded by the number of connected vertex subgraphs. Thus, we show that the amortized computation time for each iteration is $O(1)$.

As same to the matching enumeration, we consider the computation time except for the output process, since the time for the output process is linear for the other processes. An inner iteration X of the algorithm takes $O(d(r) + d(v))$ time. We assume that $T(X) = c(3d(r)+d(v))$ for a constant c, and leaf iteration takes $3c$ time, since $T^* = O(1)$. The constant factor of three seems to be unnecessary, but this is a trick to satisfy the PO condition.

Let us see the computation time of child iterations of X. The degree of r is at least $(d(r) + d(v))/2 - 1$ in $\underline{G/(r, v)}$, and $d(r) - 1$ in $G \setminus v$. Note that $d(r)$ and $d(v)$ are degrees of r and v in G. From this, we can see that the child iteration of $\underline{G/(r, v)}$ takes at least $3c((d(r) + d(v))/2 - 1)$ time, and that of $G \setminus v$ takes at least $3c(d(r) - 1)$ time. Their sum is at least

$$\begin{aligned} &3c((d(r) + d(v))/2 - 1) + 3c(d(r) - 1) \\ =\ &\frac{3}{2}c(3d(r) + d(v)) - 6c = \frac{3}{2}T(X) - 6c. \end{aligned}$$

Setting $\beta = 6$, we can see that X satisfies the PO condition. Thanks to Theorem 1, the computation time for each connected vertex induced subgraph is $O(1)$. ∎

7. Spanning Trees

This section takes into consideration the problem of enumerating all spanning trees in an undirected simple graph. A subtree T of a graph $G = (V, E)$ is called a *spanning tree* if any vertex of G is incident to at least one edge of T. The size of the spanning tree is invariant, and is $|V| - 1$. There have already been several studies on this problem[7], [13], [14]. Of these, the algorithm in [14] is the simplest and uses an amortized analysis similar to ours. This section shows how we can apply our push out amortization to this algorithm to obtain the same results in a simple way. For conciseness, we assume that input graph does not have any bridge. If there are some bridges, they are always included in any spanning trees, thus we can reduce the problem by contracting all bridges.

The idea behind the algorithm in [14] is based on a simple binary partition, that works even for multi-graphs that have parallel edges. For an edge e_1 of a graph $G = (V, E)$, the spanning trees of G are partitioned into two groups; those including e_1 and of those not including e_1. The former group consists of spanning trees in G/e_1, and the latter group consists of spanning trees in $G \setminus e_1$. This gives us a recursive algorithm for the enumeration. Further, we use other ways of partitioning to make the algorithm faster. If there are several edges $e_2, \ldots, e_k$ parallel to e_1, we can observe that at most one of them is included in a spanning tree. Thus, we solve the subproblems of graphs $(G \setminus F_1)/e_1, \ldots, (G \setminus F_k)/e_k$, and, $G \setminus F$ if it is connected, where $F = \{e_1, \ldots, e_k\}$ and $F_i = F \setminus \{e_i\}$. We can also see that $(G \setminus F_i)/e_i$ has neither bridge nor selfloop, if G does not have those. We recurse on $G \setminus F_i$ when it is connected. We do not recurse it when it is disconnected, since it has no spanning tree.

When e_1 has no parallel edges, e_1 can have series edges. If there are several edges $e_2, \ldots, e_k$ series to e_1, any spanning tree includes at least $k-1$ of these. In this case we recur with the subproblems of the graphs $(G/F_1) \setminus e_1, \ldots, (G/F_k) \setminus e_k$, and, G/F if F is not a cycle. Similar to the above, $(G/F_i) \setminus e_i$ has neither bridge nor selfloop. $(G/F_i) \setminus e_i$ is isomorphic to $(G/F_j) \setminus e_j$ for any other j, and can be constructed from $(G/F_j) \setminus e_j$ in $O(d(u_i) + d(v_i) + d(u_j) + d(v_j)$ time where $e_i = (u_i, v_i)$ and $e_j = (u_j, v_j)$. The algorithm is described as follows.

ALGORITHM Enum_Span_Tree $(G = (V, E), T)$
1: **if** $E = \emptyset$ **then output** T; **return**
2: choose an edge e_1 from E
3: $F^p := \{e_1\} \cup \{e | e$ is parallel to $e_1\}$
$F^s := \{e_1\} \cup \{e | e$ is not parallel to e_1, and e is series to $e_1\}$
4: **for** each $e_i \in F^p$
call Enum_Span_Tree $((G \setminus (F^p \setminus \{e_i\})/e_i, T \cup \{e_i\})$
5: **for** each $e_i \in F^s$
call Enum_Span_Tree $((G/(F^s \setminus \{e_i\}) \setminus e_i, T \cup (F^s \setminus \{e_i\}))$

Theorem 5 All spanning trees in a graph can be enumerated in $O(1)$ time for each, with $O(|E| + |V|)$ space.

Proof: Similar to matchings and connected vertex induced subgraphs, we can see that the space complexity of the algorithm is $O(|E| + |V|)$, since all edges parallel/series to an edge can be found by two connected component decomposition in $O(|V| + |E|)$ time. Due to short time constructions of subproblems, for an inner iteration X, we can set $T(X) = c|E|$ and $T^* = 4c$ for a constant c. If no edge is parallel or series to e_1, we generate two subproblems of $|E| - 1$ edges, thus the PO condition holds. If k edges are parallel or series to e_1, we have at least $k + 1 \geq 2$ subproblems of $|E| - (k + 1)$ edges. Since either $k + 1 \geq |E|/2$ or $(k + 1)(|E| - (k + 1)) \geq 1.2$, the PO condition holds for $\alpha = 1.2$ and some positive β. Therefore, the statement holds. ∎

8. Conclusion

We introduced a new way of looking at amortizing the computation time of enumeration algorithms, by local conditions of recursion trees. We clarified the conditions that are sufficient to give non-trivial upper bounds for the average computation time of iterations that only depended on the relation between the computation time for a parent iteration and its child iterations. We showed that many algorithms for elimination orderings have good properties so that the conditions are satisfied, and thus enumerated in constant time for each. Several other enumeration algorithms for matchings, connected vertex induced subgraphs, and spanning trees were also described, whose time complexities are $O(1)$ for each solution.

There are many problems for those enumeration algorithms that do not satisfy the conditions. An interesting future work is to develop new algorithms for these problems, that satisfy the conditions. Another direction is to study other conditions for bounding amortized computation time. Enumeration algorithms are often very fast in practice, compared to the worst case complexity. Further studies on amortized analysis will possibly fill the gaps, and clarify the mechanism for enumeration algorithms.

Acknowledgments: Part of this research is supported by the Funding Program for World-Leading Innovative R&D on Science and Technology, Japan, and Grant-in-Aid for Scientific Research (KAKENHI), Japan.

References

[1] D. Avis and K. Fukuda, Reverse Search for Enumeration, *Discrete Applied Mathematics* **65**, pp. 21-46 (1996).
[2] L. S. Chandran, L. Ibarra, F. Ruskey, and J. Sawada, Generating and Characterizing the Perfect Elimination Orderings of a Chordal Graph, *Theoretical Computer Science* **307**, pp. 303-317 (2003).
[3] D. Eppstein, Finding the k Smallest Spanning Trees, *SWAT 90, Lecture Notes in Computer Science* **447**, pp. 38-47 (1990).
[4] D. Eppstein, Finding the k Shortest Paths, *FOCS 94*, pp. 154-165 (1994).
[5] R. Ferreira, R. Grossi and R. Rizzi, Output-sensitive Listing of Bounded-size Trees in Undirected Graphs", *ESA 2011, Lecture Notes in Computer Science* **6942**, pp. 275-286 (2011).

[6] Frequent Itemset Mining Dataset Repository, http://fimi.cs.helsinki.fi/data/
[7] H. N. Kapoor and H. Ramesh, Algorithms for Enumerating all Spanning Trees of Undirected and Weighted Graphs, *SIAM Journal on Computing*, **24**, pp. 247-265 (1995).
[8] Y. Matsui, R. Uehara and T. Uno, Enumeration of the Perfect Sequences of a Chordal Graph, *Theoretical Computer Science* **411**, pp. 3635-3641 (2010).
[9] Y. Matsui and T. Matsui, Enumeration Algorithm of the Edge Colorings in Bipartite Graphs, *Lecture Notes in Computer Science* **1120**,pp. 18–26 (1995).
[10] Y. Matsui and T. Uno, On the Enumeration of Bipartite Minimum Edge Colorings, *Graph Theory in Paris, Proceedings of a Conference in Memory of Claude Berge*, Birkhaeuser Boston Inc., pp. 271–285 (2006).
[11] D. J. Rose, R. E. Tarjan, and G. S. Lueker, Algorithmic Aspects of Vertex Elimination on Graphs, *SIAM Journal on Computing* **5**, pp. 266-283 (1976).
[12] D. J. Rose and R. E. Tarjan, Algorithmic Aspects of Vertex Elimination on Directed Graphs, *SIAM Journal on Applied Mathematics* **34**, pp. 176-197 (1978).
[13] A. Shioura, A. Tamura and T. Uno, An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs, *SIAM Journal on Computing* **26**, pp. 678-692 (1997).
[14] T. Uno, A New Approach for Speeding Up Enumeration Algorithms and Its Application for Matroid Bases, *COCOON 99, Lecture Notes in Computer Science* **1627**, pp. 349-359 (1999).