

Generating All Series-parallel Graphs

(Extended Abstract)

Shin-ichiro Kawano[†]Shin-ichi Nakano[†]**Abstract**

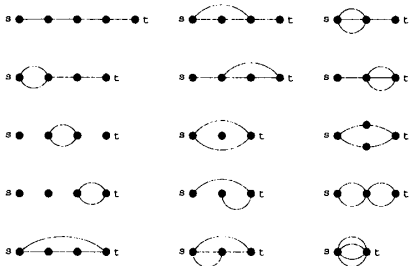
In this paper we give an algorithm to generate all series-parallel graphs with at most m edges. This algorithm generates each series-parallel graph in constant time on average.

1 Introduction

It is useful to have the complete list of graphs with a specified property. One can use such a list to search for a counter-example to some conjecture, or to experimentally measure an average performance of an algorithm over all possible input graphs.

Many algorithms to generate a particular class of graphs without repetition are already known [B80, LN01, LR99, M98, N02, N04, R78, W86]. Many nice textbooks have been published on the subject [G93, KS98, W89].

In this paper we give an algorithm to generate all series-parallel graphs having at most m edges without repetition. For example, all series-parallel graphs having four edges are shown in Fig.1. Series-parallel

Figure 1: All series-parallel graphs $G(s, t)$ with $m = 4$.

graphs are important class of recursively defined graphs having a nice tree structure.

One can generate series-parallel graphs by following the recursive definition. However such method needs much running time, and may output graphs with many repetitions. Our algorithm generates each series-parallel graph without repetition in constant time on average.

The main idea of our algorithm is as follows. We do not directly generate each series-parallel graph. First, we assign a unique ordered tree for each series-parallel graph. Then, we define a tree, called “the family tree”, so that each ordered tree assigned above corresponds to a distinct vertex of the family tree. By efficiently traversing the family tree, we generate all series-parallel trees without repetition.

Using similar method we can generate several planar structures [LN01, N02, N04]. In this paper we first extend the method for more general graphs.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 introduces the family tree. Section 4 presents our algorithm. Finally Section 5 is a conclusion.

2 Preliminaries

In this section we give some definitions. Let G be a connected graph with n vertices and m edges. A *tree* is a connected graph without cycles. A *rooted tree* is a tree with one vertex r chosen as its *root*. For each vertex v in a rooted tree, let $UP(v)$ be the unique path from v to the root r . If $UP(v)$ has exactly k edges then we say the *depth* of v is k , and write $dep(v) = k$. The *parent* of $v \neq r$ is its neighbor on $UP(v)$, and *ancestors* of $v \neq r$ are the vertices on $UP(v)$ except v . The parent and the ancestors of r are not defined. We say that if v is the parent of u then u is a *child* of v , and if v is an ancestor of u then u is a *descendant* of v . A *leaf* is a vertex having no child. An *ordered tree* is a rooted tree with a left-to-right ordering specified for the children of each vertex. We denote by $T(v)$ the ordered subtree of an ordered tree T consisting of a vertex v and all descendant of v preserving the left-to-right ordering for the children of each vertex.

A graph $G(s, t)$ is a *series-parallel graph* with terminals s and t , if (1) G consists of one edge connecting s and t , or (2) G is derived from two or more series-parallel graphs by one of the following two operations.

- *The series composition:* Given k series-parallel graphs $G_1(s_1, t_1), G_2(s_2, t_2), \dots, G_k(s_k, t_k)$, form a new graph $G(s, t)$ by identifying $s = s_1, t = t_k$, and $t_i = s_{i+1}$ for $1 \leq j \leq k - 1$.
- *The parallel composition:* Given k series-parallel graphs $G_1(s_1, t_1), G_2(s_2, t_2), \dots, G_k(s_k, t_k)$, form a new graph $G(s, t)$ by identifying $s = s_1 = s_2 = \dots = s_k$, and $t = t_1 = t_2 = \dots = t_k$.

Note that the ordering $G_1(s_1, t_1), G_2(s_2, t_2), \dots, G_k(s_k, t_k)$ matters for the series composition, while it does not matter for the parallel composition.

The recursive definition of the series-parallel graph above naturally gives a tree T , called a *series-parallel tree*, for each series-parallel graph $G(s, t)$. Each leaf in T corresponds to an edge of $G(s, t)$, and each non-leaf vertex in T corresponds to either series or parallel composition. We say that each vertex is *normal*, *series*, or *parallel*, respectively. We can observe that if

[†] Gunma University

the root vertex is series vertex, then every non-leaf vertex at even depth is also a series vertex, while every non-leaf vertex at odd depth is a parallel vertex. (The other case is similar.)

Note that a series-parallel graph G may have many corresponding series-parallel trees, since we can choose any ordering for child vertices of each parallel vertex. We are going to assign a unique ordered tree for each series-parallel graph. We need some definitions here.

Let T be an ordered tree with n vertices, and (v_1, v_2, \dots, v_n) be the vertices of T in preorder [A95]. Let $dep(v)$ be the depth of v . Then the sequence $L(T) = (dep(v_1), dep(v_2), \dots, dep(v_n))$ is called the *depth sequence*. Let T_1 and T_2 be two ordered trees, and $L(T_1) = (a_1, a_2, \dots, a_c)$ and $L(T_2) = (b_1, b_2, \dots, b_d)$. Then we say that T_1 is *heavier* than T_2 , if $a_i = b_i$ for each $i = 1, 2, \dots, k - 1$ (possibly $k = 1$) and either $a_k > b_k$ or $c > k - 1 = d$.

Now we assign the heaviest ordered tree H for each series-parallel graph G . We call such the heaviest ordered tree H the *canonical tree* of G .

Let S_m be the set of all canonical trees with at most m leaves. Note that each tree in S_m corresponds to each series-parallel graph having at most m edges.

We have the following lemma.

Lemma 2.1 *A series-parallel tree T is in S_m if and only if T has at most m leaves, and for every consecutive child vertices v_1 and v_2 of every parallel vertex, $L(T(v_1)) \geq L(T(v_2))$ holds.*

Proof. By contradiction. Omitted. □

We call the condition above “the left heavy condition”.

3 The family tree

Assume $m \geq 2$. Let $T \in S_m$, be a canonical tree. We say a vertex v in T is *un-removable* if v satisfies the following three conditions.

- (co1) v is normal,
- (co2) v is the rightmost vertex in its siblings, and
- (co3) v has exactly one sibling (except v).

A leaf v is *removable* if it is not un-removable. The last removable vertex of T in preorder is called the *last removable vertex* of T .

Let u be the last removable vertex of T , and v the parent of u . Also let w be the parent of v if v is not the root of T .

We define a new tree $P(T)$ as follows.

We have the following two cases, depending on the number of child vertices of v .

Case1: v has exactly two child vertices.

Now v has two child leaves. We have the following two subcases.

Case1-1: w has exactly two child vertices, and v is the right child of w .

(r1) Then replace $T(v)$ by one normal vertex. Note that the new vertex is un-removable.

Case1-2: Otherwise. Now we have two cases (1) w has exactly two child vertices, and v is the left child of w , or (2) w has three or more child vertices, and v is the rightmost child of w .

(r2) Then replace $T(v)$ by one normal vertex. Note that the new vertex is removable.

Case2: v has three or more child vertices.

(r3) Remove u .

Note that in all cases above, $P(T)$ has one less leaves than T . We say that $P(T)$ is the *parent* of T , and T is a *child* of $P(T)$. We have the following lemma.

Lemma 3.1 *If T is canonical then $P(T)$ is also canonical.*

Proof. In $P(T)$, only subtrees rooted at vertices on the path between the root and the new vertex lose the “weight”. So we need to check the left heavy condition for those subtrees. Since only trivial trees, consisting of one un-removable vertex, exist on the right of the subtrees above, the left heavy condition holds in $P(T)$. □

Repeatedly applying above operations to any canonical tree $T \in S_m$, we have a sequence $P(T), P(P(T)), P(P(P(T))), \dots$ of canonical trees, and the sequence eventually ends with the canonical tree having only one (normal) vertex. We denote the trivial canonical tree by T_1 . See an example in Fig.2.

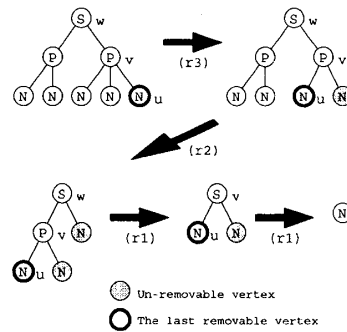


Figure 2: The removing sequence.

By merging those sequences we have a tree F_m such that each vertex corresponds to a distinct canonical tree in S_m , each edge corresponds to some relation between some T and $P(T)$. We call F_m the *family tree* of S_m .

4 Algorithm

In this section we give an algorithm to construct F_m . We only consider the case the root of T is parallel. The other case is omitted since it is similar.

Given a canonical tree T in S_m , if we have an algorithm to generate all child canonical trees of T , then in a recursive manner we can generate F_m , and which means we can generate all series-parallel graphs having at most m edges. How can we generate all child canonical trees of a given canonical tree? As we will soon see we can do this by “reversing” the operations (r1)–(r3) in Section 3.

Let T be a canonical tree in S_m , r_k be the last removable vertex of T , and $RP = (r_0, r_1, \dots, r_k)$ be the path between the root r_0 and r_k . We construct three types of new trees $T[i], T_+[i], T_-$, from T as follows.

For $i, 0 \leq i \leq k-1$, we define $T[i]$ to be the canonical tree derived from T by

(a1) adding a new vertex x as the rightmost child of r_i . See some examples in Fig.3 (b)–(d). Note that the last removable vertex of $T[i]$ is the new vertex x .

For $i, 0 \leq i \leq k-1$, if r_i has exactly two child vertices, and the right child vertex w of r_i is normal, then we define $T_+[i]$ to be the canonical tree derived from T by

(a2) replacing w by either a series or parallel vertex x and add two normal child vertices to x . See Fig.3 (e) and (f). Note that the last removable vertex of $T_+[i]$ is the left child of vertex x .

By definition, r_{k-1} always has two normal child vertices. We define T_- to be the canonical tree derived from T by

(a3) replacing r_k by either a series or parallel vertex x and add two normal child vertices to x . See Fig.3 (g). Note that the last removable vertex of T_- is the left child vertex of x .

We can observe that each operation (a1), (a2) and (a3) is the reverse of (r3), (r2) and (r1), respectively. Each derived tree has one more leaves than T .

Define $C(T) = \{T[0], T[1], \dots, T[k-1]\} \cup \{T_+[0], T_+[1], \dots, T_+[k-1]\} \cup \{T_-\}$, those are candidates for child trees of T . We can observe that each child tree of $T \in S_m$ is in $C(T)$, however, not all trees in $C(T)$ are child trees of T . For example, the tree $T_+[2]$ in Fig.3(f) is not a child tree of T , since it is not a canonical tree, so $T_+[2] \notin S_m$. Thus we need to check whether each possible child tree is actually a child tree of T or not.

We now have the following lemma.

Lemma 4.1 *Let $T \in S_m$, $T' \in C(T)$, r_k be the last removable vertex of T' and $RP = (r_0, r_1, \dots, r_k)$ be the path of T' between the root r_0 and r_k . Then T' is a child tree of T if and only if $L(T'(s_{i+1})) \geq L(T'(r_{i+1}))$ holds for every parallel vertex $r_i, 0 \leq i < k$, on RP , where s_{i+1} is the child of r_i preceding r_{i+1} .*

Proof. Since $T \in S_m$, the left heavy condition has held in T . In T' some subtrees may be heavier than in T . So we must check if left heavy condition still holds or not. The claim checks all of these possible changes to destroy the left heavy condition. \square

If we generate each tree in $C(T)$ then check whether it is actually a child tree or not based on the lemma above, then we need much running time. However we can improve the running time as follows. We need some definition here.

Let T be a canonical tree in S_m , r_k be the last removable vertex of T . $RP = (r_0, r_1, \dots, r_k)$ be the path of T between the root r_0 and r_k . Let T_r be the tree derived from T by removing all un-removable vertices. We say that T is *active* at depth $i, 0 \leq i \leq k-1$, if

- (i) r_i is a parallel vertex.
- (ii) r_i has the child vertex s_{i+1} preceding r_{i+1} .
- (iii) $L(T_r(r_{i+1}))$ is a prefix of $L(T_r(s_{i+1}))$.

Intuitively, if T is active at depth i , then we are copying subtree $T(r_{i+1})$ from $T(s_{i+1})$. We say that the *copy-depth* of T is c if T is active at depth c but not active at each $i \in \{0, 1, \dots, c-1\}$. Especially if T is not active at any in $\{0, 1, \dots, k-1\}$, then we define the copy depth of T is k .

Now we are going to check each tree in $C(T)$ is actually a child tree of T or not. Let c be the copy-depth of T . Assume that the root vertex of T is parallel vertex. (The other case is similar.)

First we consider for $T[i], 0 \leq i < k$.

Case $T[i]$

We have the following four cases.

Case 1: T has m leaves.

Then T corresponds to a leaf in F_m . Hence T has no child tree.

Case 2: Otherwise, $c = k$.

In this case $L(T_r(s_{i+1})) > L(T_r(r_{i+1}))$ holds for each parallel vertex r_i . Now $T[0], T[1], \dots, T[k-1]$ are all child trees of T . In each tree $T[i]$, the last removable vertex is x . The copy-depth of $T[i]$ is i for each even i , (that is a parallel vertex) and $i+1$ for each odd i . For example, a tree T and some child trees are shown in Fig.4. In $T[2]$, (i) r_2 is parallel vertex, (ii) r_2 has the child vertex y preceding r_3 , (iii) $L(T_r(r_3))$ is a prefix of $L(T_r(y))$. Hence $T[2]$ is active at depth 2 and the copy depth of $T[2]$ is 2. In $T[3]$, r_3 is not parallel vertex. Hence r_3 is not active and copy depth of $T[3]$ is $k = 4$.

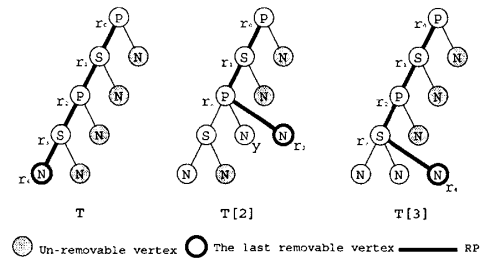


Figure 4: Illustrations for $T[i]$.

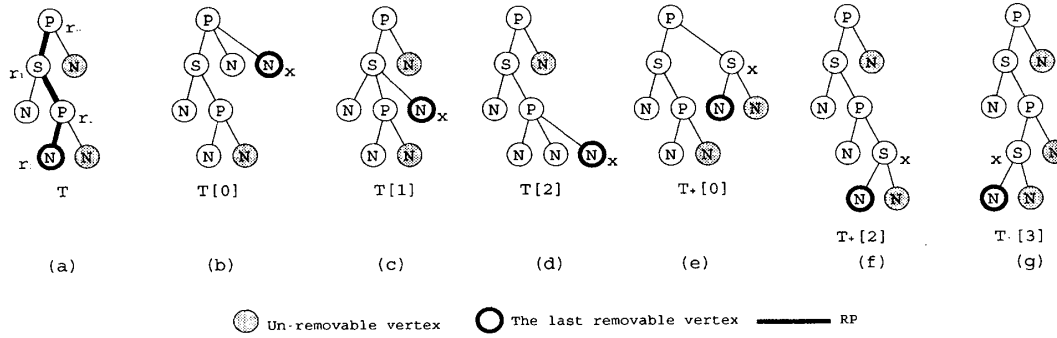


Figure 3: The possible child series-parallel trees.

Case 3: Otherwise, $L(T_r(r_{c+1})) = L(T_r(s_{c+1}))$. (Intuitively the copy has completed.)

In this case $T[0], T[1], \dots, T[c]$ are child trees of T . The copy-depth of $T[i]$ is i for each even i , and $i + 1$ for each odd i . $T[c + 1], T[c + 2], \dots, T[k - 1]$ are not child trees of T .

Case 4: Otherwise. (Intuitively the copy has not completed yet.)

Now $L(T_r(s_{c+1})) \geq L(T_r(r_{c+1}))$ holds.

Let $L(T_r(s_{c+1})) = (dep(u_1), dep(u_2), \dots, dep(u_{n'}), \dots, dep(u_{n''}))$, $L(T_r(r_{c+1})) = (dep(v_1), dep(v_2), \dots, dep(v_{n'}))$, and set $d = dep(u_{n'+1})$. (Intuitively we are copying $T_r(r_{c+1})$ from $T_r(s_{c+1})$ and $u_{n'+1}$ is the next vertex to be copied.) In this case $T[0], T[1], \dots, T[d - 1]$ are child trees of T . For $i = 0, 1, \dots, d - 2$, The copy-depth of $T[i]$ is i for each even i , and $i + 1$ for each odd i . The copy-depth of $T[d - 1]$ is remains at c .

Next we consider for $T_+[i]$, $0 \leq i \leq k - 1$.

Case $T_+[i]$ Omitted. See [KN04].

Next we consider for T_- .

Case $T_-[i]$ Omitted. See [KN04].

Based on the case analysis above we have the following theorem.

Theorem 4.2 *Given m , one can generate all series-parallel graphs with at most m edges without repetition in $O(|S_m|)$ time.*

5 Conclusion

In this paper we have given a simple algorithm to generate all series-parallel graphs with at most m edges. Our algorithm first defines a family tree such that each vertex corresponds to each series-parallel trees with at most m leaves, then outputs each graph without repetition by traversing the family tree. Our algorithm generates each series-parallel graph in constant time on average. When traverse the family tree we need $O(h)$ space, where h is the height of family tree, which is bounded by n . Thus the algorithm uses $O(n)$ space.

References

- [A95] A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, (1995).
- [B80] T. Beyer and S. M. Hedetniemi, *Constant Time Generation of Rooted Trees*, SIAM J. Comput., 9, (1980), pp.706-712.
- [G93] L. A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge University Press, New York, (1993).
- [KN04] S. Kawano and S. Nakano, *Generating All Series-parallel Graphs*, IPSJ Technical Reports, SIGAL-94, (2004), pp.41-48.
- [KS98] D. L. Kreher and D. R. Stinson, CRC Press, Boca Raton, (1998).
- [LN01] Z. Li and S. Nakano, *Efficient Generation of Plane Triangulations without Repetitions*, Proc. ICALP2001, LNCS 2076, (2001), pp.433-443.
- [LR99] G. Li and F. Ruskey, *The Advantage of Forward Thinking in Generating Rooted and Free Trees*, Proc. 10th Annual ACM-SIAM Symp. on Discrete Algorithms, (1999), pp.939-940.
- [M98] B. D. McKay, *Isomorph-free Exhaustive Generation*, J. of Algorithms, 26, (1998), pp.306-324.
- [N02] S. Nakano, *Efficient Generation of Plane Trees*, Information Processing Letters, 84, (2002), pp.167-172.
- [N04] S. Nakano, *Efficient Generation of Triconnected Plane Triangulations*, Computational Geometry Theory and Applications, Vol. 27(2), (2004), pp.109-122.
- [R78] R. C. Read, *How to Avoid Isomorphism Search When Cataloguing Combinatorial Configurations*, Annals of Discrete Mathematics, 2, (1978), pp.107-120.
- [W89] H. S. Wilf, *Combinatorial Algorithms : An Update*, SIAM, (1989).
- [W86] R. A. Wright, B. Richmond, A. Odlyzko and B. D. McKay, *Constant Time Generation of Free Trees*, SIAM J. Comput., 15, (1986), pp.540-548.