# Variations of Base Station Placement Problem on the Boundary of a Convex Region

Gautam K. Das, Sasanka Roy, Sandip Das and Subhas C. Nandy

Indian Statistical Institute, Kolkata - 700 108, India

## Abstract

Due to the recent growth in the demand of mobile communication services in several typical environments, the development of efficient systems for providing specialized services has become an important issue in mobile communication research. An important sub-problem in this area is the base-station placement problem, where the objective is to identify the location for placing the base-stations. Mobile terminals communicate with their respective nearest base station, and the base stations communicate with each other over scarce wireless channels in a multi-hop fashion by receiving and transmitting radio signals. Each base station emits signal periodically and all the mobile terminals within its range can identify it as its nearest base station after receiving such radio signal. Here the problem is to position the base stations such that each point in the entire area can communicate with at least one base-station, and total power required for all the base-stations in the network is minimized. A different variation of this problem arises when some portions of the target region is not suitable for placing the base-stations, but the communication inside those regions need to be provided. For example, we may consider the large water bodies or the stiff mountains. In such cases, we need some specialized algorithms for efficiently placing the base-stations on the boundary of the forbidden zone to provide services inside that region.

In our model, all the $k$ base-stations are similar; in other words, their range/power-requirement are same. We shall consider two variations of the problem where the region $P$ and the number of base-stations $k$ are given a priori.

**region-cover($k$):** Place the base-stations on the boundary of $P$ to cover the entire region $P$.

**vertex-cover($k$):** Place the base-stations on the boundary of $P$ to cover the vertices of $P$.

We first present a polynomial time algorithms for the *vertex-cover(2)* and *region-cover(2)* problems, where the base-stations may appear any where on the boundary of $P$. We also show that, if a pair of edges of the polygon is specified on which the base-stations can be installed, then the *vertex-cover(2)* and the *region-cover*(2) problems can be optimally solved in polynomial time. More specifically, the time complexity of these two problems are $O(n \log n)$ and $O(n^2)$ respectively. Next, we consider a restricted version of both the problems where all the $k$ ($\geq 3$)

M. Kaykobad and Md. Saidur Rahman (Eds.): WALCOM 2007, pp. 151–152, 2007.

base-stations can be placed on an edge of $P$. The objective is to minimize the (common) range of the base-stations. Our proposed algorithm for the *restricted vertex-cover(k)* problem produces optimum result in $O(min(n^2, nk \log n))$ time, whereas the algorithm for the *restricted region-cover(k)* produces an additive $\epsilon$ approximation result in the sense that if $\rho$ is the optimum solution, then our algorithm returns a value less than or equal to $\rho + \epsilon$. The time complexity of our algorithm is $O(n \log \frac{\Pi}{\epsilon})$, where $\Pi$ is the perimeter of the polygon $P$. Finally, we will describe a heuristic algorithm for the unrestricted *region-cover(k)* problem, where $k \geq 3$. Experimental results demonstrate that our proposed algorithm runs fast and produces near optimum solutions.

# Constant Time Generation of Linear Extensions
## (Extended Abstract)

Akimitsu Ono and Shin-ichi Nakano

Gunma University, Kiryu-Shi 376-8515, Japan
ono@msc.cs.gunma-u.ac.jp, nakano@cs.gunma-u.ac.jp

**Abstract.** Given a poset $\mathcal{P}$, several algorithms have been proposed for generating all linear extensions of $\mathcal{P}$. The fastest known algorithm generates each linear extension in constant time "on average". In this paper we give a simple algorithm which generates each linear extension in constant time "in worst case". The known algorithm generates each linear extension exactly twice and output one of them, while our algorithm generates each linear extension exactly once.

## 1 Introduction

A linear extension of a given poset $\mathcal{P}$ is one of the most important notion associated with $\mathcal{P}$. An example of a poset is shown in Fig. 1, and its linear extension is shown in Fig. 2. Many scheduling problems with precedence constraints are modeled by a linear extension of a poset, or equivalently a topological sort[C01] of an acyclic digraph [PR94]. Even though many such scheduling problems are NP-complete, one can solve the problem by first generating all linear extensions of a given poset and then picking the best one [PR94]. Linear extensions are also of interest to combinatorists, because of their relation to counting problems [St97].

Let $\mathcal{P} = (S, R)$ be a poset with a set $S$ and a binary relation $R$ on $S$. We write $n = |S|$ and $m = |R|$. It is known one can find a linear extension of a given poset $\mathcal{P}$ in $O(m + n)$ time [C01, p.550].

Many algorithms to generate a particular class of objects, without repetition, are already known [LN01,LR99,M98,N02,R78]. Many excellent textbooks have been published on the subject [G93,KS98,W89]. Given a poset $\mathcal{P}$, three algorithms to generate all linear extensions of $\mathcal{P}$ are explained in [KV83]. The best algorithm among them generates the first linear extension in $O(m + n)$ time, then generates each linear extension in $O(n)$ time.

Generally, generating algorithms produce huge outputs, and the outputs dominate the running time of the generating algorithms. So if we can compress the outputs, then it considerably improves the efficiency of the algorithms. Therefore many generating algorithms output objects in an order such that each

M. Kaykobad and Md. Saidur Rahman (Eds.): WALCOM 2007, pp. 153–161, 2007.

object differs from the preceding one by a very small amount, and output each object as the "difference" from the preceding one. Such orderings of objects are known as *Gray codes* [J80,R93,R00,S97].

Let $G$ be a graph, where each vertex corresponds to each object and each edge connects two similar objects. Then the Gray code corresponds to a Hamiltonian path of $G$. For the set $LE(\mathcal{P})$ of all linear extensions of a given poset $\mathcal{P}$ we can also define such a graph $G$. However, the graph $G$ may not have a Hamiltonian path. Therefore the algorithm in [PR94] first constructs a new set $S^{'}$ so that if $x \in S$ then $+x, -x \in S^{'}$, then prove that the graph $G^{'}$ corresponds to $S^{'}$ always has a Hamiltonian path. Based on this idea, the algorithm in [PR94] generates the first linear extension in $O(m+n)$ time, then generates each linear extension in only $O(1)$ time on average along a Hamiltonial path of $G^{'}$. Note that the algorithm generates each linear extension exactly twice but output exactly one of them.

The paper [PR94] proposd the following question. Is there any algorithm to generate each linear extension in $O(1)$ time "in the worst case"? In this paper we answer the question affirmatively.

In this paper we give an algorithm to generate all linear extensions of $\mathcal{P}$. Our algorithm is simple and generates each linear extension in constant time in worst case (not on average). Our algorithm also outputs each linear extension as the difference from the preceding one. Thus our algorithm also generates a Gray code for linear extensions of a given poset.

The main idea of the algorithm is as follows. We first define a rooted tree (See Fig. 3.) such that each vertex corresponds to a linear extension of $\mathcal{P}$, and each edge corresponds to a relation between two linear extensions. Then by traversing the tree we generate all linear extensions of $\mathcal{P}$. With a similar technique we have already solved some generation problems for graphs[LN01,N02,NU03] and combinatorics[KN05]. In this paper we apply the technique for linear extensions.

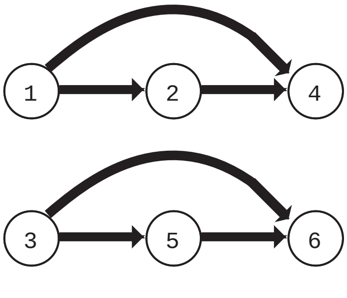

**Fig. 1.** An example of a poset $\mathcal{P}$.

The rest of the paper is organized as follows. Section 2 gives some definitions. Section 3 introduces the family tree. Section 4 presents our first algorithm. The algorithm generates each linear extension of a given poset $\mathcal{P}$ in $O(1)$ time on

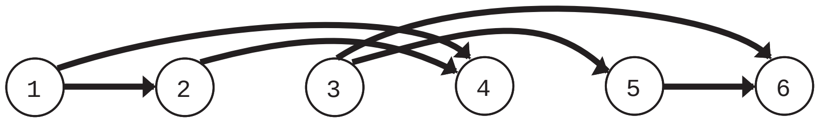

**Fig. 2.** A linear extension of a poset $\mathcal{P}$.

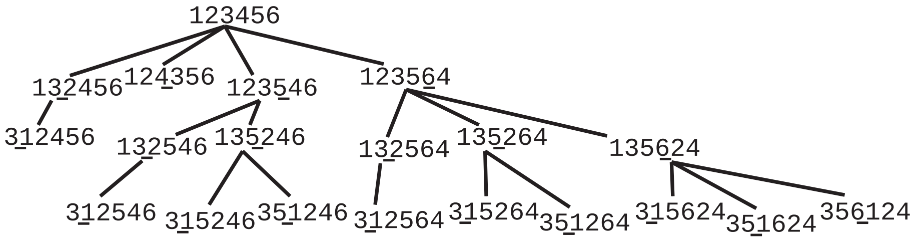

**Fig. 3.** The family tree of $L(\mathcal{P})$.

average. In Section 5 we improve the algorithm so that it generates each linear extension in $O(1)$ time in worst case. Finally Section 6 is a conclusion.

## 2 Preliminaries

*A poset* $P$ is a set $S$ with a binary relation $R$ which is reflexive, antisymmetric and transitive. Note that $R$ is a partial ordering on $S$. We denote $n = |S|$. We say $x$ precedes $y$ if $xRy$. We regard a poset P as a directed graph D such that (1) each vertex corresponds to an element in $S$, and (2) direct edge $(x, y)$ exists iff $xRy$. (See an example in Fig. 1.)

Given a poset $\mathcal{P} = (S, R)$, *a linear extension* of $\mathcal{P}$ is a permutaion ($x_1$, $x_2$, $\cdots$, $x_n$) of $S$, such that if $x_i R x_j$ then $i \leq j$. Intuitively, if we draw the directed graph corresponding to $\mathcal{P}$ so that $x_1, x_2, \cdots, x_n$ appear along a horizontal line from left to right in this order, then all directed edges go from left to right. (See an example in Fig. 2.)

## 3 The Family Tree

In this section we define a tree structure among linear extensions $LE(\mathcal{P})$ of a given poset $\mathcal{P}$.

Given a poset $\mathcal{P} = (\mathcal{S}, \mathcal{R})$, choose a linear extension $L_r \in LE(\mathcal{P})$. Without loss of generality we can assume that $S = \{1, 2, . \cdots, n\}$ and $L = (1, 2, . \cdots, n)$. (Otherwise, we rename the elements in $S$.) We call $L_r = (1, 2, . \cdots, n)$ *the root linear extension* of $\mathcal{P}$.

Then we define *the parent* $P(L)$ for each linear extension $L$ in $LE(\mathcal{P})$ (except for $L_r$) as follows. Let $L = (x_1, x_2, \cdots, x_n)$ be a linear extension of $\mathcal{P}$, and assume that $L \neq L_r$. Let $k$ be the minimum integer such that $x_k \neq k$. Since $L \neq L_r$

such $k$ always exists. We define *the level* of $L$ by $k$. For example the level of $(1, 2, 4, 3, 5, 6)$ is 3. For convenience we regard the level of $L_r$ to be $n + 1$.

By removing $k$ from $L$ then inserting the $k$ into the immediately before $x_k$ of $L$, we have a different permutation $L^{'}$. Note that $k$ has moved to the left, and so $L^{'} \neq L$. Now we have the following lemma.

**Lemma 1.** *$L^{'}$ is also a linear extensions of $\mathcal{P}$ in $LE(\mathcal{P})$.*

*Proof.* Assume otherwise. Then there must exists $x_i$ such that (1) $x_i R k$ and (2) $i > k$. However this contradicts the fact that $L_r = (1, 2, \cdots, n)$ is a linear extension of $\mathcal{P}$.) $\mathcal{Q.E.D.}$

We say that $L^{'}$ is *the parent linear extension* of $L$, and write $P(L) = L^{'}$. If $P(L)$ is the parent linear extension of $L$ then we say $L$ is *a child linear extension* of $P(L^{'})$. Note that $L$ has the unique parent linear extension $P(L)$, while $P(L)$ may have many child linear extensions.

We also have the following lemma.

**Lemma 2.** *Let $\ell$ and $\ell_p$ be the levels of a linear extension $L$ and its parent linear extension $P(L)$. Then $\ell < \ell_p$.*

*Proof.* Omitted.

The two lemmas above means the following. Given a linear extension $L$ in $LE(\mathcal{P})$ where $L \neq L_r$, by repeatedly finding the parent linear extension of the derived linear extension, we have the unique sequence $L, P(L), P(P(L)), \cdots$ of linear extensions in $LE(\mathcal{P})$, which eventually ends with the root linear extension $L_r$. Since the level is always increased, $L, P(L), P(P(L)), \cdots$ never lead into a cycle.

By merging these sequences we have *the family tree* of $LE(\mathcal{P})$, denoted by $T_{\mathcal{P}}$, such that the vertices of $T_{\mathcal{P}}$ correspond to the linear extensions in $LE(\mathcal{P})$, and each edge corresponds to each relation between some $L$ and $P(L)$. This proves that every linear extension appears in the tree as a vertex. For instance, $T_{\mathcal{P}}$ for a poset in Fig. 1 is shown in Fig. 3. In Fig. 3, the $k$ for each linear extension is underlined. By removing the $k$, then inserting it into the $k$-th position the parent linear extension is obtained.

## 4 Algorithm

In this section we give our generation algorithm. Given a poset $\mathcal{P}$, our algorithm traverses the family tree $T_{\mathcal{P}}$ and generates all linear extensions of $\mathcal{P}$.

If we can generate all child linear extensions of a given linear extension in $LE(\mathcal{P})$, then in a recursive manner we can construct $T_{\mathcal{P}}$, and generate all linear extensions $LE(\mathcal{P})$ of $\mathcal{P}$. How can we generate all child linear extensions of a given linear extension?

Given a linear extension $L = (p_1, p_2, p_3, \cdots, p_n)$ in $LE(\mathcal{P})$, let $\ell_p$ be the level of $L$. Let $C = (c_1, c_2, c_3, \cdots, c_n)$ be a child linear extension of $L$, and let $\ell_c$ be the level of $C$. By Lemma 2, the level of $C$ is smaller than the level of $L$. Thus $\ell_c < \ell_p$ holds. Therefore, for each $i = 1, 2, \cdots, \ell_p - 1$, if we generate all child linear extension of $L$ "having level $i$", then by merging those child linear extensions we can generate all child linear extensions of $L$.

Now, given $i$, $1 \leq i \leq \ell_p - 1$, we are going to generate all child linear extensions of $L = (p_1, p_2, \cdots, p_n)$ having level $i$. We can generate such child linear extensions by deleting $p_i = i$ from $L$ then insert it somewhere in $(p_{i+1}, p_{i+2}, \cdots, p_n)$ so that the resulting permutation is again a linear extension.

For example, see Fig. 3 for a poset $\mathcal{P}$ in Fig. 1. The last child $L = (1, 2, 3, 5, 6, 4)$ of the root linear extension has level 4. Thus each child linear extension has level either $1, 2$ or 3. For level 1, no child linear extensions of $L$ having level 1 exists, since $1R2$ means we cannot move 1 to the right. For level 2, child linear extensions of $L$ having level 2 are $L = (1, 3, 2, 5, 6, 4)$, $L = (1, 3, 5, 2, 6, 4)$ and $L = (1, 3, 5, 6, 2, 4)$. Note that $L = (1, 3, 5, 6, 4, 2)$ is not a linear extension because of $2R4$. For level 3, no child linear extensions of $L$ having level 3 exists, since $3R5$ means we cannot move 3 to the right.

We have the following algorithm.

```
Procedure find-all-children($L = (p_1 p_2 \cdots p_n)$,$\ell_p$)
{ $L$ is the current linear extension of $\mathcal{P}$.}
begin
  Output $L$ { Output the difference from the preceding one.}
  for $i = 1$ to $\ell_p$ - 1
  begin { generate children with level $i$ }
    j = i
    while $(p_j, p_{j+1}) \notin R$ do
    begin
      swap $p_j$ and $p_{j+1}$
      find-all-children($L = (p_1 p_2 \cdots p_n)$, $i$ )
      $j = j + 1$
    end
    insert $p_j$ into immediately after $p_{i-1}$
    { Now $p_i = i$ again holds, and $L$ is restored as it was.}
  end
end
```

```
Algorithm find-all-linear-extensions($\mathcal{P} = (S, R)$)
begin
  Find a linear extension $L_r$
  find-all-children($L_r$, n+1 )
end
```

For example, see Fig. 3. The last child $L = (1, 2, 3, 5, 6, 4)$ of the root linear extension has level 4. Assume we are going to generate child linear extensions having level 2. Since $p_2 = 2$, $p_3 = 3$ and $(2, 3) \notin R$, so we generate $L = (1, 3, 2, 5, 6, 4)$ by swapping $p_2$ and $p_3$. Then, since $p_3 = 2$, $p_4 = 5$ and $(2, 5) \notin R$, so we generate $L = (1, 3, 5, 2, 6, 4)$. Then, since $p_4 = 2$, $p_5 = 6$ and $(2, 6) \notin R$, so we generate $L = (1, 3, 5, 6, 2, 4)$. Then, since $p_5 = 2$, $p_6 = 4$ and $(2, 4) \in R$, so we do not generate $L = (1, 3, 5, 6, 4, 2)$.

Note that if $(p_i, p_{i+1}) \in R$, then $L$ has no child linear extensions having level $i$. Therefore if (1) $(p_a, p_{a+1}), (p_{a+1}, p_{a+2}), \cdots, (p_b, p_{b+1}) \in R$, and (2) the level of $L$ is $\ell_p > b$, then $L$ has no child linear extension with level $a, a+1, \cdots, b$. Then even if we execute Line 02 of the algorithm **find-all-children** several times, no linear extension is generated. Thus we cannot generate $k$ child linear extensions in $O(k)$ time.

However, we can preprocess the root linear extension and provide a simple list to solve this problem, as follows. First let $LIST = L_r = (1, 2, \cdots, n)$. For each $i = 1, 2, \cdots, n-1$, if $(p_i, p_{i+1}) \in R$, then we remove $p_i$ from $LIST$. Then the resulting $LIST$ tell us all levels at which at least one child linear extension exists. Using $LIST$ we can skip the levels at which no child linear extension exists.

For instance, see $T_{\mathcal{P}}$ in Fig. 3 for a poset in Fig. 1. For the root linear extension $L_r = (1, 2, 3, 4, 5, 6)$, $LIST = (2, 3, 4, 6)$. The last child $L = (1, 2, 3, 5, 6, 4)$ of $L_r$ has level 4.

Insted of generating all child linear extension at level $i$ for $i = 1, 2, \cdots, \ell_p - 1$ by the **for** loop in Line 02, we generate all child linear extensions at level $i$ only for each integer $i$ in $LIST$ up to $\ell_p - 1$. Thus now we can generate $k$ child linear extensions in $O(k)$ time.

**Theorem 1.** *The algorithm uses $O(n)$ space and runs in $O(|LE(\mathcal{P})|)$ time.*

*Proof.* Since we traverse the family tree $T_{\mathcal{P}}$ and output each linear extension at each corresponding vertex of $T_{\mathcal{P}}$, we can generate all linear extensions in $LE(\mathcal{P})$ without repetition.

Since we trace each edge of the family tree in constant time, the algorithm runs in $O(|LE(\mathcal{P})|)$ time.

The argument $L$ of the recursive call in Line 08 is passed by reference. Note that we restore $L$ as it was when return occurs.

The algorithm outputs each linear extension as only the difference from the preceding one. For each recursive call we need a constant amount of space, and the depth of the recursive call is bounded by $n$. Thus the algorithm uses $O(n)$ space. $\mathcal{Q.E.D.}$

## 5 Modification

The algorithm in Section 4 generates all linear extensions in $LE(\mathcal{P})$ in $O(|LE(\mathcal{P})|)$ time. Thus the algorithm generates each linear extension in $O(1)$ time "on average". However, after generating a linear extension corresponding to the last vertex in a large subtree of $T_{\mathcal{P}}$, we have to merely return from the deep recursive call without outputting any linear extension. This may take much time. Therefore, we cannot generate each linear extension in $O(1)$ time in worst case.

However, a simple modification [NU03] improves the algorithm to generate each linear extension in $O(1)$ time. The algorithm is as follows.

```
Procedure find-all-children2($L$, $depth$)
{ $L$ is the current sequence and $depth$ is the depth of the recursive call.}
begin
  if $depth$ is even
  then Output $L$ { before outputting its child.}
  Generate child linear extensions $L_1, L_2, \cdots, L_x$ by the method in Section 4, and
    recursively call find-all-children2 for each child linear extension.
  if $depth$ is odd
  then Output $L$ { after outputting its child.}
end
```

One can observe that the algorithm generates all linear extensions so that each sequence can be obtained from the preceding one by tracing at most three edges of $T_{\mathcal{P}}$. Note that if $L$ corresponds to a vertex $v$ in $T_{\mathcal{P}}$ with odd depth, then we may need to trace three edges to generate the next linear extension. Otherwise, we need to trace at most two edges to generate the next linear extension. Note that each linear extension is similar to the preceding one, since it can be obtained with at most three (delete then insert) operations. Thus, we can regard the derived sequence of the linear extensions as a combinatorial Gray code [J80,S97,R93,W89] for linear extensions.

## 6 Conclusion

In this paper we gave a simple algorithm to generate all linear extensions of a given poset. The algorithm is simple and generates each lenear extension in constant time in worst case. This solve an open question in [PR94].

We have another choice for the definition of the family tree for $LE(\mathcal{P})$ as follows. Given a linear extension $L = (c_1, c_2, \cdots, c_n) \neq L_r$ in $\mathcal{P}$, let $k$ be the level of $L$. Then let $i$ be the index such that $c_i = k$. By definition $i > k$ holds. Now be swapping $c_i$ with its left neighbour we obtain another linear extension $P(L)$, and we say $P(L)$ is the parent of $L$. Based on this parent-child relation we can define another family tree for $LE(\mathcal{P})$. For instance see Fig. 4. Based on this family tree, in a similar manner, we can design another simple algorithm to generate all linear extensions of a given poset. Note that each linear extension is also similar to the preceding one. The next linear extension can be obtained with at most three "adjacent transposition" operations.

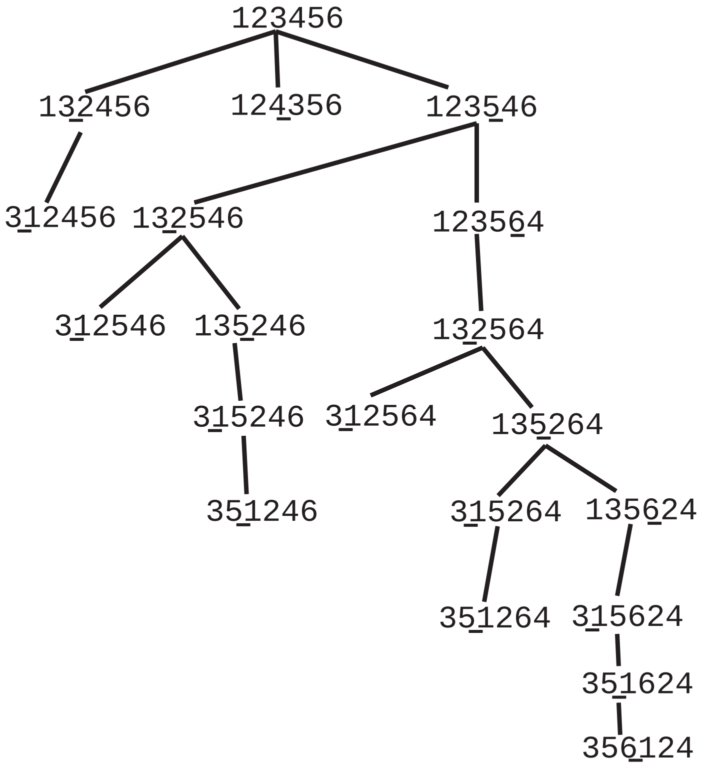

**Fig. 4.** Another family tree of $L(\mathcal{P})$.

## References

[C01] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, MIT Press, (2001).

[G93] L. A. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge University Press, New York, (1993).

[J80] J. T. Joichi, D. E. White and S. G. Williamson, *Combinatorial Gray Codes*, SIAM J. on Computing, 9, (1980), pp.130-141.

[KS98] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms*, CRC Press, Boca Raton, (1998).

[KV83] A. D. Kalvin and Y. L. Varol, *On the Generation of All Topological Sortings*, J. of Algorithms, 4, (1983), pp.150–162.

[KN05] S. Kawano and S. Nakano, *Constant Time Generation of Set Partitions*, IEICE Trans. Fundamentals, accepted, to appear, (2005).

[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.

[NU03] S. Nakano and T. Uno, *A Simple Constant Time Enumeration Algorithm for Free Trees*, IPSJ Technical Report, 2003-AL-91-2, (2003). (http://www.ipsj.or.jp/members/SIGNotes/Eng/16/2003/091/article002.html)

[PR94] G. Pruesse and F. Ruskey, *Generating Linear Extensions Fast*, SIAM Journal on Computing, 23, (1994), pp.373–386.

[R78] R. C. Read, *How to Avoid Isomorphism Search When Cataloguing Combinatorial Configurations*, Annals of Discrete Mathematics, 2, (1978), pp.107–120.

[R93] F. Ruskey, *Simple Combinatorial Gray Codes Constructed by Reversing Sublists*, Proc. ISAAC93, LNCS 762, (1993), pp.201–208.

[R00] K. H. Rosen (Eds.), *Handbook of Discrete and Combinatorial Mathematics*, CRC Press, Boca Raton, (2000).

[S97] C. Savage, *A Survey of Combinatorial Gray Codes*, SIAM Review, 39, (1997) pp. 605-629.

[St97] R. Stanley, *Enumerative Combinatorics Volume 1*, Cambridge University Press, (1997).

[W89] H. S. Wilf, *Combinatorial Algorithms : An Update*, SIAM, (1989).