An $O(1)$ Time Algorithm for Generating Multiset Permutations

Tadao Takaoka

Department of Computer Science, University of Canterbury
Christchurch, New Zealand
tad@cosc.canterbury.ac.nz

Abstract. We design an algorithm that generates multiset permutations in $O(1)$ time from permutation to permutations, using only data structures of arrays. The previous $O(1)$ time algorithm used pointers, causing $O(n)$ time to access an element in a permutation, where $n$ is the size of permutations. The central idea in our algorithm is tree traversal. We associate permutations with the leaves of a tree. By traversing this tree, going up and down and making changes when necessary, we spend $O(1)$ time from permutation to permutation. Permutations are generated in a one-dimensional array.

1 Introduction

Algorithms for generating combinatorial objects, such as (multiset) permutations, (multiset) combinations, well-formed parenthesis strings are a well studied area and many results are documented in Nijenhuis and Wilf [6], and Reingold, Nievergelt, and Deo [8], etc.

Let $n$ be the size of the objects to be generated. The most primitive algorithms are recursive ones for generating those objects in lexicographic order, causing $O(n)$ changes from object to object, and thus $O(n)$ time. To overcome this drawback, many algorithms were invented, which generate objects with a constant number of changes, $O(1)$ changes, from object to object. This idea of generating combinatorial objects with $O(1)$ changes is named "combinatorial Gray codes", and a good survey is given in [11]. In many cases, these changes are made by swappings of two elements, that is, two changes. It is still easy to design recursive algorithms for combinatorial generation with $O(1)$ changes, since we can control the paths of the tree of recursive calls and thus we can rather easily identify changing places. Note that combinatorial objects correspond to the leaves of the tree, meaning that it takes $O(n)$ time from object to object as the height of the tree is $n$. Further to overcome this shortcoming, several attempts were made to design iterative algorithms, which are called loopless algorithms in some literature, removing recursion, so that $O(1)$ time is achieved from object to object. At this stage, we need some care in defining the $O(1)$ time from object to object. In Korsh and Lipschutz [3], $O(1)$ time was achieved to generate multiset permutations, whose algorithm is a refinement of that by Hu and Tien.
In this algorithm, multiset permutations are given one after another in a linked list. The operations on the list are manipulated by pointers, involving shift operations in $O(1)$ time. For example, the list $(1, 1, 1, 2, 2, 2)$ with $n = 6$ can be converted to $(2, 2, 2, 1, 1, 1)$ in $O(1)$ time by changing pointers. We assume that the above conversion takes $O(n)$ time in this paper, and we claim that multiset permutations can be generated in $O(1)$ time using arrays, not pointers.

This kind of strict requirement for $O(1)$ time was demonstrated in the recent development in parenthesis strings generation. An $O(1)$ change algorithm was developed in Ruskey and Proskurowski [10] and an $O(1)$ time algorithm with pointer structures was achieved in Roelants van Baronaigien [9], and they challenged the readers, asking whether there could be $O(1)$ algorithms with arrays, whereby stricter $O(1)$ time could be achieved. This problem was recently solved by three independent works of Mikawa and Takaoka [5], Vajnovski [13], and Walsh [14]. Note that we can access any element of a combinatorial object in $O(1)$ time in array implementation, whereas we need $O(n)$ time in linked list implementation, as we must traverse the pointer structure. The algorithm by Ko and Ruskey [2] generates multiset permutations with swappings of two elements, but not with $O(1)$ time from permutation to permutation.

The main idea of $O(1)$ time for multiset permutation generation in this paper is tree traversal. The generation tree for a set of permutations, arranged in some order, on the given multiset is a tree whose paths to the leaves correspond to the permutations. Basically we traverse the tree in movements of (up, cross, down). The move ”up” is to go up the tree from a node to one of its ancestors. The move ”cross” is to move from a node to its adjacent sibling, causing a swapping with the element at that level and the one at a level closer to the leaf. The move ”down” is to go down from a node to one of its descendents, which we call the landing point. The landing point has no sibling and the path to the leaf has no branching, causing a straight line. It is important that we avoid traversing this straight line node by node. The core part of the algorithm is centered on how to compute the positions to which we go up and down, and where we should perform swappings. Although the use of tree structure for combinatorial generation was originated in Lucas [4] and Zerling [15], and well known, the technique of tree traversal in this paper is new.

Since the final algorithm is rather complicated, we go through a stepwise refinement process, going from simple structures to details. In Section 2, we define the generation tree and design a recursive algorithm that traverses this tree to generate multiset permutations. We give a formal proof of the recursive algorithm. In Section 3, we design an iterative algorithm based on the recursive algorithm. We first describe an informal framework for an iterative algorithm, and translate the recursive algorithm into an iterative one guided by the framework. The resulting iterative algorithm generates multiset permutations in $O(1)$ time in a one-dimensional array. As additional data structures, we use a few more arrays, causing $O(kn)$ space requirement, where $k$ is the number of distinct elements in the multiset. In Section 4, we give concluding remarks.