Discrete Applied Mathematics 217 (2017) 722–728

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

Note

Linear-time generation of uniform random derangements encoded in cycle notation

Kenji Mikawa[a,*], Ken Tanaka[b]

[a] *Center for Academic Information Service, Niigata University, 8050 Ikarashi 2-no-cho, Nishi-ku, Niigata 950-2181, Japan*
[b] *Faculty of Science, Kanagawa University, 2946 Tsuchiya, Hiratsuka, Kanagawa 259-1293, Japan*

ARTICLE INFO

Article history:
Received 24 September 2014
Received in revised form 27 May 2015
Accepted 1 October 2016
Available online 22 October 2016

Keywords:
Combinatorial problem
Algorithm
Random generation
Linear-time
Derangement

ABSTRACT

We present a linear-time algorithm for generating random derangements. Several algorithms for generating random derangements have recently been published. They are enhancements of the well-known Fisher–Yates shuffle for random permutations, and use the rejection method. The algorithms sequentially generate random permutations, and therefore pick only derangements by omitting the other permutations. A probabilistic analysis has shown that these algorithms could be run in an amortized linear-time. Our algorithm is the first to achieve an exact linear-time generation of random derangements. We use a computational model such that arithmetic operations and random access for $O(\log n!) = O(n \log n)$ bits can be executed in constant time.

1. Introduction

A derangement over n integers $[n] = \{1, 2, \ldots, n\}$ is defined as a permutation $\delta = \delta(1)\delta(2)\ldots\delta(n)$ of the integers, without fixed points, i.e., $\delta(i) \neq i$ for all $i \in [n]$. The subfactorial $!n$ is the number of possible derangements. It is given by the recurrence relation

$$!n = (n-1)\left(!(n-1) + !(n-2)\right), \qquad (1)$$

with the initial conditions $!0 = 1$ and $!1 = 0$. We can derive a recurrence relation from Eq. (1), that is,

$$!n = n \times !(n-1) + (-1)^n. \qquad (2)$$

The subfactorial also satisfies $!n = \lfloor (n! + 1)/e \rfloor$, where e is Napier's constant.

Durstenfeld proposed an excellent implementation of the Fisher–Yates shuffle, which is one of the best-known algorithms for generating random permutations [1]. Whereas a naive implementation of the Fisher–Yates shuffle takes $O(n^2)$ time to generate one random permutation, Durstenfeld's implementation realizes $O(n)$ time to generate one random permutation. The stringent condition that derangements have no fixed points complicates linear-time algorithms for generating random derangements, in contrast with linear-time algorithms for generating random permutations. Existing algorithms have enhanced the Fisher–Yates shuffle to suit this problem, using the rejection method to sequentially generate random permutations, thus picking up only derangements by omitting other permutations [3,2]. The probability that a random permutation is a derangement is expected to be close to $1/e$. In fact, a probabilistic analysis has shown that both the proposed algorithms [3,2] should run in an amortized linear-time.

* Corresponding author.
E-mail addresses: mikawa@cais.niigata-u.ac.jp (K. Mikawa), ktanaka@info.kanagawa-u.ac.jp (K. Tanaka).

http://dx.doi.org/10.1016/j.dam.2016.10.001
0166-218X/© 2016 Elsevier B.V. All rights reserved.

In our recent report [4], we proposed a lexicographic ranking and unranking of derangements encoded in cycle notation. The unranking algorithm takes $O(n \log n)$ time to generate one derangement for a given random integer. The time complexity is estimated on the traditional computational model, such that arithmetic operations and random access with $O(\log x)$ bits for input size x can be done in $O(1)$ time. Our algorithm requires arithmetic on large integers to calculate a probability of belonging to two classes of derangements. The large integers in the algorithm are at most $!n = \lfloor (n! + 1)/e \rfloor$. We use a computational model such that arithmetic operations and random access with $O(\log !n) = O(n \log n)$ bits can be done in $O(1)$ time. Hence, we achieve a linear-time generation of uniform random derangements under this model.

2. Uniformly random derangement generation

Cycle notation is an intuitive way to describe the order of a permutation that gives a mapping from $[n]$ to $[n]$ as a list of disjoint cycles. Stanley introduced the *standard representation* for this notation, and described some of its properties in [5]. In our proposed algorithm, we deal with permutations which are written in standard representation with *right-to-left* minima. For example, given the permutation $\pi = 5743126$, the standard representation of π is $\sigma = 5176243$. For derangements, we observe the property that there are no two consecutive right-to-left minima and the first value cannot be 1. The linear-time complexity of converting between a permutation and its standard representation is also well known.

We explain the Fisher–Yates shuffle introduced by Durstenfeld in [1]. The Fisher–Yates shuffle for $\pi = \pi(1)\pi(2)\dots\pi(n)$ on $[n]$ is shown below. Initially, $\pi = 12\dots n$. The function rand(i) outputs a uniform random integer between 1 and i.

```
**for** $i := 1$ **downto** $n - 1$ **do begin**
  $j := \text{rand}(n - i + 1) + i - 1$;
  $\text{swap}(\pi(i), \pi(j))$;
**end**;
```

Once $\pi(i)$ is fixed, the algorithm will never alter the prefix $\pi(1)\dots\pi(i)$ in subsequent steps.

Our approach is to specialize the Fisher–Yates shuffle as applied to random derangements encoded in cycle notation. A candidate element for the position $\sigma(i)$ exists in the remaining elements $\{\sigma(i), \dots, \sigma(n)\}$, but the candidates do not have equal probabilities $1/(n - i + 1)$. When a cycle in the prefix $\sigma(1)\dots\sigma(i-1)$ closes at $\sigma(i)$ (which implies that $i > 1$ and no cycles close at $\sigma(i-1)$), the minimum element must be uniquely selected from $\{\sigma(i), \dots, \sigma(n)\}$. Otherwise, when a cycle does not close at $\sigma(i)$, an element can be selected from the $n - i$ possible elements in $\{\sigma(i), \dots, \sigma(n)\}$, except for the minimum. The $n - i$ possible elements have an equal probability of $1/(n - i)$. Thus, the probability of selecting the minimum element for the position $\sigma(i)$ is given by

$$P(A_i) = \frac{!(n-i)}{!(n-i) + !(n-i+1)}, \tag{3}$$

under the conditions that a cycle does not close at $\sigma(i-1)$, where $i > 1$. The outline of our algorithm is shown below. We use a Boolean flag, *closed*, to determine if a cycle closes at the previous step. The function $\sigma^{-1}(x)$ returns the position of a given element x, and the function $\min(X)$ returns the minimum element in a given set X. Initially, $\sigma = 12\dots n$.

```
A 1: closed := **true**;
A 2: **for** $i := 1$ **to** $n - 1$ **do begin**
A 3:   **if** $(i > 1)$ **and** (closed = **false**) **and**
              $\text{rand}(!(n-i) + !(n-i+1)) \le !(n-i)$ **then begin**
         // some cycle closes at $\sigma(i)$.
A 4:     $j := \sigma^{-1}(\min(\{\sigma(i), \sigma(i+1), \dots, \sigma(n)\}))$;
A 5:     $\text{swap}(\sigma(i), \sigma(j))$;
A 6:     closed := **true**;
A 7:   **end else begin**
         // otherwise, a cycle does not close at $\sigma(i)$.
A 8:     $j := \text{rand}(n - i) + i - 1$;
A 9:     **if** $\sigma(j) = \min(\{\sigma(i), \sigma(i+1), \dots, \sigma(n)\})$
           **then** $\text{swap}(\sigma(i), \sigma(n))$ **else** $\text{swap}(\sigma(i), \sigma(j))$;
A10:     closed := **false**;
A11:   **end**;
A12: **end**;
```

The algorithm contains a tricky swap process in lines A8 and A9. This process corresponds to selecting a non-minimum element among the $n - i$ possible elements in $\{\sigma(i), \dots, \sigma(n)\}$. It is equivalent to selecting an element from the set $\{\sigma(i), \dots, \sigma(n)\} \setminus \{\min(\{\sigma(i), \dots, \sigma(n)\})\}$. To achieve a linear-time generation, the time taken to calculate $\min(X)$ and the probability of selecting the minimum element for the position $\sigma(i)$ must be constant for each loop.

Our strategy to find the minimum element of a given set is based on a linked list using three arrays: used$[i]$, which stores a Boolean value indicating whether an element $i \in [n]$ is used or not, head$[i]$, which is a pointer to the head of successive used