Fast Algorithms to Enumerate All Common Intervals of Two Permutations

Takeaki Uno[♠] and Mutsunori Yagiura[♣]

Abstract Given two permutations of n elements, a pair of intervals of these permutations consisting of the same set of elements is called a *common interval*. Some genetic algorithms based on such common intervals have been proposed for sequencing problems and have exhibited good prospects. In this paper, we propose three types of fast algorithms to enumerate all common intervals: i) a simple $O(n^2)$ time algorithm (LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations, ii) a practically fast $O(n^2)$ time algorithm (MNG) using the reverse Monge property, and iii) an $O(n+K)$ time algorithm (RC), where K ($\leq \binom{n}{2}$) is the number of common intervals. It will be also shown that the expected number of common intervals for two random permutations is $O(1)$. This result gives a reason for the phenomenon that the expected time complexity $O(n)$ of the algorithm LHP is independent of K. Among the proposed algorithms, RC is most desirable from the theoretical point of view; however, it is quite complicated compared to LHP and MNG. Therefore, it is possible that RC is slower than the other two algorithms in some cases. For this reason, computational experiments for various types of problems with up to $n = 10^6$ are conducted. The results indicate that i) LHP and MNG are much faster than RC for two randomly generated permutations, and ii) MNG is rather slower than LHP for random inputs; however, there are cases that LHP requires $\Omega(n^2)$ time, but MNG runs in $o(n^2)$ time and is faster than both LHP and RC.

Keywords: common intervals of permutations, genetic algorithm, linear time algorithm, random permutations, Monge property, subtour exchange crossover.

[♠]Department of Systems Science, Tokyo Institute of Technology, 2-12-1 Oh-okayama, Meguro-ku, Tokyo 152, Japan. E-mail: uno@is.titech.ac.jp

[♣]Corresponding author: Department of Applied Mathematics and Physics, Graduate School of Engineering, Kyoto University, Kyoto 606-01, Japan. Phone: +81-75-753-5514, Fax: +81-75-761-2437, E-mail: yagiura@kuamp.kyoto-u.ac.jp

1 Introduction

Two permutations σ_A and σ_B of set $N = \{1, \ldots, n\}$ are given as the input, where $\sigma_A(i) = j$ (or $\sigma_A^{-1}(j) = i$) denotes that j is the i-th element of σ_A (σ_B is similarly defined). Let $[x, y]$ denote the index set $\{x, x+1, \ldots, y\}$. We call a pair of intervals $([x_A, y_A], [x_B, y_B])$ $(1 \leq x_A < y_A \leq n, 1 \leq x_B < y_B \leq n)$ a *common interval* if it satisfies

$$\{\sigma_A(i) \mid i \in [x_A, y_A]\} = \{\sigma_B(i) \mid i \in [x_B, y_B]\}. \tag{1}$$

The length of a common interval $([x_A, y_A], [x_B, y_B])$ is defined to be $y_A - x_A + 1$.

Some genetic algorithms based on common intervals have been proposed for sequencing problems (e.g., traveling salesman problem, single machine scheduling problem, etc.) and have exhibited good prospect [1, 2, 3, 4].

In this paper, we consider enumeration of all common intervals of length 2 to n. Three algorithms are proposed, which are improved versions of a simple $O(n^2)$ time algorithm proposed in [5]:

1. A simple $O(n^2)$ time algorithm (called LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations.

2. A practically fast $O(n^2)$ time algorithm (called MNG) using the reverse Monge property.

3. An $O(n + K)$ time algorithm (called RC), where K $(\leq \binom{n}{2})$ is the number of outputs.

It will be also shown that the expected number of common intervals of length 2 to $n-2$ for two random permutations is $2 + O(n^{-1})$. This implies that the expected number of common intervals of length 2 to n is $O(1)$, since the number of common intervals of length $n-1$ or n is at most 3. This result gives a reason for the phenomenon that the expected time complexity $O(n)$ of the algorithm LHP is independent of K. We also give an example for which both LHP and MNG requires $\Omega(n^2)$ time, although $K = O(n)$.

Among the three algorithms proposed in this paper, RC is most desirable from the theoretical point of view; however, it is quite complicated compared to LHP and MNG. Therefore, it is possible that RC is slower than the other two algorithms in some cases. For this reason, computational

experiments for various types of problems with up to $n = 10^6$ are conducted. The results indicate that

1. LHP and MNG are much faster than RC for two randomly generated permutations (e.g., LHP is about 13 times faster than RC).

2. MNG is rather slower than LHP for random inputs; however, there are cases that LHP requires $\Omega(n^2)$ time, but MNG runs in $o(n^2)$ time and is faster than both LHP and RC.

A recommendation about the use of the three algorithms is discussed in Section 7 based on the computational results.

These results are applicable to the similar problem defined on two cyclic permutations [5, 6].

2 Basic Algorithm

Here, we describe the basic $O(n^2)$ time algorithm [5], which is the starting point of all the algorithms proposed in this paper. For convenience, we denote $\sigma_B^{-1} \cdot \sigma_A$ by π_{AB} (i.e., $\pi_{AB}(i) = \sigma_B^{-1}(\sigma_A(i))$ holds for all i, and $\pi_{AB}(i) = j$ means that the i-th element of σ_A is located in the j-th position of σ_B) throughout this paper, which can be calculated from σ_A and σ_B in $O(n)$ time. We also define the following functions for an interval $[x, y]$ of σ_A:

$$l(x,y) = \min_{i \in [x,y]} \pi_{AB}(i) \tag{2}$$

$$u(x,y) = \max_{i \in [x,y]} \pi_{AB}(i) \tag{3}$$

$$f(x,y) = u(x,y) - l(x,y) - (y - x). \tag{4}$$

Since $f(x,y)$ is the number of elements in $\{\sigma_B(i) \mid i \in [l(x,y), u(x,y)]\} \backslash \{\sigma_A(i) \mid i \in [x,y]\}$, a pair $([x,y],[l(x,y),u(x,y)])$ is a common interval if and only if $f(x,y) = 0$. Then all common intervals can be enumerated by calculating $f(x,y)$ for all (x,y) pairs satisfying $1 \leq x < y \leq n$. This gives rise to the following algorithm.

Algorithm BSC

Line 1: **for** $x = 1, \ldots, n-1$ **do**

Line 2: $l{:=}u{:=}\pi_{AB}(x)$;

Line 3: **for** $y = x+1, \ldots, n$ **do**

Line 4: $l{:=}\min\{l, \pi_{AB}(y)\}$;

Line 5: $u{:=}\max\{u, \pi_{AB}(y)\}$;

Line 6: **if** $u - l - (y - x) = 0$ **then**

Line 7: output $([x, y], [l, u])$

Line 8: **end for**

Line 9: **end for**.

The variables u and l in BSC correspond to the function values $u(x,y)$ and $l(x,y)$ defined above. The time complexity of this algorithm is $O(n^2)$, since Lines 4, 5, 6 and 7 can be executed in $O(1)$ time.

3 Simple Improvements of the Basic Algorithm

In this section, we propose two improved versions of BSC, called LHP and MNG, both of which detect some redundant inner loop iterations from Line 3 to 8 of BSC by simple tests, and remove them from execution. They still require $O(n^2)$ time in the worst case; however, it is observed that they are practically much faster than BSC for many types of problems.

3.1 The Algorithm LHP

Here we describe the algorithm LHP. It is shown in Section 5 that the expected running time of this algorithm for two randomly generated permutations is $O(n)$. For convenience, only the common intervals of length 2 to $n-2$ are considered in this subsection, and Line 3 of BSC is modified as

"Line 3′: **for** $y = x+1, \ldots, \min\{n, x+n-3\}$ **do**".

Modification of the algorithm to the original problem (where common intervals of length 2 to n are considered) is easy and the results of this paper are not affected by this assumption by the following reasons. The pair of intervals of length n (i.e., $([1,n],[1,n])$) is always a common interval. There are four pairs of intervals, $([1,n-1],[1,n-1])$, $([1,n-1],[2,n])$, $([2,n],[1,n-1])$ and $([2,n],[2,n])$, which are the candidates for common intervals of length $n-1$. The pair of intervals $([1,n-1],[1,n-1])$ is a common interval if and only if $\pi_{AB}(n) = n$. The other cases are similar. Therefore, we can enumerate all common intervals of length $n-1$ in constant time by checking if $\pi_{AB}(1) = 1$, $\pi_{AB}(1) = n$, $\pi_{AB}(n) = 1$ or $\pi_{AB}(n) = n$ holds. We improve the basic algorithm BSC in the following two respects.

The first is that, if

$$u - l > \min\{n - x, n - 3\} \tag{5}$$

is satisfied just before entering Line 6 of BSC in the x-th iteration, then the rest of current inner loop can be omitted, and we move into the $(x+1)$st iteration immediately. Note that $u - l$ is monotonically nondecreasing during the x-th iteration. Condition (5) implies that the length of interval $[l,u]$ of σ_B exceeds the maximum length of interval $[x,y]$ of σ_A when y is increased up to $\min\{n, x+n-3\}$. We call this condition *length condition*.

Let HP be the set

$$\begin{aligned} HP &= N \backslash \{\pi_{AB}(w) \mid w = x, x+1, \ldots, \min\{n, x+n-3\}\} \\ &= \{\pi_{AB}(w) \mid w \in [1, x-1] \cup \{z \in [1,n] \mid z \equiv x-2 \pmod{n} \text{ or } z \equiv x-1 \pmod{n}\}\}. \end{aligned}$$

The second is that, if an $h \in HP$ satisfies

$$l < h < u \tag{6}$$

just before entering Line 6 of BSC, then the rest of current inner loop can be omitted. HP is the set of indices of the elements which will not be included in any interval $[x,y]$ $(y = x+1, \ldots, \min\{n, x+n-3\})$ of σ_A. We call each element of HP a *hole point*, and call condition (6) *HP condition*. It is not advantageous to check the HP condition for all $h \in HP$, since the whole running time increases to $O(n^3)$. Hence, we check the HP condition for only a sufficiently small portion of HP, which we

call HP', so that the original worst case time complexity $O(n^2)$ is preserved. For this, $|HP'|$ should be kept constant. After trying several in preliminary computational experiments, we choose HP' as follows:

$$HP' = \{\pi_{AB}(w) \mid w \in \{z \in [1,n] \mid z \equiv x-2 \pmod{n} \text{ or } z \equiv x-1 \pmod{n}\}\}. \tag{7}$$

As natural candidates, one may consider

$$\begin{aligned} HP_1 &= \{\pi_{AB}(w) \mid w \in [1,n] \text{ and } w \equiv x-1 \pmod{n}\} \text{ or} \\ HP_2 &= \{\text{an element randomly chosen from } HP\}. \end{aligned} \tag{8}$$

However, it is observed that $O(n \log n)$ average time is needed for two randomly generated permutations if we use HP_1, and it is also observed that the algorithm becomes slower if we use HP_2 (one of the conceivable reasons for this phenomenon is that generating random values frequently is too expensive). More discussion is in [6].

3.2 The Algorithm MNG

Here we describe the second algorithm MNG. It uses the fact that the function f defined by (4) satisfies reverse Monge property, that is,

$$f(x',y) + f(x,y') \geq f(x',y') + f(x,y) \tag{9}$$

holds for all x', x, y, y' satisfying $x' < x \leq y < y'$ (see Appendix A for the proof). From (9), we have

$$\begin{aligned} f(x,y') &\geq f(x,y) - \{f(x',y) - f(x',y')\} \\ &\geq f(x,y) - \{f(x',y) - \min_{z \in [y+1,n]} f(x',z)\}. \end{aligned} \tag{10}$$

Since the above inequalities hold for every x' $(< x)$,

$$f(x,y') \geq f(x,y) - \min_{w \in [1,x-1]} \{f(w,y) - \min_{z \in [y+1,n]} f(w,z)\} \tag{11}$$

holds. The value of $\min_{w \in [1,x-1]}\{f(w,y) - \min_{z \in [y+1,n]} f(w,z)\}$ gives an upper bound for the decrease of $f(x,y)$ when y is increased up to n. Hence, if $x \geq 2$ and

$$f(x,y) - \min_{w \in [1,x-1]} \{f(w,y) - \min_{z \in [y+1,n]} f(w,z)\} > 0 \tag{12}$$

holds just before entering Line 8 of BSC in the x-th iteration, then the rest of current inner loop can be omitted, and we can move to the $(x+1)$st iteration immediately. Here y_{last} is defined as the value of y at Line 9 when we exit the inner loop. If $y_{last} \leq n-1$, then we will not complete computing $\min_{z\in[y_{last}+1,n]} f(x,z)$. Hence, we may fail to check condition (12) for larger x. Thus we define a function

$$LD(x,y) = \begin{cases} \infty, & (x=1, y=2,3,\ldots,n-1) \\ \min\{LD(x-1,y), f(x-1,y) - \min\limits_{z\in[y+1,n]} f(x-1,z)\}, & \\ & (x\geq 2, y=x,\ldots,y_{last}-1, y_{last}=n) \\ \min\{LD(x-1,y), f(x-1,y) - \min\{\min\limits_{z\in[y+1,y_{last}]} f(x-1,z), & \\ \quad f(x-1,y_{last}) - LD(x-1,y_{last})\}\}, & \\ & (x\geq 2, y=x,\ldots,y_{last}, y_{last}\leq n-1) \\ LD(x-1,y), & (x\geq 2, y=y_{last}+1,\ldots,n-1, y_{last}\leq n-1). \end{cases} \tag{13}$$

The function $LD(x,y)$ can be calculated even if $y_{last} \leq n-1$ and satisfies

$$LD(x,y) \geq \min_{w\in[1,x-1]} \{f(w,y) - \min_{z\in[y+1,n]} f(w,z)\}. \tag{14}$$

An inner loop can be terminated if condition

$$f(x,y) - LD(x,y) > 0 \tag{15}$$

holds. The correctness of the algorithm is retained, since condition (15) implies condition (12). We call condition (15) *Monge condition*.

We defined LD as a function of both x and y for convenience; however, the value of $LD(x,y)$ can be overwritten on the same memory space with $LD(x-1,y)$ in practice. Such an update of LD is executed every time we exit the inner loop, which is possible in $O(y_{last}-x)$ time. Hence, the worst case running time $O(n^2)$ of the algorithm BSC is preserved for MNG.

We further set a parameter $R \in (0,1]$, and do not exit the inner loop for $y > R(n-x)+x$ even if Monge condition is satisfied. ($R=1$ means the case we do not use this modification.) Once $y > R(n-x)+x$ holds, y_{last} is forced to be n and we can update LD by using the second formula of (13); hence, LD value may improve. The total time spent to inner loops increases at most $1/R$

times compared to the case with $R = 1$. We set R to 0.5 in the computational experiments, since remarkable improvement was observed in some problem instances compared to $R = 1$.

3.3 Remarks about the Two Algorithms

Two algorithms LHP and MNG can be combined; however, slight modifications are needed to the way of updating LD. Using a parameter $R \in (0, 1]$, that is, an inner loop is terminated by length condition, HP condition or Monge condition only for $y < R(n - x) + x$, will be useful. Since the computational time gains at most $1/R$ times, expected running time of this combined algorithm is $O(n)$ for two randomly generated permutations. It is also noted that some LD values may become larger than when MNG is used alone, and this combined algorithm will not necessarily improve the performance of MNG.

Although it is observed that algorithms of this type are much faster than the algorithm BSC for many types of problems, they always require $\Omega(n^2)$ time for some problem instances. For example, consider the problem given by setting $\sigma_A(i) = i$ $(i = 1, \ldots, n)$ and

$$\sigma_B(i) = \begin{cases} 2i - 1, & i \leq \lceil n/2 \rceil \\ 2(n - i + 1), & i \geq \lceil n/2 \rceil + 1. \end{cases}$$

The function f takes

$$\begin{aligned} f(x, y) &> 0,\ y = x + 1, \ldots, n - 1,\ x = 1, \ldots, n - 1 \\ f(x, n) &= 0,\ x = 1, \ldots, n - 1 \end{aligned}$$

and the number of outputs $K = O(n)$. Any algorithm improved from BSC by "omitting redundant loops" requires $\Omega(n^2)$ time for this example. It shows a limitation of the algorithms of this type.

4 An Algorithm with $O(n + K)$ Worst Case Running Time

In this section, we propose an algorithm called the *reduce candidate algorithm* (abbreviated as RC) which runs in $O(n + K)$ time in the worst case. Since the algorithm runs in time proportional to the number of inputs and outputs, it is optimal in the sense of time complexity. On the other hand,

those algorithms proposed in the previous section may take much time, e.g., $\Omega(n^2)$ time even if the number of outputs K is $O(n)$, though they are very simple and fast for most of the tested problem instances.

For a fixed x, we call a y *wasteful* if it satisfies $f(x', y) > 0$ for all $x' \leq x$. The main idea of the algorithm RC is to save the time to check whether $f(x, y) = 0$ or not for some y which can be concluded as wasteful from the past search information. The framework of the algorithm is described as follows.

Algorithm RC

Line 1: $Y := N$.

Line 2: **for** $x = n - 1, \cdots, 1$ **do**

Line 3: Output all y $(> x)$ in Y satisfying $f(x, y) = 0$.

Line 4: Set $Y := Y \backslash W$ where $W \subseteq \{y \in N \mid y \geq x \text{ and } f(x', y) > 0 \text{ for all } x' < x\}$.

Line 5: **end for.**

The key to this algorithm is the way to find wasteful ys. The following lemmas help us to identify them.

Lemma 4.1 *Suppose that we are given $x > 1$ and $y > x$. If $u(x, y) < u(x, y')$ and $u(x - 1, y) = u(x - 1, y')$ hold for some $y' > y$, y satisfies $f(x', y) > 0$ for all $x' < x$.*

Proof. From $u(x, y) < u(x, y')$, there exists a $y'' \in [y + 1, y']$ satisfying $\pi_{AB}(y'') \in [u(x, y) + 1, u(x, y')]$. By $u(x - 1, y) = u(x - 1, y')$, we have $[u(x, y) + 1, u(x, y')] \subseteq [l(x', y), u(x', y)]$ and $\pi_{AB}(y'') \in [l(x', y), u(x', y)]$. As y'' is not included in $[x', y]$, $f(x', y)$ is greater than 0. □

Lemma 4.2 *Suppose that we are given $x > 1$ and $y > x$. If $f(x, y) > f(x, y')$ hold for some $y' > y$, y satisfies $f(x', y) > 0$ for all $x' \leq x$.*

Proof. From $f(x, y) > f(x, y')$, there exists a $y'' \in [y+1, y']$ which satisfies $\pi_{AB}(y'') \in [l(x, y), u(x, y)]$. Since y'' is not included in $[x', y]$, $f(x', y)$ is greater than 0. □

We can find a part of wasteful y from these properties. We will show an algorithm that removes every y that satisfies the conditions of Lemma 4.1 or 4.2 from the set Y at Line 4 of the algorithm RC.

To maintain Y, the algorithm uses a doubly linked list called *ylist* composed of cells $y_1, ..., y_r$ corresponding to each $y \in Y$. The cells are sorted in increasing order of their values. For convenience, we consider only the case with $\pi_{AB}(x-1) > \pi_{AB}(x)$ throughout this section. The opposite case is treated similarly. The algorithm for trimming the wastful y from *ylist* is as follows.

Algorithm TRIMMING_YLIST

Step 1: Find $y^* \in N$ which is maximum among y satisfying $u(x,y) < u(x-1,y)$.

Step 2: If the cell y on the head of *ylist* satisfies $u(x,y) < u(x,y^*)$, then remove it from *ylist* (from Lemma 4.1) and go to Step 2.

Step 3: Let y_i and y_{i+1} be adjacent cells of *ylist* satisfying $y_i \leq y^* < y_{i+1}$. If $f(x-1,y_i) > f(x-1,y_{i+1})$ then remove y_i from *ylist* (from Lemma 4.2) and go to Step 3.

In Step 2, if there exists a $y' \leq y^*$ satisfying $u(x,y') < u(x,y^*)$, then the head y of *ylist* also satisfies $u(x,y) < u(x,y^*)$, since $u(x,y)$ is monotonically nondecreasing with y. Therefore all y satisfying $u(x,y) < u(x,y^*)$ are removed from *ylist* during the iteration of Step 2.

Initially, the *ylist* is composed of only one element. Suppose $f(x,y_i) \leq f(x,y_{i+1})$ holds for every i. For every $y_i > y^*$, $f(x-1,y_i) - f(x,y_i) = -1$ holds and for every $y_i \leq y^*$ which is not removed in Step 2 of TRIMMING_YLIST, $f(x-1,y_i) - f(x,y_i) = c$ (c is a constant satisfying $c \geq 0$) holds (i.e., c is the same for all $y_i \leq y^*$). Therefore, by induction, $f(x-1,y_i) \leq f(x-1,y_{i+1})$ hold for all y_i ($\geq x$) which remain in the *ylist* at the end of the algorithm TRIMMING_YLIST.

We have to spend $O(y-x)$ time to calculate $u(x,y)$ without any data structure. To achieve linear time, we have to obtain them in sufficiently short time. In our algorithm, we represent the functions u and l by lists called *ulist* and *llist*. For a fixed x, u (resp., l) is a monotonically nondecreasing

Figure 1 (resp., nonincreasing) function of y. (See Figure 1.) We will describe the construction of the linked list only for u, since the construction of *llist* is similar. The interval $[x+1,n]$ is decomposed into intervals $[y'_0 = x+1, y'_1 - 1], [y'_1, y'_2 - 1], \ldots, [y'_{r-1}, y'_r = n]$ where $u(x,y') = u(x,y'')$ holds if and only if both y' and y'' are included in $[y'_i, y'_{i+1} - 1]$. From this decomposition, we represent u by *ulist* composed of cells which correspond to these intervals. Each cell keeps the corresponding interval and the value $u(x,y)$ for y which the interval includes. A pair of cells are doubly linked by pointers

if they correspond to adjacent intervals. We say y is included in the cell of *ulist* if the interval that corresponds to it includes y.

To get the value of $u(x, y)$, we have to find the cell which includes y. To realize the operation in short time, we make a pointer from each cell y_i of *ylist* to the cell of *ulist* which includes y_i. We also make a pointer from each cell of *ulist* to the cell y_i of *ylist*, where y_i is the maximum among

Figure 2 those included in the same cell of *ulist*. (See Figure 2.)

The update of *ulist* and *llist* when x changes to $x-1$ is executed as follows. We update *llist* by adding a cell corresponding to interval $[x-1, x-1]$ on the head of it. (Recall that we treat only the case $\pi_{AB}(x-1) > \pi_{AB}(x)$.) We delete all the cells of *ulist* which include y satisfying $u(x, y) < u(x, y^*)$. For the cell including y^*, we change its interval to $[x-1, y^*]$ and its value from

Figure 3 $u(x, y^*)$ to $u(x-1, y^*)$. (See Figure 3.) Note that we do not remove the cell representing $u(x, y^*)$, but use it to represent $u(x-1, y^*)$. By doing this, pointers from y included in the cell corresponding to $u(x, y^*)$ to *ulist* need not to be changed. It is one of the key points to speeding up of the algorithm.

In Step 2 of TRIMMING_YLIST, if the pointer of the cell y of *ylist* indicates a deleted cell of *ulist*, we remove it from *ylist*, since this implies $u(x, y) < u(x, y^*)$. Thus it is not necessary to update pointers between *ylist* and *ulist*.

Let us consider the time complexity of the algorithm RC. Since those update operations of *ulist* are done by tracing *ulist* from its head to the cell including y^*, Step 1 and 2 of the algorithm TRIMMING_YLIST take $O(d+1)$ time, where d is the number of deleted cells in Step 2. The total number of deleted cells during the execution of the algorithm RC can not exceed the number of created cells, which is $O(n)$, and thus the total time of those operations in the algorithm RC is $O(n)$.

In Step 3 of the algorithm TRIMMING_YLIST, we can find y_i and y_{i+1} in $O(1)$ time by tracing a pointer from the cell of *ulist* including y^* to the cell of *ylist*. (See Figure 3.) Step 3 is repeated while the current cell is deleted. This is done in time proportional to the number of deleted cells. Thus the total time spent for Step 3 of the algorithm TRIMMING_YLIST in all iteration of the algorithm RC is proportional to the total number of deleted cells. It can not exceed the number of created cells, and the total time is $O(n)$.

In Line 3 of the algorithm RC, the cells $y_1, \cdots, y_r$ of *ylist* satisfy $f(x, y_i) \leq f(x, y_{i+1})$ $(i = 1, \ldots, r-1)$. Therefore we can enumerate all y satisfying $f(x, y) = 0$ by tracing *ylist* from its head without scanning y with $f(x, y) > 0$ in the middle. When we encounter y with $f(x, y) > 0$, we stop the tracing since $f(x, y') > 0$ holds for all $y' > y$. It takes time proportional to the number of outputs, which is $O(n + K)$.

As a result, the following theorem holds.

Theorem 4.1 *Algorithm RC with* TRIMMING_YLIST *outputs all common intervals in* $O(n + K)$ *time.*

4.1 Enumerating Common Intervals within a Specified Length

For two specified lengths $b_l \leq b_u$, we consider the problem enumerating the common intervals of two permutations whose length are not smaller than b_l and not greater than b_u. We enumerate them by using the above algorithm with slight modifications.

In each iteration, we keep the minimum cell $\hat{y}$ of *ylist* satisfying $\hat{y} - x + 1 \geq b_l$. At the end of Line 4 of the algorithm RC, we find the minimum cell $\hat{y}'$ satisfying $\hat{y}' - x + 2 \geq b_l$ and set $\hat{y} := \hat{y}'$. Since $\hat{y}'$ is either adjacent to $\hat{y}$ in the *ylist* or $\hat{y}' = \hat{y}$, the update can be done in $O(1)$ time. The enumeration of y satisfying $f(x, y) = 0$ and $b_l \leq y - x + 1 \leq b_u$ can be done in $O(n + K')$ time by tracing *ylist* from $\hat{y}$, where K' is the number of outputs for this problem.

4.2 Finding the Common Interval of Maximum Length within a Specified Length

In this subsection, we consider the problem finding a common interval of the maximum length whose length is less than or equal to a specified length b_u $(< n)$.

The basic idea is similar to the above algorithm. We keep the maximum cell $\bar{y}$ satisfying $\bar{y} - x + 1 > b^*$ and *ylist* is scanned from $\bar{y}$ in Line 3 of the algorithm RC, where $b^* \leq b_u$ is the maximum length of common intervals among those the algorithm found so far.

At the end of each iteration of the algorithm RC, we update b^* if the maximum length of common intervals whose length is less than b_u is larger than b^*. In such a case, we update $\bar{y}$ to it by tracing *ylist* from b^* while the cell satisfies $f(x, y) = 0$. Otherwise we find the minimum $\bar{y}'$ satisfying

$\bar{y}' - x > b^*$ and set $\bar{y} := \bar{y}'$, which is done in $O(1)$ time. Since the number of forward scans of *ylist* can not exceed b_u ($< n$) and the number of backward scans can not exceed n, the algorithm is executed in $O(n)$ time.

5 Random Inputs

In this section we will show two properties of two permutations generated uniformly at random (i.e., every permutation appears with probability $1/n!$): i) expected number of common intervals is $O(1)$, and ii) expected running time of the algorithm LHP is $O(n)$. For convenience, only the common intervals of length 2 to $n-2$ are considered in this section. This assumption does not change the above results as discussed in Subsection 3.1.

5.1 Expected Number of Common Intervals

We define random variables as follows. A variable X_{kx} ($x = 1, \ldots, n-k+1$, $k = 2, \ldots, n-2$) takes value 1 if $f(x, x+k-1) = 0$, and 0 otherwise. We also define $X_k = \sum_{x=1}^{n-k+1} X_{kx}$ and $X = \sum_{k=2}^{n-2} X_k$. These variables represent the number of common intervals of the length k and the number of common intervals of the length from 2 to $n-2$, respectively.

Theorem 5.2 *For $n \geq 5$, $E(X) = 2 + O(n^{-1})$. To be precise, $E(X_2) = 2 - \frac{2}{n}$, $E(\sum_{k=3}^{n-2} X_k) = O(n^{-1})$.*

Proof. For fixed x_A and x_B,

$$Pr(\{\sigma_A(i) \mid i \in [x_A, x_A + k - 1]\} = \{\sigma_B(i) \mid i \in [x_B, x_B + k - 1]\}) = \frac{(n-k)!k!}{n!}. \tag{16}$$

Since possible values of x_B is from 1 to $n-k+1$,

$$E(X_{kx_A}) = \frac{(n-k)!k!}{n!} \times (n-k+1). \tag{17}$$

By the linearity of expectation, we have

$$E(X_k) = \sum_{x=1}^{n-k+1} E(X_{kx}), \tag{18}$$

$$E(X) = \sum_{k=2}^{n-2} E(X_k). \tag{19}$$

To observe the increase and decrease of $E(X_k)$, we consider the solution of $E(X_k)/E(X_{k-1}) < 1$. From

$$\frac{E(X_k)}{E(X_{k-1})} = \frac{k(n-k+1)}{(n-k+2)^2} < 1, \tag{20}$$

$$2k^2 - (3n+5)k + (n^2+4n+4) > 0 \tag{21}$$

is obtained, and we get the solution $k < \alpha_-(n), \alpha_+(n) < k$, where $n \geq 4$ and

$$\alpha_-(n) = \frac{3n+5-\sqrt{n^2-2n-7}}{4}, \tag{22}$$

$$\alpha_+(n) = \frac{3n+5+\sqrt{n^2-2n-7}}{4}. \tag{23}$$

It is easy to check $0 < \alpha_-(n) \leq n$. By the fact

$$(n-3)^2 \leq n^2 - 2n - 7 \tag{24}$$

for $n \geq 4$, we have $\alpha_+(n) > n$. Therefore, $E(X_k)$ is monotonically nonincreasing with k when $2 \leq k \leq \alpha_-(n)$, and is monotonically nondecreasing with k when $\alpha_-(n) \leq k \leq n$. By using $E(X_4) < \frac{24}{n^2}$ and $E(X_{n-2}) \leq \frac{24}{n^2}$ $(n \geq 4)$, we have

$$E(X_k) \leq \frac{24}{n^2}, \quad (k = 4, 5, \ldots, n-2, n \geq 4), \tag{25}$$

and from (18),

$$E(X_2) = 2 - \frac{2}{n}, \tag{26}$$

$$E(X_3) = \frac{6(n-2)}{n(n-1)}. \tag{27}$$

Hence, we can conclude for $n \geq 5$,

$$E(\sum_{k=3}^{n-2} X_k) \leq E(X_3) + (n-5) \cdot \frac{24}{n^2} \tag{28}$$

$$= O(n^{-1}), \tag{29}$$

$$E(X) = E(X_2) + E(\sum_{k=3}^{n-2} X_k) \tag{30}$$

$$= 2 + O(n^{-1}). \ \square \tag{31}$$

By estimating the variance of X_2 and using Chebyshev bound and Markov inequality, the following theorem is also shown [5].

Theorem 5.3 *If $n \geq 5$, $Pr(X \geq \sqrt{2}t + 3) \leq \frac{1}{t^2} + O(n^{-1})$ holds for arbitrary $t > 0$.*

5.2 Expected Running Time of the Algorithm LHP

For each x ($x = 1, \ldots, n-1$), let T_x be a random variable representing the number of iterations for an inner loop of LHP. We also define $T = \sum_{x=1}^{n-1} T_x$, which represents the number of the whole inner loop iterations.

Theorem 5.4 *For $n \geq 4$, $E(T) \leq 3n$.*

Theorem 5.4 holds even if we do not incorporate the length condition (5) into LHP.

Before proving this theorem, we consider the following problem. We have k white balls and $m-k$ black balls ($0 \leq k \leq m-1$, $m \geq 1$) in an urn. The probability of taking out a ball is the same for all balls. Take out one ball. If it is white, we do not replace the ball into the urn and continue the same trial, otherwise (i.e., once a black ball is taken) we terminate the trial. Let $E_{\mathrm{urn}}(m,k)$ denotes the expected number of trials until a black ball is taken, then

$$E_{\mathrm{urn}}(m,k) = \frac{m+1}{m-k+1} \tag{32}$$

holds (see Appendix B). We define $E_{\mathrm{urn}}(m,m) = m$ for convenience. Let $E^*_{\mathrm{urn}}(m,k,j)$ denotes the expected number of trials until a black ball is taken or the number of trials becomes j, then

$$E^*_{\mathrm{urn}}(m,k,j) \leq E_{\mathrm{urn}}(m,k) \tag{33}$$

holds for $1 \leq j$, $0 \leq k \leq m$ and $m \geq 1$ (see also Appendix B). These facts are used in the proof.

Proof. By linearity of expectation,

$$E(T) = \sum_{x=1}^{n-1} E(T_x). \tag{34}$$

For a fixed x, let $r(x)$ be $\min\{n-x, n-3\}$, which is the maximum number of inner loop iterations for x. Since the two permutations are generated uniformly at random, HP' is $\{i,j\}$ with probability $\binom{n}{2}^{-1}$ for any i and j ($i,j \in [1,n]$, $i < j$). For such i and j, probability that $1 \leq \pi_{AB}(x) \leq i-1$ holds is $\frac{i-1}{n-2}$, and in this case, the expected number of inner loop iterations is $E^*_{\mathrm{urn}}(n-3, i-2, r(x))$, secondly, the probability that $i+1 \leq \pi_{AB}(x) \leq j-1$ holds is $\frac{j-i-1}{n-2}$, and in this case, the expectation is $E^*_{\mathrm{urn}}(n-3, j-i-2, r(x))$, and thirdly, the probability that $j+1 \leq \pi_{AB}(x) \leq n$ holds is $\frac{n-j}{n-2}$,

and in this case, the expectation is $E^*_{\rm urn}(n-3, n-j-1, r(x))$. Therefore,

$$\begin{aligned}
E(T) &= \sum_{x=1}^{n-1} \binom{n}{2}^{-1} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left\{ \frac{i-1}{n-2} E^*_{\rm urn}(n-3, i-2, r(x)) \right. \\
&\quad \left. + \frac{j-i-1}{n-2} E^*_{\rm urn}(n-3, j-i-2, r(x)) + \frac{n-j}{n-2} E^*_{\rm urn}(n-3, n-j-1, r(x)) \right\} \\
&\leq \frac{n-1}{\binom{n}{2}} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left\{ \frac{i-1}{n-2} E_{\rm urn}(n-3, i-2) \right. \\
&\quad \left. + \frac{j-i-1}{n-2} E_{\rm urn}(n-3, j-i-2) + \frac{n-j}{n-2} E_{\rm urn}(n-3, n-j-1) \right\} \\
&= \frac{n-1}{\binom{n}{2}} \cdot 3 \cdot \sum_{i=1}^{n-1} (n-i) \cdot \frac{i-1}{n-2} E_{\rm urn}(n-3, i-2) \\
&= \frac{6}{n} \left\{ \sum_{i=1}^{n-2} (n-i) \cdot \frac{i-1}{n-2} \cdot \frac{n-2}{n-i} + (n-3) \right\} \\
&= 3n - 9 \leq 3n. \ \square
\end{aligned}$$

6 Computational Results

In this section, we compare the algorithms BSC, LHP, MNG and RC by applying them to six types of problem instances of size up to 10^6.

6.1 Generation of Problem Instances

Following six types of problem instances are examined.

RAND: Two permutations σ_A and σ_B are randomly generated (any permutation is chosen with probability $1/n!$).

SWAP: Initially two permutations σ_A and σ_B are set as $\sigma_A(i) = \sigma_B(i) = i$ $(i = 1, \ldots, n)$. Then we repeat s times a swap of two elements $\sigma_B(i)$ and $\sigma_B(j)$ for two integers i and j $(i \neq j)$ randomly chosen from $[1, n]$. We set $s = n$ in the experiment.

NBRAND: The permutation σ_A is set as $\sigma_A(i) = i$ $(i = 1, \ldots, n)$. For an integer k, let p and q be integers satisfying $n = kp + q$ and $0 \leq q < k$. For each i $(i = 0, 1, \ldots, k)$, a permutation $\sigma_i : N_i \to N_i$ is randomly generated, where $N_i = \{ip + 1, ip + 2, \ldots, \min\{(i+1)p, n\}\}$, and σ_B is set as $\sigma_B = \sigma_0 \sigma_1 \cdots \sigma_k$. We set $k = \lfloor \sqrt{n} + 0.5 \rfloor$ in the experiment.

NBSWAP: Initially two permutations σ_A and σ_B are set as $\sigma_A(i) = \sigma_B(i) = i$ $(i = 1, \ldots, n)$. Then a swap of two elements $\sigma_B(i+j)$ and $\sigma_B(j)$ for an integer i randomly chosen from $[1, k]$ and an integer j randomly chosen from $[1, n-i]$ is repeated s times, where k is a parameter to restrict the swap distance. We set $k = \lfloor \sqrt{n} + 0.5 \rfloor$ and $s = n$ in the experiment.

SLIDE: For an integer k, let p and q be integers satisfying $n = kp + q$ and $0 \leq q < k$. Two permutations are set as $\sigma_A(i) = i$ $(i = 1, \ldots, n)$ and

$$\sigma_B(i) = \begin{cases} i - 2k - 1 \pmod{kp} + 1, & i \equiv 0 \pmod{k} \\ i, & \text{otherwise.} \end{cases}$$

Figure 4 An example with $n = 20$ and $k = 3$ is exhibited in Figure 4. We set $k = 4$ in the experiment.

NET: Two permutations are set as $\sigma_A(i) = i$ $(i = 1, 2, \ldots, n)$ and

$$\sigma_B(i) = \begin{cases} (i+1)/2, & i\text{: odd} \\ \lceil n/2 \rceil + i/2, & i\text{: even.} \end{cases}$$

Figure 5 An example with $n = 10$ is shown in Figure 5.

For type RAND instances, the expected number of common intervals is $2 + O(n^{-1})$ as shown in Section 5. By the similar discussion, we can show that the expected number of common intervals for type NBRAND instances is at most $k^2/2 + o(k^2)$ if $k = o(n)$. Recall that we choose $k = O(\sqrt{n})$ in the experiment, and hence, the expected number of outputs is $O(n)$.

For type SWAP and NBSWAP instances, it is observed that the number of common intervals is

Table 1 $O(n)$ as shown in Table 1, where each entry is the average of five instances examined in the next subsection.

For type SLIDE instances, the number of common intervals is at most

$$p\binom{k-1}{2} + \binom{q}{2} + k(q+1) \leq \frac{1}{2}kn + \frac{3}{2}k^2.$$

Recall that we choose $k = 4$, hence, the number of outputs is $O(n)$. For type NET instances, the number of common intervals is at most one.

6.2 Computational Results

All the tested algorithms were coded in C language and run on a workstation Sun SPARC classic. A simple multiplicative congruential method was used to generate random sequences. For each

type of problem (except for type SLIDE and NET problems), we generate five instances for each $n = 10^3 \sim 10^6$, and exhibit the average computational time (etc.) of each tested algorithm. Since type SLIDE and NET problems include no randomness, we exhibit the average data of three runs for each tested algorithms.

Table 2

Table 2 shows the average number of inner loop iterations of BSC, LHP and MNG divided by n for $n = 10^4$. (This implies the average number of iterations for an inner loop.) The mark '$*$' is put if this value was not increase more than 5% when $n = 10^6$, and for others, we mark '$\triangle$' if the instances with $n = 10^6$ was solved in one minute. Table 3 shows the average of the total number of scans on *ulist*, *llist* and *ylist* of the algorithm RC divided by n for $n = 10^6$. Figures $6 \sim 11$ show the average computational time (in μ secs.) divided by n. (Note that the data are identical to the average computational time in seconds when $n = 10^6$.)

Table 3

Figures 6 to 11

From these, we can observe the following:

- In Table 2, the marks '$*$' and '$\triangle$' imply the effectiveness of the speed up techniques proposed in Section 3. Especially for those with '$*$' marks, it is no problem to conclude that the problem instances were solved in $O(n)$ time on the average. For each of those with '$\triangle$' marks, the value increases with about 13% (resp., 38%) for NBSWAP (resp., NET) when $n = 10^6$. For NBSWAP, this is because the variance of the data of MNG is rather large. The same tendency was observed for LHP. Indeed, the value decreases with about 23% for LHP, NBSWAP combination when $n = 10^6$. It is known that MNG needs $O(n \log n)$ time for type NET instances, which is the cause of the increase with about 38%.

- The performances of BSC and RC are hardly affected by the type of instances: BSC always requires $O(n^2)$ time, while RC always runs in $O(n)$ time (recall that $K = O(n)$ for all tested problem instances). Note that the values in Table 3 are almost the same for other tested sizes.

- The algorithm LHP is quite effective for type RAND and SWAP instances. It is also effective for type NBSWAP instances, though about three times longer computational time is required compared to type RAND and SWAP instances. On the contrary, $O(n^{3/2})$ time is spent for type NBRAND instances and $O(n^2)$ time is needed for type SLIDE and NET instances.

- The algorithm MNG is quite effective for almost all types of problems except for NBRAND, for which it requires $O(n^{3/2})$ time. It is noted, however, that the running time of MNG is about three times larger than LHP in the case of type RAND and SWAP instances, and MNG requires $O(n \log n)$ time for type NET instances. It is also noted that problem types SLIDE and NET are quite artificial, and these results do not necessarily imply that MNG is more robust than LHP.

7 Conclusion

For the common interval enumeration problem, we proposed the following three algorithms i) a simple $O(n^2)$ time algorithm (LHP), whose expected running time becomes $O(n)$ for two randomly generated permutations, ii) a practically fast $O(n^2)$ time algorithm (MNG) using reverse Monge property, and iii) an $O(n+K)$ time algorithm (RC). It was observed in the computational experiment that: 1) LHP is very fast for randomly generated problem instances, 2) MNG is rather slower than LHP for random instances; however, there are cases that MNG can run in $o(n^2)$ time while LHP needs $\Omega(n^2)$ time, and 3) the performance of RC is quite robust about the type of problem instances, though it is rather slower than MNG for many of the tested problem instances. It is noted that LHP and MNG are very simple and easy to program (LHP is much simpler than MNG), while RC is rather complicated. On the other hand, it is also noted that there are cases that both LHP and MNG require $\Omega(n^2)$ time as mentioned in the end of Section 3. From these, we recommend RC if one wants to solve large instances (e.g., $n \geq 10^5$) with much time for programming, and LHP if one wants to solve the instances which seem to include randomness or one prefers a simpler algorithm. MNG is recommended if LHP fails to solve efficiently some problem instances one wants to solve.

Acknowledgement

The authors are grateful to Dr. Takeshi Tokuyama, currently at IBM Japan Ltd., Professor Dao-Zhi Zeng, currently at Kagawa Univ., Professor Hiroshi Nagamochi and Professor Toshihide Ibaraki, currently at Kyoto Univ., who gave them valuable comments for improving this paper.

References

[1] R. M. Brady, Optimization Strategies Gleaned from Biological Evolution, *Nature*, **317** (1985), 804–806.

[2] S. Kobayashi, I. Ono and M. Yamamura, An Efficient Genetic Algorithm for Job Shop Scheduling Problems, in *Proc. 6th International Conference on Genetic Algorithms* (L. J. Eshelman, ed.), Morgan Kaufmann, San Francisco, 1995, pp. 506–511.

[3] H. Mühlenbein, M. Gorges-Schleuter and O. Krämer, Evolution Algorithms in Combinatorial Optimization, *Parallel Computing*, **7** (1988), 65–85.

[4] M. Yamamura, T. Ono and S. Kobayashi, Character-Preserving Genetic Algorithms for Traveling Salesman Problem (in Japanese), *Journal of Japanese Society for Artificial Intelligence*, **7** (1992), 117–127.

[5] M. Yagiura, H. Nagamochi and T. Ibaraki, Two Comments on the Subtour Exchange Crossover Operator (in Japanese), *Technical Report of IEICE* (COMP94-18), **94**, No. 88, 1994, pp. 1–10 (a short version is in: *Journal of Japanese Society for Artificial Intelligence*, **10** (1995), 464–467).

[6] M. Yagiura and T. Ibaraki, Fast Algorithms to Enumerate All Common Intervals of Two Permutations (in Japanese), *Technical Report of IEICE* (COMP94-83), **94**, No. 479, 1995, pp. 65–74.

[7] R. L. Graham, D. E. Knuth and O. Patashnik, *Concrete Mathematics — A Foundation for Computer Science*, Addison-Wesley, 1989.

Appendix A

Here we prove the *reverse Monge property* of $f(\cdot,\cdot)$, that is,

$$f(x',y) + f(x,y') \geq f(x,y) + f(x',y')$$

holds for all x', x, y, y' satisfying $x' < x \leq y < y'$. Subtracting right-hand side from left-hand side, we get

$$u(x',y)+u(x,y')-\{u(x,y)+u(x',y')\}+l(x,y)+l(x',y')-\{l(x',y)+l(x,y')\}.$$

It is sufficient to show that $u(\cdot,\cdot)$ and $l(\cdot,\cdot)$ satisfy

$$\begin{aligned} u(x',y)+u(x,y') &\geq u(x,y)+u(x',y') && \text{(reverse Monge property)} \\ l(x',y)+l(x,y') &\leq l(x,y)+l(x',y') && \text{(Monge property).} \end{aligned}$$

We prove this only for $u(\cdot,\cdot)$, since the latter case is symmetrically proven. Either $u(x',y')=u(x,y')$ or $u(x',y')=u(x',y)$ holds, since

$$\begin{aligned} \max_{z\in[x',x-1]}\pi_{AB}(z) < u(x,y') &\Rightarrow u(x',y')=u(x,y') \\ \max_{z\in[x',x-1]}\pi_{AB}(z) \geq u(x,y') &\Rightarrow u(x',y')=u(x',y). \end{aligned}$$

This fact, combined with $u(x,y')\geq u(x,y)$ and $u(x',y)\geq u(x,y)$, implies that $u(\cdot,\cdot)$ satisfies reverse Monge property, and hence, reverse Monge property of $f(\cdot,\cdot)$ is proven.

Appendix B

Here, we prove that

$$E_{\mathrm{urn}}(m,k)=\frac{m+1}{m-k+1} \tag{35}$$

for $0\leq k\leq m-1$ and $m\geq 1$, and

$$E^{*}_{\mathrm{urn}}(m,k,j)\leq E_{\mathrm{urn}}(m,k) \tag{36}$$

for $1\leq j$, $0\leq k\leq m$ and $m\geq 1$, where $E_{\mathrm{urn}}(\cdot,\cdot)$ and $E^{*}_{\mathrm{urn}}(\cdot,\cdot,\cdot)$ are defined in Section 5. Let us define a random variable Z representing the number of trials until a black ball is taken out. The probability that a black ball is taken out after i trials or more is equal to the probability that white balls are taken in the first $i-1$ trials, so

$$Pr(Z\geq i)=\frac{[k]_{i-1}}{[m]_{i-1}},\quad i=1,\ldots,k+1 \tag{37}$$

holds, where

$$[m]_i=\begin{cases} 1, & i=0, \\ m(m-1)\cdots(m-i+1), & i>0. \end{cases} \tag{38}$$

By using this fact, we can conclude

$$\begin{aligned} E_{\mathrm{urn}}(m,k) &= \sum_{i=1}^{k+1} iPr(Z=i) \\ &= \sum_{i=1}^{k+1} Pr(Z \geq i) \\ &= \sum_{i=0}^{k} \frac{[k]_i}{[m]_i} \\ &= \sum_{i=0}^{k} \frac{\binom{m-i}{k-i}}{\binom{m}{k}} \\ &= \frac{m+1}{m-k+1}. \end{aligned} \tag{39}$$

See for example [7] for the last sigma calculation. When $k \leq m-1$, if $1 \leq j \leq k$, then

$$\begin{aligned} E^*_{\mathrm{urn}}(m,k,j) &= \sum_{i=1}^{j-1} iPr(Z=i) + jPr(Z \geq j) \\ &= \sum_{i=1}^{j} Pr(Z \geq i) \\ &\leq E_{\mathrm{urn}}(m,k), \end{aligned} \tag{40}$$

and if $j \geq k+1$, then $E^*_{\mathrm{urn}}(m,k,j) = E_{\mathrm{urn}}(m,k)$. When $k=m$, $E^*_{\mathrm{urn}}(m,m,j) = j \leq m = E_{\mathrm{urn}}(m,k)$. (Recall that we defined $E_{\mathrm{urn}}(m,m) = m$ for convenience.)

List of Figures

- Figure 1: Functions $u(2,y)$, $u(3,y)$, $l(2,y)$ and $l(3,y)$ corresponding to permutations $\sigma_A = \langle 1,2,3,4,5,6,7\rangle$ and $\sigma_B = \langle 5,3,1,4,2,7,6\rangle$.
- Figure 2: Examples of *ulist* and *llist* corresponding to $u(3,y)$ and $l(3,y)$ of Figure 1.
- Figure 3: The process of updating *ulist*,*llist* and *ylist*. The cells represented by dotted lines are deleted when *ulist* is updated.
- Figure 4: An example of type SLIDE instance with $n = 20$ and $k = 3$.
- Figure 5: An example of type NET instance with $n = 10$.
- Figure 6: Computational time against n (type RAND).
- Figure 7: Computational time against n (type SWAP).
- Figure 8: Computational time against n (type NBRAND).
- Figure 9: Computational time against n (type NBSWAP).
- Figure 10: Computational time against n (type SLIDE).
- Figure 11: Computational time against n (type NET).

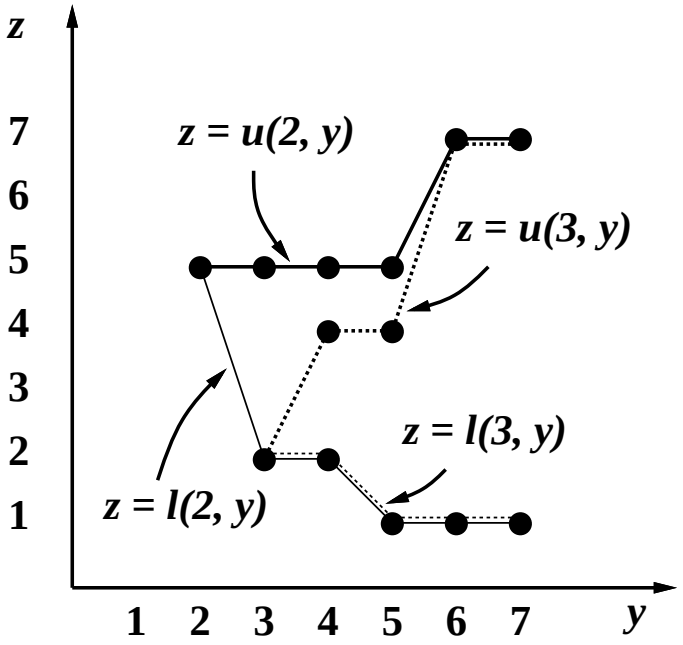

Figure 1: Functions $u(2,y)$, $u(3,y)$, $l(2,y)$ and $l(3,y)$ corresponding to permutations $\sigma_A = \langle 1,2,3,4,5,6,7 \rangle$ and $\sigma_B = \langle 5,3,1,4,2,7,6 \rangle$.

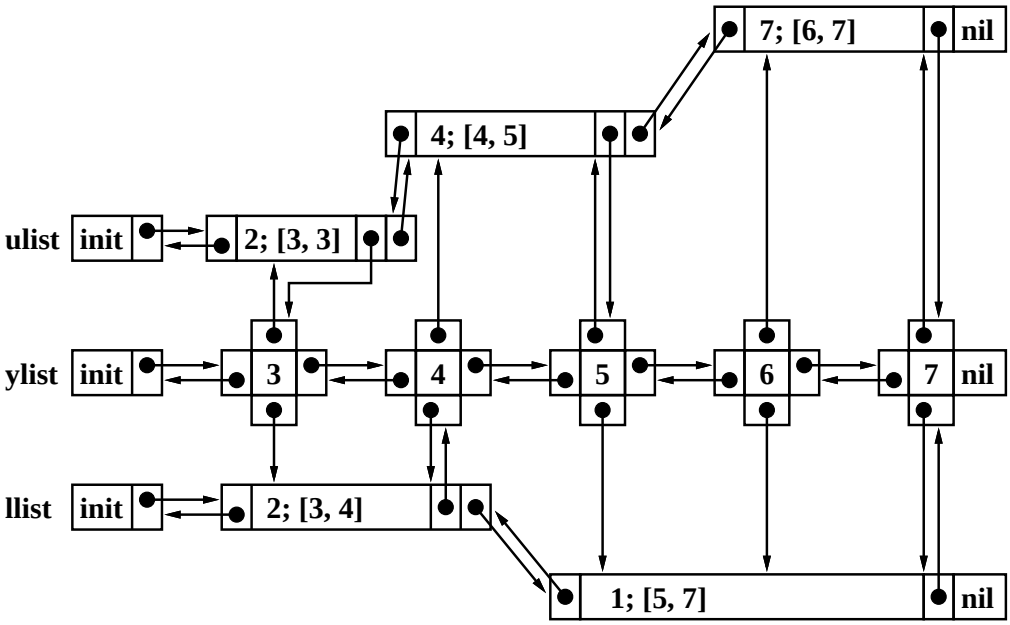

Figure 2: Examples of *ulist* and *llist* corresponding to $u(3, y)$ and $l(3, y)$ of Figure 1.

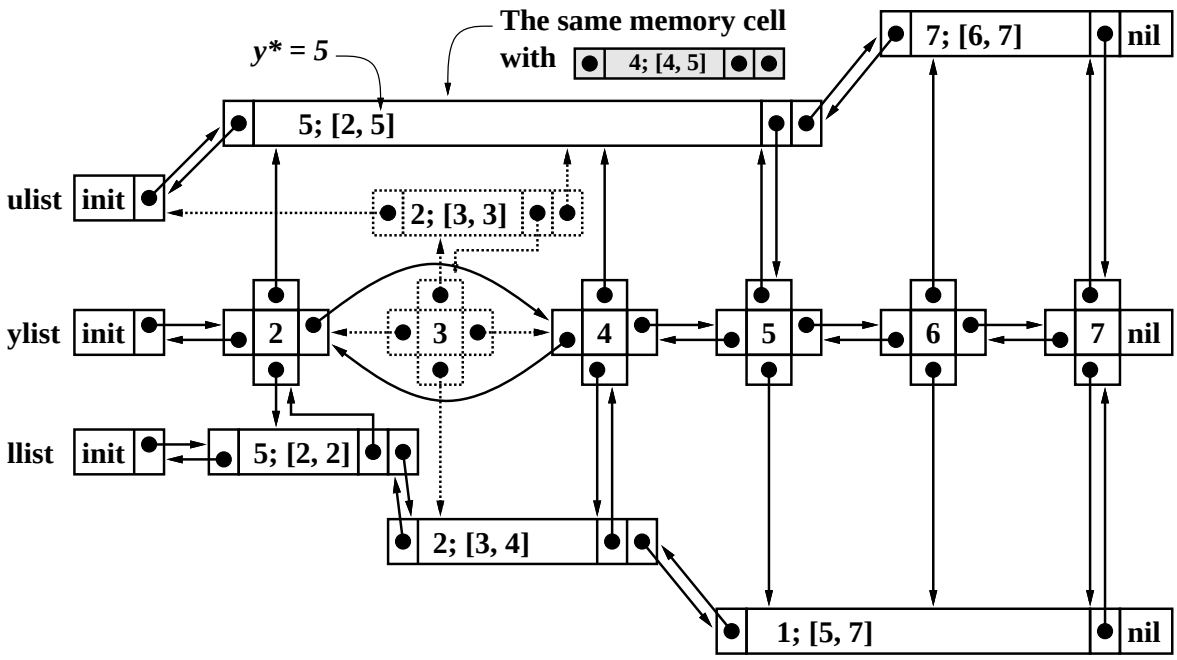

Figure 3: The process of updating *ulist*, *llist* and *ylist*. The cells represented by dotted lines are deleted when *ulist* is updated.

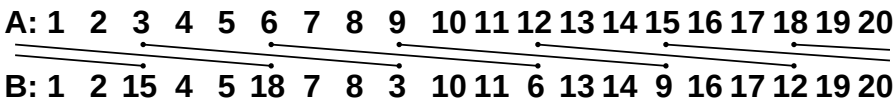

Figure 4: An example of type SLIDE instance with $n = 20$ and $k = 3$.

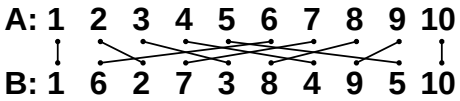

Figure 5: An example of type NET instance with $n = 10$.

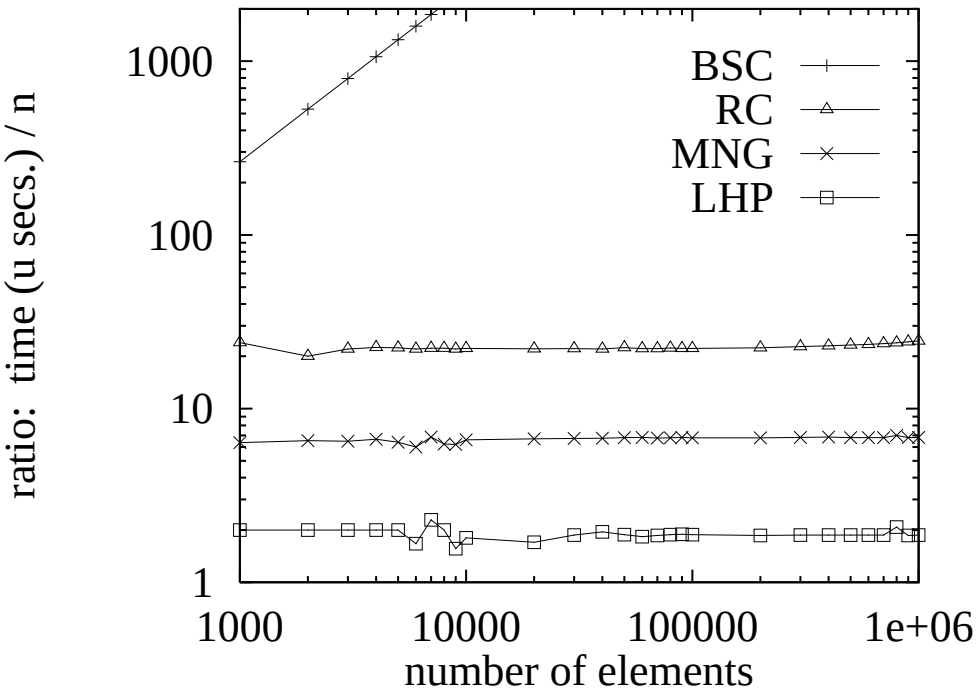

Figure 6: Computational time against n (type RAND).

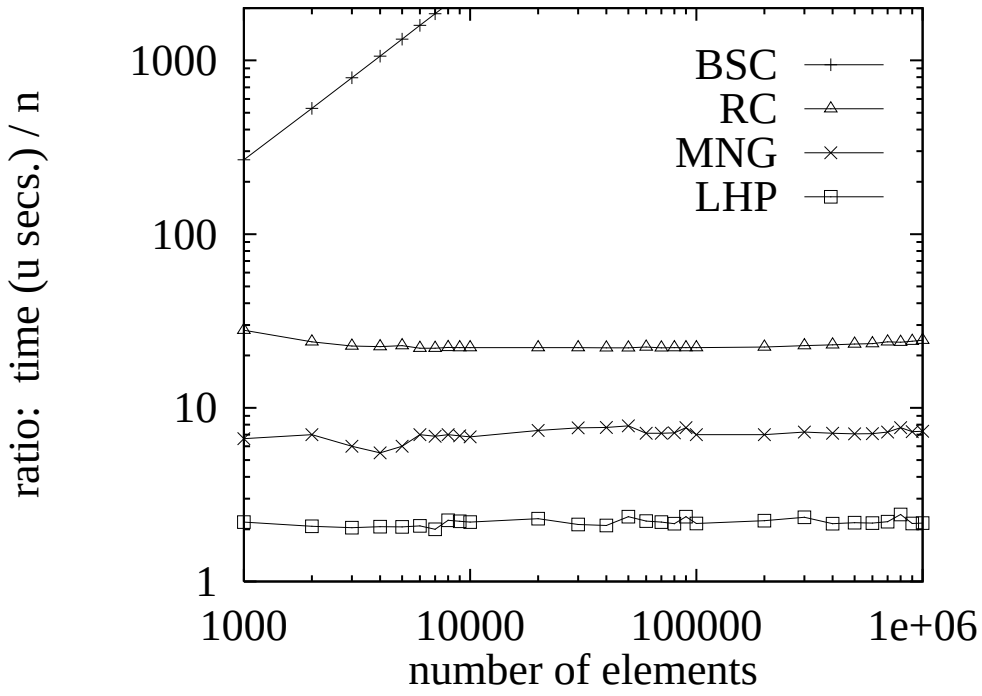

Figure 7: Computational time against n (type SWAP).

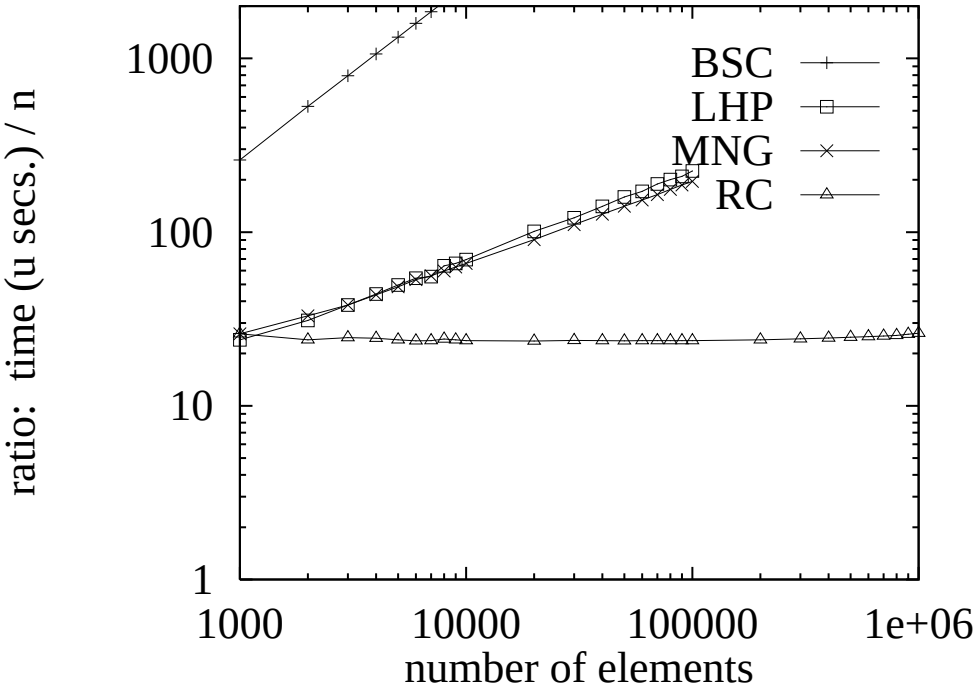

Figure 8: Computational time against n (type NBRAND).

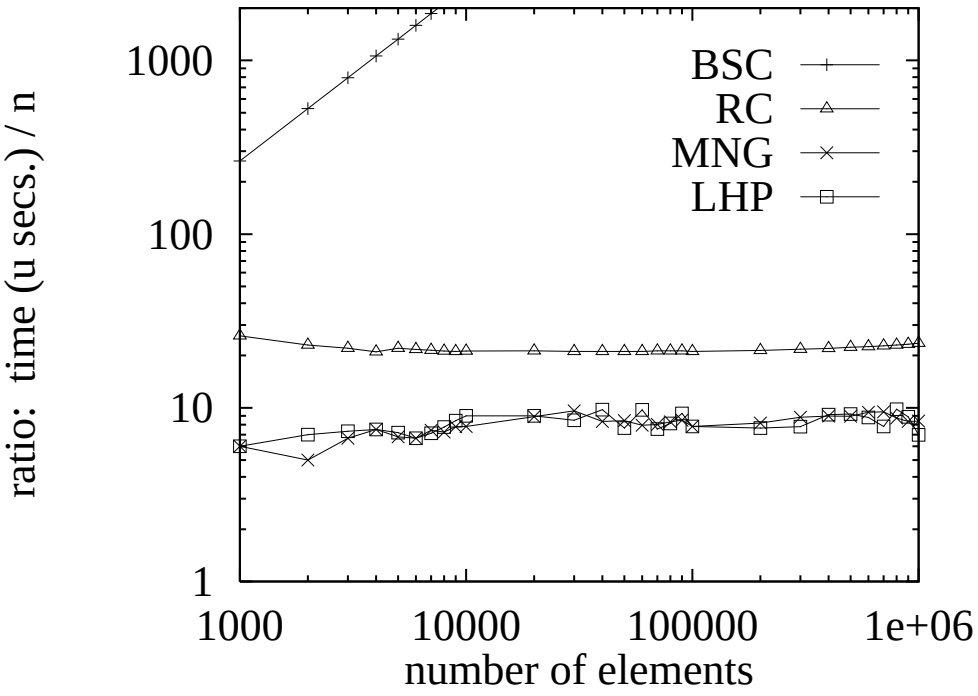

Figure 9: Computational time against n (type NBSWAP).

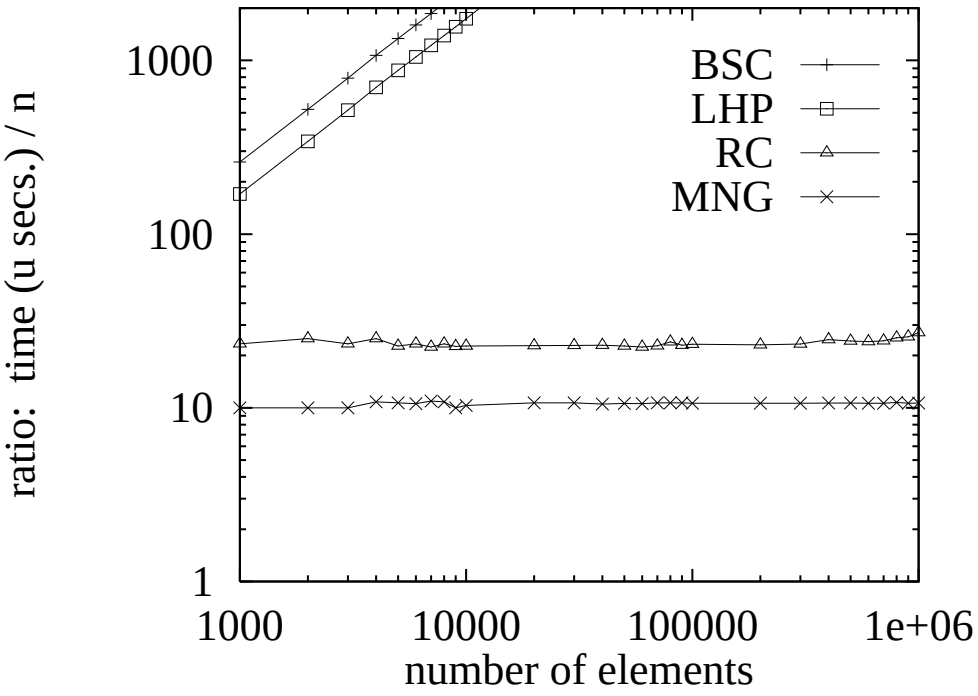

Figure 10: Computational time against n (type SLIDE).

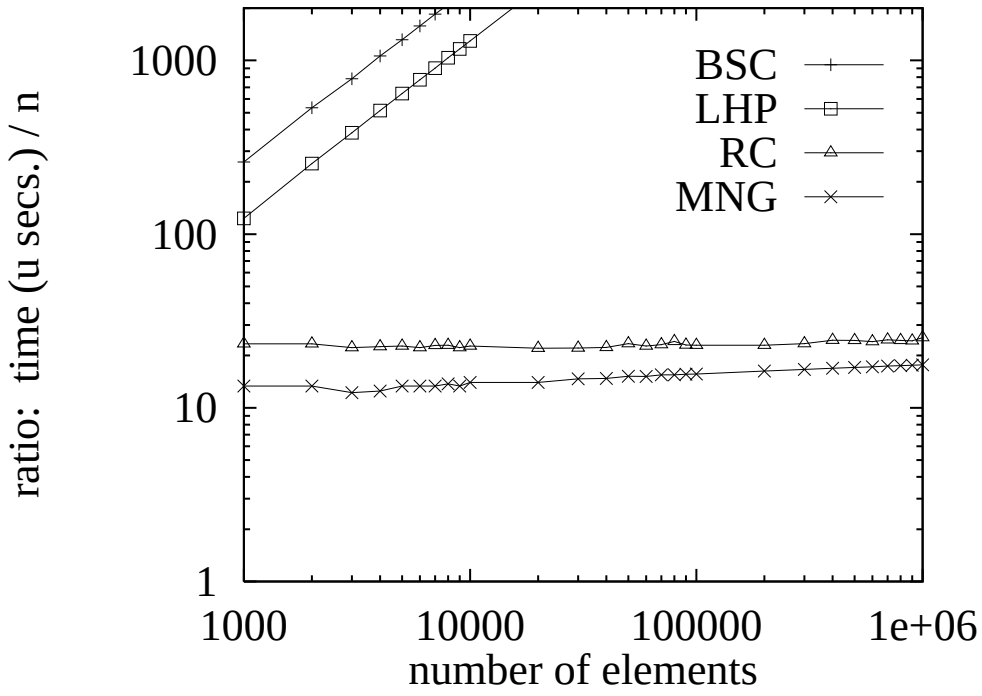

Figure 11: Computational time against n (type NET).

List of Tables

- Table 1: Average number of common intervals divided by n for type SWAP and NBSWAP instances.
- Table 2: Average number of inner loop iterations of BSC, LHP and MNG divided by n ($n = 10^4$).
- Table 3: Average of the total number of scans on *ulist*, *llist* and *ylist* of RC divided by n ($n = 10^6$).

Table 1: Average number of common intervals divided by n for type SWAP and NBSWAP instances.

	K/n	
n	SWAP	NBSWAP
1000	0.022	0.084
10000	0.021	0.050
100000	0.021	0.032
1000000	0.021	0.026

Table 2: Average number of inner loop iterations of BSC, LHP and MNG divided by n ($n = 10^4$).

	RAND	SWAP	NBRAND	NBSWAP	SLIDE	NET
BSC	4999.50	4999.50	4999.50	4999.50	4999.50	4999.50
LHP	*1.99	*2.33	99.62	*11.13	2498.50	1876.00
MNG	*3.40	*3.66	53.50	△4.39	*6.25	△8.68

Table 3: Average of the total number of scans on *ulist*, *llist* and *ylist* of RC divided by n ($n = 10^6$).

	RAND	SWAP	NBRAND	NBSWAP	SLIDE	NET
RC	27.45	27.44	28.94	27.60	29.75	28.00