Received 2 July 2022, accepted 28 July 2022, date of publication 3 August 2022, date of current version 8 August 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3196006

RESEARCH ARTICLE

Generating Almost Descending Sequences to Reduce the Number of Movements in Sorting

YUYIN YU[1] AND XUAN XIAO[2]
[1]School of Mathematics and Information Sciences, Guangzhou University, Guangzhou 510006, China
[2]School of Management, Guangzhou University, Guangzhou 510006, China

Corresponding author: Xuan Xiao (xiaoxuan@gzhu.edu.cn)

This work was supported in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2019A1515110992 and Grant 2021A1515011904, and in part by the National Natural Science Foundation of China under Grant 61502113.

ABSTRACT Sorting algorithms usually generate strictly descending sequences. However, almost descending sequences may be sufficient to fulfill our needs in practice, and the number of movements needed for generating such sequences can be greatly reduced. We call a sequence S almost descending if $S[j]-S[i] < C$ for any $i < j$ and a positive number C, which depends on the practical circumstances. Unless the difference between $S[j]$ and $S[i]$ exceeds the bound C, we deem them indistinguishable. For example, when people are arranged in a line from tall to short, two people with heights of 1.751 meters and 1.752 meters can be considered visually indistinguishable (we can set the bound to $C = 0.005$ meters in this case). Therefore, we do not need to adjust their positions, which leads to fewer movements in sorting. In this paper, three algorithms are provided to generate almost descending sequences, and their correctness is proven. The experimental results of the proposed algorithms demonstrate that our technique can reduce the number of movements in sorting. In addition, almost descending sequences are more stable than strictly descending sequences in dynamic sorting, which is useful in online sorting and sorting in noisy environments.

INDEX TERMS Almost descending sequence, sorting, algorithm, dynamic sorting.

I. INTRODUCTION

Sorting is a method of rearranging items in ascending or descending order; it is one of the fundamental operations in computer science and is widely used in database management, searching, network communications, etc. Famous sorting algorithms include bubble sort, selection sort, quick sort, insertion sort, heap sort, merge sort, shell sort, and radix sort [8], [9]. Bubble sort is typically included in textbooks when discussing sorting algorithms, and scholars have found interesting theoretical implications of it, such as bubble sort graphs [3], [15], bubble entropy [18], and bubble sort networks [21]. Nevertheless, it is seldom used to sort random sequences in practice due to its low efficiency. Selection sort has the same time complexity, $O(n^2)$, as bubble sort, while the number of movements is less than that of bubble sort.

Compared with bubble sort and selection sort, quick sort makes sorting more efficient, and thus scholars and practitioners have given more attention to it. For example, Kocamaz [14] implemented a faster variant of quick sort by using an artificial neural network-based algorithm selection approach. Aumuller *et al.* [4] studied the possible advantages multipivot quick sort might offer in general. Edelkamp *et al.* [10] mixed quick sort with another sorting algorithm to obtain quickXsort as a general template for transforming an external algorithm into an internal algorithm. As mentioned, each sorting algorithm has its own characteristics and advantages, and no one sorting algorithm is best for every situation. Therefore, scholars usually modify traditional algorithms or even design new algorithms to solve problems in different situations.

In this paper, we present a new case, inspired by sorting problems in real life, in which we usually do not distinguish or reorder objects unless they have large enough differences to be perceived by human eyes. We define such objects

The associate editor coordinating the review of this manuscript and approving it for publication was Mostafa M. Fouda.

as "indistinguishable" and aim to sort effectively when some elements in a sequence are indistinguishable. Some scholars have considered almost ordered sequences [12], roughly sorted sequences [1] and nearly sorted sequences [6], [7]. However, their research situations vary remarkably. For instance, Foster [12] presented an algorithm for sorting a collection of items that are almost ordered; this means that most of the records or sentences have been entered correctly, and the replacement of an old record or the insertion of a new one into already entered text occasionally occurs. Altman and Chlebus [1] studied how to sort a roughly sorted sequence, i.e., a sequence in which data that should be close to each other after sorting are already not very far from each other. Cook *et al.* [7] and Castro [6] proposed algorithms to sort a nearly sorted sequence, in which only a few elements are out of order.

In response to our specific research situation, we endeavor to generate almost descending sequences by modifying bubble sort, selection sort and quick sort. Unlike previous research, the almost descending sequence is not strictly decreasing due to indistinguishable elements, while the almost ordered sequence, the roughly sorted sequence and the nearly sorted sequence are completely ordered after sorting. Additionally, previous sorting solutions focus on reducing the number of data comparisons, but we aim at moving data as little as possible to minimize the number of movements and thus enhance the stability of the sequence. Our experimental results show that the almost descending versions of bubble, selection and quick sort perform better than their original versions when the differences between some elements are negligible.

This paper is organized as follows: In section 2, we give some notation used in this paper, followed by a detailed description of the new situation in section 3. Next, we introduce three algorithms to perform almost descending sort and prove their correctness. In section 5, we provide the experimental results of the three algorithms and the stability test of Algorithm 2. In section 6, we compare our algorithms with other "almost" or "nearly" sorting algorithms and illustrate how to use almost descending algorithms in online sorting and in noisy environments. Finally, we conclude by discussing the theoretical and practical contributions of our work.

II. NOTATION

We use the following notation throughout this paper.

$[i, j]$ denotes a sequence with $j-i+1$ continuous elements $i, i+1, i+2, \cdots, j-1, j$. If $i > j$, then $[i, j]$ is empty.

If S denotes a sequence with n elements, then $S[i, j]$ denotes a subarray of S consisting of $j-i+1$ elements $S[i]$, $S[i+1]$, $S[i+2]$, $\cdots$, $S[j]$. If $i > j$, then $S[i, j]$ is empty.

If S and R denote two sequences with n elements, then $S + R = [S[1] + R[1], S[2] + R[2], \cdots, S[n] + R[n]]$.

Repetition rate: Suppose T is a set with m different elements and S is a sequence with n elements. If each element of S is randomly chosen from T, then we say the repetition rate of S is $\frac{n}{m}$.

III. PROBLEM DESCRIPTION

A. BASIC IDEA

Generally, the efficiency of sorting algorithms is evaluated by important parameters, such as the number of comparisons, the number of movements, and the space requirements. In this paper, we focus on the situation in which each element in a sequence has a corresponding entity in real life, and the entity moves synchronously in real life when sorting. The problem is how to make as few moves as possible to generate an almost descending sequence. We use Problem 1 to explain our idea.

Problem 1: Five people with heights of 1.852 meters, 1.8 meters, 1.854 meters, 1.751 meters, and 1.752 meters are in a line. How can we move as few as possible to make them seem to be sorted in descending order?

Note the differences between "sorted in descending order" and "seem to be sorted in descending order". The former means sorting in a strictly descending sequence, while the latter implies an almost descending sequence. To give a strict definition of the almost descending sequence, we introduce the notion of "indistinguishable", which is similar to poly-time indistinguishable [13] but simplified.

Definition 1: We call two elements a and b ***indistinguishable under parameter*** *C if*

$$|a - b| < C \tag{1}$$

for a given positive number C.

Note that there always exists a positive number C, no matter how small, such that if the height difference between two people is less than C, then we cannot distinguish their differences by sight. Thus, we can ignore the height difference between these two people and treat them as indistinguishable when sorting. Based on this idea, almost descending can be defined as follows:

Definition 2: Suppose S is a sequence of n numbers; we call S ***almost descending with parameter*** *C if*

$$S[j] - S[i] < C \tag{2}$$

for any $1 \leq i < j \leq n$ and a positive number C.

Similarly, we can define almost ascending with parameter C if we change equation (2) to

$$S[i] - S[j] < C \tag{3}$$

for any $1 \leq i < j \leq n$ and a positive number C.

The reverse of an almost descending sequence is an almost ascending sequence, and vice versa. Thus, we only discuss the almost descending situation in this paper.

The positive number C, which sets a "bound" in the almost descending sequence, depends on the practical background. Regarding the height of two people, we can choose 0.005 meters as a bound. Suppose two people with a height difference of less than 0.005 meters are indistinguishable; then, Problem 1 can be solved as follows:

Solution of Problem 1: The heights of five people are to be listed as a sequence S.

1.852	1.8	1.854	1.751	1.752

The problem becomes how to make as few moves as possible to ensure that

$$S[j] - S[i] < 0.005 \tag{4}$$

for any $1 \leq i < j \leq 5$.

Obviously, we can obtain a new sequence by exchanging the positions of the second and third person:

1.852	1.854	1.8	1.751	1.752

The new sequence can be regarded as an almost descending sequence with parameter 0.005 meters. This sequence is not strictly descending, but it seems to be descending. Compared with generating a strictly descending sequence by at least three adjustments involving the five people, it is more efficient to exchange the positions of the second and third person to generate an almost descending sequence. Since we cannot visually distinguish people with tiny height differences in real life, the best sorting strategy is to permute their positions in a descending trend. Sporadic indistinguishable height differences that do not break the descending trend can be ignored to reduce the number of movements in sorting.

B. OUR GOAL

We transform the above requirement into the following mathematical problem, which our paper aims to solve:

Problem 2: Given a sequence S of n numbers, reorder this sequence to satisfy

$$S[j] - S[i] < C \tag{5}$$

for any $1 \leq i < j \leq n$ and a positive number C.

IV. ALGORITHMS AND THEIR CORRECTNESS

In this section, we modify bubble sort, selection sort and quick sort to provide three almost descending algorithms, namely, almost descending bubble sort, almost descending selection sort and almost descending quick sort, to solve Problem 2. In addition, we prove their correctness and illustrate their advantages compared to the corresponding strictly descending algorithms.

A. ALMOST DESCENDING BUBBLE SORT

In this subsection, we modify bubble sort to obtain almost descending bubble sort (Algorithm 1).

Proposition 1: Algorithm 1 will terminate and produce an almost descending sequence with parameter C.

Proof: **1)** Let $i = 1$; compare $S[i]$ with $S[i+1]$, $S[i+2], \cdots, S[n]$ one by one. If $S[j] - S[i] \geq C$ ($j = i+1, i+2, \cdots, n$), then exchange the values of $S[j]$ and $S[i]$. After these operations, we state that

$$S[j] - S[i] < C \text{ for all } j \in [i+1, n]. \tag{6}$$

Next, we prove the correctness of this assertion.

Algorithm 1 Almost-Descending-Bubble(S, C)

```
Input: a sequence $S$ of $n$ numbers and a positive number $C$.
Output: a sequence $S$ with $S[j] - S[i] < C$ for any $i < j$.
1: for $i = 1$ to $n - 1$ do
2:   for $j = i + 1$ to $n$ do
3:     if $S[j] - S[i] \geq C$ then
4:       Swap($S[i]$, $S[j]$);
5:     end if
6:   end for
7: end for
```

1.1) Let $t \in [i+1, n]$ be the minimum index such that

$$S[t] - S[i] \geq C; \tag{7}$$

then, we have

$$S[k] - S[i] < C \text{ for all } k \in [i+1, t-1]. \tag{8}$$

1.2) Let $\lambda = S[i]$; after exchanging the values of $S[i]$ and $S[t]$, we have $S[t] = \lambda$ and $S[i] \geq \lambda + C$. Together with equation (8), we can deduce that

$$S[k] - S[i] < C \text{ for all } k \in [i+1, t]. \tag{9}$$

1.3) Let $t' \in [t+1, n]$ be the minimum index such that $S[t'] - S[i] \geq C$, similar to **1.1)** and **1.2)**. After exchanging the values $S[i]$ and $S[t']$, we can obtain

$$S[k] - S[i] < C \text{ for all } k \in [i+1, t']. \tag{10}$$

By performing similar operations, we can obtain a sequence that satisfies equation (6). If we cannot find a t that satisfies equation (7), then the original sequence $S[i+1, n]$ must satisfy equation (6).

Note that $S[i] = \lambda$ in the original sequence; after exchanging the values of $S[i]$ and $S[t]$, we have $S[i] \geq \lambda + C$, that is, $S[i]$ becomes larger. Thus, equation (8) is still true after the exchange operation. This is the key point for understanding our algorithm.

2) Let $i = i + 1$; loop **(1)**. Algorithm 1 will terminate after running the last loop $i = n - 1$. □

B. ALMOST DESCENDING SELECTION SORT

In this subsection, we modify selection sort to obtain almost descending selection sort (Algorithm 2).

Proposition 2: Algorithm 2 will terminate and produce an almost descending sequence with parameter C.

Proof: This proof resembles the proof of Proposition 1. In Algorithm 1, if $S[j] - S[i] \geq C$ for some $i < j$, then exchange the values of $S[i]$ and $S[j]$. In Algorithm 2, if $S[j] - S[k] \geq C$ for some $k < j$, then set $k = j$. We can find an index k such that

$$S[j] - S[k] < C \text{ for all } j \in [i, n]. \tag{11}$$

Exchanging the values of $S[i]$ and $S[k]$, we have

$$S[j] - S[i] < C \text{ for all } j \in [i+1, n]. \tag{12}$$

Algorithm 2 Almost-Descending-Selection (S, C)

```
Input: a sequence $S$ of $n$ numbers and a positive number $C$.
Output: a sequence $S$ with $S[j] - S[i] < C$ for any $i < j$.
for $i = 1$ to $n - 1$ do
    j=i+1;
    k=i;
    while $j \le n$ do
        if $S[j] - S[k] \ge C$ then
            k=j;
        end if
        j=j+1;
    end while
    if $k \ne i$ then
        Swap($S[i]$, $S[j]$);
    end if
end for
```

The above explanation describes Lines 2-12; then, we set $i = i + 1$ and loop Lines 2-12 again. Algorithm 2 will terminate after running the last loop $i = n - 1$. □

C. ALMOST DESCENDING QUICK SORT

In this subsection, we modify quick sort to obtain almost descending quick sort (Algorithm 3).

Algorithm 3 Almost-Descending-Quick (S, C, L, R)

```
Input: a sequence $S$ of $R - L + 1$ numbers and a positive number $C$.
Output: a sequence $S$ with $S[j] - S[i] < C$ for all $L \le i < j \le R$.
1:  if $L < R$ then
2:      $p = S[R]$;
3:      $i = L$;
4:      $j = R$;
5:      while $i < j$ do
6:          while $i < j$ and $p - S[i] < \frac{C}{2}$ do
7:              $i = i + 1$;
8:          end while
9:          $S[j] = S[i]$;
10:         while $i < j$ and $S[j] - p < \frac{C}{2}$ do
11:             $j = j - 1$;
12:         end while
13:         $S[i] = S[j]$;
14:     end while
15:     $S[j] = p$;
16:     Almost-Descending-Quick($S, C, L, j - 1$);
17:     Almost-Descending-Quick($S, C, j + 1, R$);
18: end if
```

Proposition 3: Algorithm 3 will terminate and produce an almost descending sequence with parameter C.

Proof: **1)** Suppose $S[j] = p$. If

$$p - S[i1] < \frac{C}{2} \text{ for } i1 \in [L, j-1] \tag{13}$$

and

$$S[i2] - p < \frac{C}{2} \text{ for } i2 \in [j+1, R], \tag{14}$$

then there must be

$$S[i2] - S[i1] < C \text{ for } i1 \in [L, j-1] \text{ and } i2 \in [j+1, R]. \tag{15}$$

Based on equations (13), (14), and (15), if $S[L, j]$ and $S[j, R]$ are both almost descending sequences with parameter $\frac{C}{2}$, then $S[L, R]$ is an almost descending sequence with parameter C.

We assert that equations (13), (14), and (15) are true after Line 15. Lines 6-9 can ensure the correctness of equation (13), and Lines 10-13 can ensure the correctness of equation (14).

Equation (15) is true after Line 15, and the sequence has been divided into three parts, $S[L, j-1]$, p, and $S[j+1, R]$.

2) Loop **(1)** for $S[L, j-1]$; then, $S[L, j-1]$ will become almost descending after Line 16.

3) Loop **(1)** for $S[j+1, R]$; then, $S[j+1, R]$ will become almost descending after Line 17. □

V. PERFORMANCE ANALYSIS

In the previous section, we proposed three almost descending algorithms and proved their correctness. In this section, we investigate the performance of different algorithms followed by a stability test to solve the following two problems:

Problem 3: Given a random sequence, are fewer movements required to generate an almost descending sequence than to generate a strictly descending sequence?

A sequence is **dynamic** if the values of the sequence vary from time to time, and sorting such a sequence is called **dynamic sorting**. Web page ranking can be considered a dynamic sorting problem, since the ranks of pages change from time to time. We investigate dynamic sequences in two different conditions.

Condition 1: Suppose S is a descending sequence with n integer elements; let $S[i] = S[i] + r_i$ for any $1 \le i \le n$, where each r_i is randomly selected in [-M,M] ($M > 0$).

Condition 2: Suppose S is an almost descending sequence with parameter C and n integer elements, and let $S[i] = S[i] + r_i$ for any $1 \le i \le n$, where each r_i is randomly selected from [-M,M] ($M > 0$).

On this basis, we propose the following problem:

Problem 4: Are fewer movements required to maintain an almost descending sequence in condition 2 than to maintain a strictly descending sequence in condition 1?

For simplicity, we only consider integer sequences and integer parameters C in what follows. Other conditions are similar to the integer condition.

A. COMPARISON OF DATA MOVEMENTS

To solve Problem 3, let us recall some fundamental concepts about probability. $P(A)$: The probability of event A. $P(A|S)$: The conditional probability of A relative to S.

TABLE 1. Experimental results.

	n=100			n=1000			n=10000		
	C=2	C=20	C=200	C=2	C=20	C=200	C=2	C=20	C=200
Bubble	7197			550632			13338915		
Algorithm 1	6717	3117	438	364728	54096	5262	6516726	666381	58221
Advantage 1	6.67%	56.69%	93.91%	33.76%	90.18%	99.04%	51.15%	95.00%	99.56%
Selection	294			2973			29943		
Algorithm 2	288	267	225	2964	2928	2304	29907	29280	23676
Advantage 2	2.04%	9.18%	23.46%	0.30%	1.51%	22.50%	0.12%	2.21%	20.93%
Quick	476			6024			73460		
Algorithm 3	476	454	412	6024	5482	4510	73460	60426	47540
Advantage 3	0%	4.62%	13.45%	0%	9.00%	25.13%	0%	17.74%	35.28%

Proposition 4: Two integers k and t are randomly chosen from $[E, F]$ ($E < F$ are two integers). C is an integer such that $0 < C < F - E$. Let $D = F - E + 1$. Then, we have

$$P(t > k) = \frac{1}{2} - \frac{1}{2D} \tag{16}$$

and

$$P(t - k \geq C) = \frac{(D-C)(D-C+1)}{2D^2}. \tag{17}$$

Proof: 1) First, let us prove Equation (16).

Note that $P(k > t)$ denotes the probability that $k > t$. k and t are randomly chosen from $[E, F]$, so we have

$$\begin{cases} P(k < t) + P(k > t) + P(k = t) = 1, \\ P(k < t) = P(k > t), \\ P(k = t) = \dfrac{1}{F - E + 1}. \end{cases} \tag{18}$$

According to Equation (18), we can obtain

$$P(k > t) = \frac{1}{2} - \frac{1}{2(F - E + 1)}. \tag{19}$$

2) We prove Equation (17) in this section. Note that k and t are randomly chosen from $[E, F]$, so we have

$$\begin{cases} P(t - k \geq C | k \in (F - C, F]) = 0, \\ P(t - k \geq C | k \in [E, F - C]) = \dfrac{F - k - C + 1}{F - E + 1}. \end{cases} \tag{20}$$

According to Equation (20), we have

$$\begin{aligned} P(t - k \geq C) &= \sum_{k=E}^{F-C} \frac{F - k - C + 1}{F - E + 1} \cdot \frac{1}{F - E + 1} \\ &= \frac{(D-C)(D-C+1)}{2D^2}. \end{aligned} \tag{21}$$

□

Corollary 1: According to Proposition 4, we have
1) $P(t > k) > P(t - k \geq C)$,
2) $P(t - k \geq C)$ decreases as C increases.

Given a random sequence S, for any $i < j$, let $k = S[i]$ and $t = S[j]$; then, $P(k > t)$ denotes the probability that we need to exchange the values of $S[i]$ and $S[j]$ to make them descending, while $P(t - k \geq C)$ denotes the probability that we need to exchange the values of $S[i]$ and $S[j]$ to make them almost descending.

Therefore, 1) in Corollary 1 implies that for any random integers $S[i]$ and $S[j]$, fewer movements are needed to make them almost descending than to make them descending.

2) in Corollary 1 implies that for any random integers $S[i]$ and $S[j]$, fewer movements are needed to make them almost descending with parameter C as C becomes larger.

Proposition 4 and Corollary 1 partly solve Problem 3. In addition, we run some experiments to demonstrate our view (see Table 1). For each parameter n, we generate a random sequence S of n elements, and each $S[i] (1 \leq i \leq n)$ is randomly chosen between 1 and 1000. Then, we run these algorithms under different parameters. Table 1 illustrates the number of data movements for these algorithms.

Take $n = 100$ and $C = 200$ (Column 4) as an example. We randomly choose 100 elements from [1..1000] to obtain a sequence S. When using bubble sort on S, the number of data movements is 7197. When using Algorithm 1 (with parameter $C = 200$) to sort the same S, the number of data movements is 438. Accordingly, the advantage is $\frac{7197-438}{7197} = 93.91\%$, which means that Algorithm 1 (with parameter $C = 200$) can save 93.91% of the data movements of bubble sort.

As we can see in Table 1, the number of data movements depends greatly on two parameters: the repetition rate of S and the bound C. An increase in C leads to a decrease in the number of data movements of Algorithms 1, 2 and 3, which accords with 2) in Corollary 1. Given the same bound C, Algorithms 1, 2 and 3 work more effectively when S has a higher repetition rate. Regardless of which almost descending algorithm is used, an increase in C generally brings about fewer data movements and thus a greater advantage over the corresponding algorithm, especially for sequences with high repetition rates. In addition, Algorithm 1 improves bubble sort to a great extent, although the number of data movements remains the largest among the three almost descending algorithms. Moreover, Algorithm 2 results in the fewest data movements when generating an almost descending sequence.

In addition to Algorithms 1, 2 and 3, we can follow the same logic to modify other sorting algorithms, such as insertion sort, merge sort and heap sort, as almost descending versions to solve Problem 2. However, insertion sort requires too many movements when inserting a new element into its position. Merge sort requires linear extra space. Heap sort with n levels has to generate almost descending subsequences with parameter $\frac{C}{n}$. These algorithms may not perform as well as Algorithms 1, 2 or 3; thus, we list only three algorithms to explain our idea.

B. STABILITY TEST

In this section, we present the stability test of Algorithm 2 to solve Problem 4. We first provide a simple example to explain the problem.

Example 1: (i) Define

$$\begin{cases} S = [991, 990, 987, 745, 743, 742, 543, 534, 304], \\ R = [-5, 5, 4, 3, -4, 4, 2, -3, 1]. \end{cases}$$

Note that S is a strictly descending sequence, and R is a random sequence with each element selected from $[-5, 5]$, but

$$S + R = [986, 995, 991, 748, 739, 746, 545, 531, 305]$$

is no longer strictly descending. In this case, we have to change the positions of five elements to make the sequence $S + R$ strictly descending again.

(ii) Define

$$\begin{cases} S = [987, 990, 991, 742, 745, 743, 543, 534, 304], \\ R = [-5, 5, 4, 3, -4, 4, 2, -3, 1]. \end{cases}$$

Note that S is an almost descending sequence with parameter $C = 20$, and R is a random sequence with each element selected from $[-5, 5]$; therefore,

$$S + R = [982, 995, 995, 745, 741, 747, 545, 531, 305]$$

is still an almost descending sequence with parameter $C = 20$.

In Example 1, $S + R$ simulates the dynamic condition that the value of S varies slightly.

Anagnostopoulos *et al.* [2] investigated how to output an approximate solution under dynamic conditions. In their case, the approximate solution resembled Cook *et al.* [7] and Castro's [6] "nearly sorted sequence", in which only a few elements are out of order. However, we provide a different solution based on the idea of almost descending algorithms, in which many elements may be out of order, but the differences between these elements are rather small and negligible.

Example 1 illustrates that an almost descending sequence may preserve the almost descending property if the values of its elements vary slightly. Therefore, the cost of maintaining an almost descending sequence is lower than that of maintaining a strictly descending sequence in dynamic sorting. However, slight variations may sometimes change the almost descending property of a sequence. In this regard, is the cost still lower to maintain the almost descending property than to maintain the strictly descending property in the dynamic condition? Proposition 5, Figure 1 and Figure 2 demonstrate this problem. We need the following lemma to prove Proposition 5:

Lemma 1: Suppose that M is a positive integer and ϵ_1, ϵ_2 are randomly chosen from $[-M, M]$. Then, we have

$$P(\epsilon_1 - \epsilon_2 = i) = \frac{(2M + 1 - |i|)}{(2M + 1)^2} \ (i \in [-2M, 2M]). \quad (22)$$

Proof: Note that ϵ_1 and ϵ_2 are randomly chosen from $[-M, M]$. When $i \in [0, 2M]$, we have

$$\begin{cases} P(\epsilon_1 - \epsilon_2 = i | \epsilon_2 \in (M - i, M]) = 0 \\ P(\epsilon_1 - \epsilon_2 = i | \epsilon_2 \in [-M, M - i]) = \dfrac{1}{2M + 1}, \end{cases} \quad (23)$$

which implies that

$$\begin{aligned} P(\epsilon_1 - \epsilon_2 = i) &= \sum_{\epsilon_2=-M}^{M-i} \frac{1}{2M + 1} \cdot \frac{1}{2M + 1} \\ &= \frac{2M + 1 - i}{(2M + 1)^2}. \end{aligned} \quad (24)$$

Similarly, when $i \in [-2M, 0]$, we have

$$P(\epsilon_1 - \epsilon_2 = i) = \frac{(2M + 1 + i)}{(2M + 1)^2}. \quad (25)$$

Equation (24) and Equation (25) imply Equation 22. □

Proposition 5: Two integers k and t are randomly chosen from $[E, F]$ ($E < F$ are two integers). C is an integer such that $0 < C < F - E$. Let $D = F - E + 1$. M is a positive integer such that $M < \frac{C}{2}$, and ϵ_1, ϵ_2 are randomly chosen from $[-M, M]$. Then, we have

$$\begin{aligned} &P(k + \epsilon_1 < t + \epsilon_2 | k \geq t) \\ &\quad = \frac{M[-2M^2 + (4D - 1)M + 4D + 1]}{3D(D + 1)(2M + 1)} \end{aligned} \quad (26)$$

and (27), as shown at the bottom of the page.

Proof: 1) For any $i \in [0, 2M]$, we have

$$\begin{aligned} P(t - k \leq -i) &= \sum_{k=E+i}^{F} \frac{1}{D} \cdot \frac{k - i - E + 1}{D} \\ &= \frac{(D - i)(D - i + 1)}{2D^2}. \end{aligned} \quad (28)$$

$$P(k + C + \epsilon_1 \leq t + \epsilon_2 | k + C > t) = \frac{-3M^2(2M + 1) + 2(M + C - D)M(4M + 1) - 3(2M + 1)(2C - 2D - 1)M}{3[D^2 + (2C - 1)D - C(C - 1)](2M + 1)}. \quad (27)$$

According to Equation (16) and Equation (28), we have

$$\begin{aligned}
&P(k+\epsilon_1 < t+\epsilon_2 | k \geq t) \\
&= P(\epsilon_1 - \epsilon_2 < t-k | t-k \leq 0) \\
&= \frac{P(\epsilon_1-\epsilon_2 < t-k \leq 0)}{P(t-k\leq 0)} \\
&= \sum_{i=1}^{2M} \frac{P(-i < t-k \leq 0)}{P(t-k \leq 0)} \cdot P(\epsilon_1-\epsilon_2 = -i) \\
&= \sum_{i=1}^{2M} \frac{P(t-k\leq 0) - P(t-k \leq -i)}{P(t-k\leq 0)} \cdot P(\epsilon_1-\epsilon_2=-i) \\
&= \sum_{i=1}^{2M} \frac{\frac{1}{2D^2}\cdot[(2D+1)i-i^2]}{\frac{D+1}{2D}} \cdot \frac{2M+1-i}{(2M+1)^2} \\
&= \frac{\sum_{i=1}^{2M}[i^3-(2M+2D+2)i^2+(2D+1)(2M+1)i]}{(2M+1)^2D(D-1)} \\
&= \frac{M[-2M^2+(4D-1)M+4D+1]}{3D(D+1)(2M+1)}.
\end{aligned}$$

2) For any $i \in [0, 2M]$, we have

$$\begin{aligned}
&P(t-k-C < -i) \\
&= \sum_{k=E}^{F+i-C} \frac{1}{D}\cdot\frac{k+C-i-E}{D} + \sum_{k=F+i-C+1}^{F}\frac{1}{D}\cdot 1 \\
&= \frac{-i^2+(2C-2D-1)i+D^2-C^2+2DC+C-D}{2D^2}.
\end{aligned} \tag{29}$$

According to Equation (17) and Equation (29), we have $P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t)$, as shown at the bottom of the page. □

Corollary 2: According to Proposition 5, we have the following:

1) When M increases, $P(k+\epsilon_1 < t+\epsilon_2|k\geq t)$ and $P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t)$ increase.

2) When C becomes larger, $P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t)$ decreases.

3) $P(k+\epsilon_1 < t+\epsilon_2|k\geq t) > P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t)$.

Proof: 1) According to Equation (26), we have

$$P(k+\epsilon_1 < t+\epsilon_2|k \geq t) = \frac{-2M^2+(4D-1)M+4D+1}{3D(D+1)(2+\frac{1}{M})}. \tag{30}$$

Let $f(M) = -2M^2+(4D-1)M+4D+1$; then, the derivative $f'(M) = -4M+(4D-1) > 0$. Thus, $f(M)$ is an increasing function on M. Therefore, an increase in M leads to a decrease in $3D(D+1)(2+\frac{1}{M})$ and an increase in $-2M^2+(4D-1)M+4D+1$. Thus, as M increases, $P(k+\epsilon_1 < t+\epsilon_2|k\geq t)$ increases. Similarly, according to Equations (27), we find that as M increases, $P(k+C+\epsilon_1\leq t+\epsilon_2|k+C>t)$ increases.

2) Let $u(C) = -3M^2(2M+1)+2(M+C-D)M(4M+1)-3(2M+1)(2C-2D-1)M$ and $v(C) = 3[D^2+(2C-1)D-C(C-1)](2M+1)$. Then, we have

$$u'(C) = 2M(4M+1) - 6(2M+1)M = -4M^2-4M < 0 \tag{31}$$

and

$$v'(C) = 3[2D-2C+1](2M+1) > 0, \tag{32}$$

which implies that $u(C)$ is a monotonically decreasing function on C and $v(C)$ is a monotonically increasing function on C. Thus, $P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t)$ is a monotonically decreasing function on C.

3) Since we have proved that $P(k+C+\epsilon_1\leq t+\epsilon_2|k+C>t)$ is a monotonically decreasing function on C, we only need

$$\begin{aligned}
P(k+C+\epsilon_1 \leq t+\epsilon_2|k+C>t) &= P(\epsilon_1-\epsilon_2 \leq t-k-C|t-k-C<0) \\
&= \frac{P(\epsilon_1-\epsilon_2\leq t-k-C<0)}{P(t-k-C<0)} \\
&= \sum_{i=1}^{2M}\frac{P(-i\leq t-k-C<0)}{P(t-k-C<0)}\cdot P(\epsilon_1-\epsilon_2=-i) \\
&= \sum_{i=1}^{2M}\frac{P(t-k-C<0)-P(t-k-C<-i)}{P(t-k-C<0)}\cdot P(\epsilon_1-\epsilon_2=-i) \\
&= \sum_{i=1}^{2M}\frac{\frac{i^2-(2C-2D-1)i}{2D^2}}{\frac{D^2+(2C-1)D-C(C-1)}{2D^2}}\cdot\frac{2M+1-i}{(2M+1)^2} \\
&= \frac{\sum_{i=1}^{2M}[-i^3+(2M+2C-2D)i^2-(2M+1)(2C-2D-1)i]}{[D^2+(2C-1)D-C(C-1)](2M+1)^2} \\
&= \frac{-3M^2(2M+1)+2(M+C-D)M(4M+1)-3(2M+1)(2C-2D-1)M}{3[D^2+(2C-1)D-C(C-1)](2M+1)}
\end{aligned}$$

to prove that

$$P(k+\epsilon_1 < t+\epsilon_2 | k \geq t) > \frac{u(2M)}{v(2M)} = \frac{M[-6M^2+(4D-3)M+4D+3]}{[D^2+(4M-1)D-2M(2M-1)](2M+1)}. \quad (33)$$

According to Equation (26) and the analysis of $v'(C)$ in the proof of part 2), we can conclude that Equation (33) is true, since $\frac{u(2M)}{v(2M)}$ has a smaller numerator and larger denominator. □

Given a sequence S, for any $i < j$, let $k = S[i]$ and $t = S[j]$. With the same notation as in Proposition 5, $k > t$ denotes that $S[i]$ and $S[j]$ satisfy the descending property, and $k+\epsilon_1 < t+\epsilon_2$ denotes that $S[i]+\epsilon_1$ and $S[j]+\epsilon_2$ do not satisfy the descending property after a small variation (note that we use $+\epsilon_1$ and $+\epsilon_2$ to denote a small variation in dynamic sorting). We have to exchange the values of $S[i]+\epsilon_1$ and $S[j]+\epsilon_2$ to preserve the descending property, which will lead to data movements. Therefore, $P(k+\epsilon_1 < t+\epsilon_2 | k \geq t)$ can be interpreted as the probability that we need to exchange $S[i]+\epsilon_1$ and $S[j]+\epsilon_2$ to retain the descending property in dynamic sorting. Similarly, $P(k+C+\epsilon_1 \leq t+\epsilon_2 | k+C > t)$ can be interpreted as the probability that we need to exchange $S[i]+\epsilon_1$ and $S[j]+\epsilon_2$ to retain the almost descending property in dynamic sorting.

Therefore, we can obtain the following properties:

Property 1) in Corollary 2 implies that for any $S[i] > S[j]$ or $S[i]+C > S[j]$ $(i < j)$, more movements are needed to maintain the descending property or the almost descending property in dynamic sorting when M becomes larger.

Property 2) in Corollary 2 implies that for any $S[i]+C > S[j]$ $(i < j)$, fewer movements are needed to maintain the almost descending property in dynamic sorting when C becomes larger.

Property 3) in Corollary 2 implies that for any $S[i] > S[j]$ or $S[i]+C > S[j]$ $(i < j)$, fewer movements are needed to maintain the almost descending property than to maintain the descending property in dynamic sorting.

Proposition 5 and Corollary 2 partly solve Problem 4. In addition, we run some experiments to support our view (see Figure 1 and Figure 2). Figure 1 and Figure 2 show the numbers of movements when generating descending and almost descending sequences for different parameters. In Figure 1, $M = 5$, and the integer $C \in [10, 29]$. In Figure 2, $C = 40$, and the integer $M \in [1, 20]$. We run 300 random trials for each M and C (count = 300) and calculate the mean of the number of movements. In each experiment, we generate a random almost descending sequence S with $n(= 1000)$ elements and $S[i] \in [1, 1000]$ for $1 \leq i \leq 1000$, then generate a random sequence R with 1000 elements and $R[i] \in [-M, M]$ for $1 \leq i \leq 1000$. Suppose S' is the corresponding strictly descending sequence after reordering sequence S with Selection. Thus, the number of movements denotes the number of data movements when running Selection$(S'+R, C)$ and Almost-Descending-Selection$(S+R, C)$.

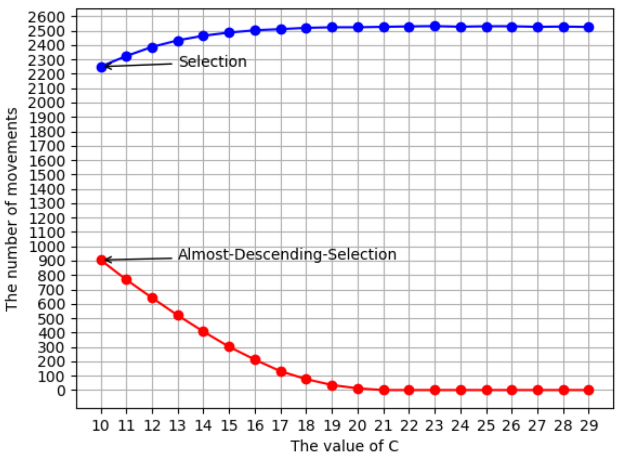

FIGURE 1. n = 1000, count = 300, M = 5.

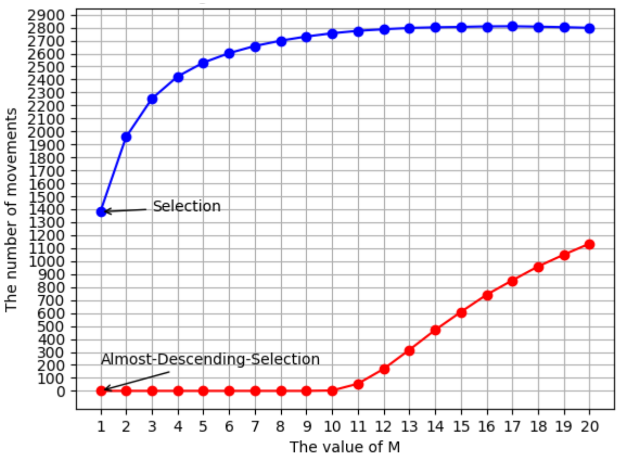

FIGURE 2. n = 1000, count = 300, C = 40.

Figure 1 and Figure 2 can demonstrate Properties 1), 2) and 3) in Corollary 2. We can see that the number of movements for Selection is far greater than that for Almost-Descending-Selection in dynamic sorting. The almost descending sequence remains almost descending when the values of each element vary slightly, especially when C is sufficiently large or M is sufficiently small.

In sum, almost descending sequences may preserve the almost descending property if the values of their elements vary slightly. Even if a slight variation changes the almost descending property, these sequences need fewer adjustments to become almost descending again than strictly descending sequences need. Therefore, we believe that almost descending sequences are more stable than strictly descending sequences in dynamic sorting.

VI. DISCUSSION AND APPLICATION

We investigated the properties of almost descending algorithms in the above sections. In this section, we will compare another algorithm for "almost" or "nearly" sorting.

Additionally, we will provide situations in which almost descending algorithms can be of help.

A. ALMOST ORDERED

Foster [12] introduced a way to sort almost ordered arrays in the case of a time-sharing environment in which most of the records or sentences have been entered correctly but there may occasionally be a replacement of an old record or an insertion of new records into already entered text. The almost ordered algorithm first constructs a table with one entry for each part of the source text that is internally ordered and then performs an N-way merge of the parts. Similarly, we can design an N-way merge almost descending algorithm. However, this algorithm has at least two weaknesses. First, the number of movements is larger than that of Algorithm 1. Second, we have to choose the bound $\frac{C}{2}$ when merging two sequences to obtain an almost descending sequence with parameter C (similar to that in Algorithm 3). Therefore, it is not a good choice to construct a merge sort version of the almost descending algorithm.

B. ROUGHLY SORTED

A sequence $\overline{a} = (a_1, a_2, \cdots, a_n)$ is k-sorted if and only if $j - i > k$ implies $a_i < a_j$ for all $1 \leq i, j \leq n$. A sequence is sorted if and only if it is 0-sorted. Define the radius of $\overline{a}$ as the smallest k such that $\overline{a}$ is k-sorted. Then, a sequence with a radius $k = \mathrm{O}(n)$ is called a roughly sorted sequence. Yoshihide and Derick [20] designed two k-sorted algorithms, namely, k-Bubblesort and k-Quicksort, which generalized bubble sort and quick sort. Altman and Chlebus [1] studied sorting on a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) when the input is roughly sorted. They proved that there is an algorithm that can run on a CRCW PRAM using a linear number of processors that sorts each sequence $\overline{a}$ in time $\mathrm{O}(\log k)$, where k is the radius of $\overline{a}$.

k-sorted algorithms generalize sorting algorithms, and almost descending algorithms also generalize sorting algorithms. The difference is that they use different ways to generalize sorting algorithms and have different properties. k-sorted algorithms aim to speed up sorting with the roughly sorted property. However, our aim is to reduce the number of movements and remain stable in sorting; thus, we focus on the indistinguishability property in some special conditions. A common point is that both types of algorithms achieve their aims through modifying sorting algorithms, such as bubble sort and quick sort. In addition, all k-sorted sequences can be included in almost ascending sequences (the reverse of almost descending sequences; see Definition 2). For example, [200, 300, 100, 500, 400] is not a sorted sequence but a 3-sorted sequence, and it is also an almost ascending sequence with parameter 201. Nevertheless, we are only interested in almost ascending/descending sequences with small parameters C, while most k-sorted sequences are almost ascending sequences with large parameters C, which are too large for the items to be indistinguishable. Therefore, almost ascending/descending sequences with small parameters have the same properties as k-sorted sequences.

C. NEARLY SORTED

A sequence is nearly sorted if it requires few operations to sort it or it was created from a sorted sequence with a few perturbations. Cook *et al.* [7] defined the sortedness ratio as $\frac{k}{N}$ for a nearly sorted sequence with N elements, where k is the minimum number of elements that can be removed to leave the remaining sequence sorted. They compared five classic sorting algorithms on sequences of various sizes (50, 100, 200, 500, 1000, 2000) with sortedness ratios of 0.00, 0.02, 0.04, $\cdots$, 0.20. The test results indicated that straight insertion sort is the best sorting algorithm for small or very nearly sorted sequences. Based on their experiments, Cook *et al.* [7] developed a new sorting algorithm for nearly sorted sequences. It is a novel combination of straight insertion sort and quickersort with merging, which performs better than straight insertion sort. Castro [6] studied how to generate nearly sorted sequences, since such sequences are necessary for generating benchmark test sets for a series of important computational problems beyond sorting, for example, error-correcting graph isomorphisms.

Similar to roughly sorted sequences, nearly sorted sequences are included in almost ascending/descending sequences if the parameter C is sufficiently large. Even so, many nearly sorted sequences may be almost ascending/descending sequences with small parameters C. For example, in Problem 1, the sequence [1.852, 1.854, 1.8, 1.751, 1.752] is a nearly sorted sequence with a sortedness ratio of 0.4, and it is also an almost descending sequence with parameter 0.005. Therefore, if a nearly sorted sequence is an almost ascending/descending sequence with a small parameter C, they have similar properties, which means that such sequences are also useful in error-correcting graph isomorphisms.

D. ONLINE SORTING

In classic sorting problems, we know all the inputs initially, and the inputs are static. However, in online conditions, the inputs are dynamic, and we may not know all the changes at a certain time. Anagnostopoulos *et al.* [2] studied a new computational model in which the data change gradually, and the goal of their algorithm was to compute an approximate solution. Their idea was inspired by the online voting website Bix (owned by Yahoo), web search, recommendation systems, and online ad selection. They focused on fundamental problems of sorting and selection: that a good ranking in the past may not remain good and that ranking changes are typically gradual over time. The ranking system aims to track the changing perception of rankings by selecting feedback to request from the user. Let π^t be the true ordering at time t, and let $\tilde{\pi}^t$ be a sequence that is generated by an approximation algorithm. This algorithm has only limited access to the changes. Anagnostopoulos *et al.* used the Kendall tau distance to measure how close an estimate is to the true ordering.

The Kendall tau distance $KT(\pi_1, \pi_2)$ between permutations π_1 and π_2 is defined as follows:

$$KT(\pi_1, \pi_2) = |\{(x, y) : x <_{\pi_1} y \wedge y <_{\pi_2} x\}|.$$

Anagnostopoulos *et al.* stated that no algorithm can guarantee that at every time step, the distance between π^t and $\tilde{\pi}^t$ is less than $O(n)$. They proposed an algorithm that can guarantee with high probability that this distance is at most O(nlnlnn).

Let S^t be an almost descending sequence with parameter C at time t, and let $T = S^t + R$, where R is a random sequence with $R[i] \in [-M, M]$ for some small positive number M. Suppose S^{t+1} is the corresponding almost descending sequence of T. According to our test (see Figure 1 and Figure 2), if C is sufficiently large or M is sufficiently small, fewer than n movements are needed to modify T to S^{t+1}, which means that the Kendall tau distance between T and S^{t+1} is quite small. Therefore, if the almost descending sequence is adequate in dynamic sorting, then we can maintain a low Kendall tau distance during the sorting process.

E. SORTING IN NOISY ENVIRONMENTS

Fault tolerance is an important consideration in large systems since noisy information always exists in some conditions. Thus, it is necessary to devise algorithms that work despite unreliable information. Feige *et al.* [11] studied noisy Boolean decision trees and noisy comparison trees (such as are used in sorting, selection and searching). Makarychev *et al.* [16] investigated how to sort noisy data with partial information. According to our results, almost descending algorithms have fault tolerance property, which can counteract some noise. For example, almost descending algorithms can be used in PageRank [5], since the PageRank of pages changes from time to time and can be considered a dynamic almost descending sequence S with some parameter C. Web spam [19], search engine optimization (SEO) [17] and similar noisy information may also change the values of PageRank, and this information can be regarded as a noise sequence R. Then, the problem is how to make $S = S + R$ almost descending again in dynamic conditions. Choosing the proper parameter C can avoid unnecessary movement when some elements are indistinguishable in dynamic sorting. This is why almost descending algorithms can reduce the number of movements and are more stable.

VII. CONCLUSION

In this paper, we consider a realistic sorting problem, where we do not or cannot distinguish two elements unless they have large enough differences to be perceived. We deem such elements indistinguishable and thus do not need to adjust their positions in sorting. In this regard, we define the notion of an almost descending sequence with parameter C, which sets the bound for measuring whether two elements are indistinguishable. Then, we provide three algorithms to generate almost descending sequences based on bubble sort, selection sort and quick sort. Compared to the three original algorithms, our almost descending algorithms have two advantages: a reduced number of movements and enhanced stability. The number of movements for almost descending sequences decreases, as demonstrated in Corollary 1 and Table 1. In addition, almost descending sequences are more stable, as demonstrated in Corollary 2, Figure 1 and Figure 2. This means that they need fewer adjustments to become almost descending again if the values of some elements change slightly. Therefore, our algorithms can be used to sort noisy data, and the parameter C is a bound that measures noise. Our algorithms, which have fault tolerance to some extent, can maintain relatively steady sorting results in dynamic conditions. If a nearly sorted sequence with a low sortedness ratio is an almost ascending/descending sequence with a small parameter C, it is also useful in error-correcting graph isomorphisms.

REFERENCES

[1] T. Altman and B. S. Chlebus, "Sorting roughly sorted sequences in parallel," *Inf. Process. Lett.*, vol. 33, no. 6, pp. 297–300, Feb. 1990.

[2] A. Anagnostopoulos, R. Kumar, M. Mahdian, and E. Upfal, "Sorting and selection on dynamic data," *Theor. Comput. Sci.*, vol. 412, no. 24, pp. 2564–2576, May 2011.

[3] T. Araki and Y. Kikuchi, "Hamiltonian laceability of bubble-sort graphs with edge faults," *Inf. Sci.*, vol. 177, no. 13, pp. 2679–2691, Jul. 2007.

[4] M. Aumüller, M. Dietzfelbinger, and P. Klaue, "How good is multi-pivot quicksort?" *ACM Trans. Algorithms*, vol. 13, no. 1, pp. 1–47, 2016.

[5] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Comput. Netw. ISDN Syst.*, vol. 30, nos. 1–7, pp. 107–117, Apr. 1998.

[6] V. Estivill-Castro, "Generating nearly sorted sequences—The use of measures of disorder," *Electron. Notes Theor. Comput. Sci.*, vol. 91, pp. 56–95, Feb. 2004.

[7] C. R. Cook and D. J. Kim, "Best sorting algorithm for nearly sorted lists," *Commun. ACM*, vol. 23, no. 11, pp. 620–624, Nov. 1980.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[9] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 3. Reading, MA, USA: Addison-Wesley, 1973.

[10] S. Edelkamp, A. Weiß, and S. Wild, "QuickXsort: A fast sorting scheme in theory and practice," *Algorithmica*, vol. 82, no. 3, pp. 509–588, Mar. 2020.

[11] U. Feige, D. Peleg, P. Raghavan, and E. Upfal, "Computing with unreliable information," in *Proc. 22nd STOC*, 1990, pp. 128–137.

[12] C. C. Foster, "Sorting almost ordered arrays," *Comput. J.*, vol. 11, no. 2, pp. 134–137, Aug. 1968.

[13] S. Goldwasser and M. Bellare, "Lecture notes on cryptography," Cambridge, MA, USA, Jul. 2008, p. 41. [Online]. Available: https://cseweb.ucsd.edu/mihir/papers/gb.pdf

[14] U. E. Kocamaz, "Increasing the efficiency of quicksort using a neural network based algorithm selection model," *Inf. Sci.*, vol. 229, pp. 94–105, Apr. 2013.

[15] E. V. Konstantinova and A. N. Medvedev, "Small cycles, generalized prisms and Hamiltonian cycles in the bubble-sort graph," *Inf. Process. Lett.*, vol. 168, Jun. 2021, Art. no. 106094.

[16] K. Makarychev, Y. Makarychev, and A. Vijayaraghavan, "Sorting noisy data with partial information," in *Proc. 4th Conf. Innov. Theor. Comput. Sci. (ITCS)*, 2013, pp. 515–528.

[17] R. A. Malaga, "Worst practices in search engine optimization," *Commun. ACM*, vol. 51, no. 12, pp. 147–150, Dec. 2008.

[18] G. Manis, M. Aktaruzzaman, and R. Sassi, "Bubble entropy: An entropy almost free of parameters," *IEEE Trans. Biomed. Eng.*, vol. 64, no. 11, pp. 2711–2718, Nov. 2017.

[19] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly, "Detecting spam web pages through content analysis," in *Proc. 15th Int. Conf. World Wide Web*, Edinburgh, Scotland, May 2006, pp. 83–92.

[20] I. Yoshihide and W. Derick, "Roughly sorting: A generalization of sorting," *J. Inf. Process.*, vol. 14, no. 1, pp. 36–42, 1991.

[21] S. L. Zhao and R. X. Hao, "The fault tolerance of (n,k)-bubble-sort networks," *Discrete Appl. Math.*, vol. 285, pp. 204–211, Oct. 2020.

YUYIN YU received the B.S. degree in computer science from Henan Normal University, in 2007, and the Ph.D. degree from the Institute of Information Engineering, Chinese Academy of Sciences, in 2013. He is currently an Associate Professor with Guangzhou University. His research interests include algorithms, Boolean functions, and related areas.

XUAN XIAO received the Ph.D. degree in management from the Harbin Institute of Technology, China. She is currently an Assistant Professor at the School of Management, Guangzhou University. Her research interests include social networking service and open source software.

• • •