EAI Endorsed Transactions
on Industrial Networks and Intelligent Systems
Research Article EAI.EU

Kendall tau sequence distance: Extending Kendall tau from ranks to sequences

Vincent A. Cicirello*

Computer Science, Stockton University, 101 Vera King Farris Drive, Galloway, NJ 08205

Abstract

An edit distance is a measure of the minimum cost sequence of edit operations to transform one structure into another. Edit distance can be used as a measure of similarity as part of a pattern recognition system, with lower values of edit distance implying more similar structures. Edit distance is most commonly encountered within the context of strings, where Wagner and Fischer's string edit distance is perhaps the most well-known. However, edit distance is not limited to strings. For example, there are several edit distance measures for permutations, including Wagner and Fischer's string edit distance since a permutation is a special case of a string. However, another edit distance for permutations is Kendall tau distance, which is the number of pairwise element inversions. On permutations, Kendall tau distance is equivalent to an edit distance with adjacent swap as the edit operation. A permutation is often used to represent a total ranking over a set of elements. There exist multiple extensions of Kendall tau distance from total rankings (permutations) to partial rankings (i.e., where multiple elements may have the same rank), but none of these are suitable for computing distance between sequences. We set out to explore extending Kendall tau distance in a different direction, namely from the special case of permutations to the more general case of strings or sequences of elements from some finite alphabet. We name our distance metric Kendall tau sequence distance, and define it as the minimum number of adjacent swaps necessary to transform one sequence into the other. We provide two $O(n \lg n)$ algorithms for computing it, and experimentally compare their relative performance. We also provide reference implementations of both algorithms in an open source Java library.

Received on 08 February 2020; accepted on 02 April 2020; published on 07 April 2020

Keywords: edit distance, Kendall tau, pattern recognition, sequences, similarity, strings

doi:10.4108/XX.X.X.XX

1. Introduction

There exists a wide variety of metrics for computing the distance between permutations (Campos *et al.*, 2005; Cicirello, 2016, 2018, 2019; Cicirello and Cernera, 2013; Fagin *et al.*, 2003; Martí *et al.*, 2005; Meilă and Bao, 2010; Ronald, 1995, 1997, 1998; Sevaux and Sörensen, 2005; Sörensen, 2007). The different permutation metrics that are available consider different characteristics of the permutation depending upon what it represents (e.g., a mapping between two sets, a ranking over the elements of a set, or a path through a graph). There is at least one instance where a metric on strings is suggested for permutations. Specifically, Sörensen (2007) suggested using string edit distance to measure distance between permutations. The specific edit distance suggested by Sörensen was the string edit distance of Wagner and Fischer (1974). In general, the edit distance between two structures is the minimum cost sequence of edit operations to transform one structure into the other. Wagner and Fischer's string edit distance is the minimum cost sequence of edit operations to transform one string into the other where the edit operations are element removals, insertions, and replacements. The usual algorithm for computing it is the dynamic programming algorithm of Wagner and Fischer (1974), which has a runtime of $O(nm)$ where n and m are the lengths of the strings (in the case of permutations, runtime is $O(n^2)$ since lengths are the same).

In this paper, we begin with a metric on permutations, and adapt it to measure the distance between

*Corresponding author. Email: vincent.cicirello@stockton.edu

sequences (i.e., strings, arrays, or any other sequential data). The specific metric that we adapt to sequences is Kendall tau distance. Kendall tau distance is a metric defined for permutations that is itself an adaptation of Kendall tau rank correlation (Kendall, 1938). As a metric on permutations, Kendall tau distance assumes that a permutation represents a ranking over some set (e.g., an individual's preferences over a set of songs or books, etc), and is the count of the number of pairwise element inversions. We review Kendall tau distance, for permutations, in Section 2, along with existing extensions for handling partial rankings (i.e., instead of a permutation or total ordering, partial orderings with tied ranks are compared).

In the case of permutations, where each element of the set is represented exactly one time in each permutation, Kendall tau distance is the minimum number of adjacent swaps necessary to transform one permutation into the other. Thus, in the case of permutations, Kendall tau distance is an edit distance where the edit operations are adjacent swaps. Due to this relationship, it is sometimes referred to as bubble sort distance, since bubble sort functions via adjacent element swaps. However, as soon as you leave the realm of permutations, existing forms of Kendall tau no longer correspond to an adjacent swap edit distance. We provide an example of this in Section 2.5.

In the case of comparing partial rankings, the existing extensions of Kendall tau distance to partial rankings are fine. However, if we are comparing sequences (e.g., strings, arrays of data points, etc) that do not represent a ranking, then the partial ranking versions of Kendall tau distance do not apply. We propose a new extension of Kendall tau distance for sequences in Section 3. We call it Kendall tau sequence distance, and show that it meets the requirements of a metric. It is applicable for computing the distance between pairs of sequences, where both sequences are of the same length, and consist in the same set of elements (i.e., duplicates are allowed, but both sequences must have the same duplicated elements). It is otherwise applicable to strings over any alphabet or any other form of sequence (such as an array of integers or an array of floating-point values, etc). We argue that it is more relevant as a measure of array sortedness than the existing partial ranking adaptations of Kendall tau. In Section 3.3, we provide two $O(n \lg n)$ algorithms for computing Kendall tau sequence distance.

We implemented both algorithms in Java, and we have added those reference implementations to JavaPermutationTools (JPT), an open source Java library of data structures and algorithms for computation on permutations and sequences (Cicirello, 2018), which can be found at https://jpt.cicirello.org/. In Section 4, we experimentally compare the relative performance of the two algorithms. The code to replicate these experiments is also available in the code repository of the JPT.

2. Kendall tau distance for permutations

2.1. Notation and Assumptions

Without loss of generality, we will assume a permutation of length n is a permutation of the integers in the set $S = 1, 2, \ldots, n$. Let $\sigma(i)$, where $i \in S$, be the position of element i in the permutation σ. If the permutation is a ranking over a set of n objects, then $\sigma(i)$ represents the rank of object i in that ranking. Let $p(r)$, where $r \in S$, be the element in position r of the permutation (or with rank r). Our notation assumes that the index into the permutation begins at 1.

The σ and p are two alternative representations of the permutation. They are related as follows: $\sigma(i) = r \iff p(r) = i$. Throughout the paper, we will use whichever is more convenient in the given context.

We will initially assume that permutations (whether defined with σ or with p) are true permutations. That is, we assume $\sigma(i) = \sigma(j) \iff i = j$ and also that $p(r_1) = p(r_2) \iff r_1 = r_2$. Therefore, if the application is one of rankings, we assume that there are no ties. In other words, two objects have the same rank only if they are the same object; and each object has only one rank. We relax this assumption later in Section 2.4.

2.2. Kendall tau rank correlation

Kendall tau distance for permutations is strongly based on the Kendall tau rank correlation coefficient. Consider two permutations σ_1 and σ_2. The Kendall tau rank correlation coefficient (Kendall, 1938) is defined as:

$$\tau(\sigma_1, \sigma_2) = \frac{2}{n(n-1)} \sum_{i,j \in S \wedge i<j} \text{sign}(\sigma_1(i) - \sigma_1(j)) \, \text{sign}(\sigma_2(i) - \sigma_2(j)). \quad (1)$$

The function sign(x) evaluates to 1 if x is positive, and -1 if x is negative. The summation has a maximum value of $n(n-1)/2$, which occurs when $\sigma_1 = \sigma_2$; and the summation has a minimum value of $-n(n-1)/2$, which occurs when σ_1 is the reverse of σ_2. The $2/(n(n-1))$ term scales such that $\tau \in [-1, 1]$.

Another way of expressing it is as follows:

$$\tau(\sigma_1, \sigma_2) = \frac{2}{n(n-1)}(|C| - |D|), \quad (2)$$

where C is the set of concordant pairs, defined as:

$$C = \{(i,j) \in S \times S \mid i < j \wedge (\sigma_1(i) < \sigma_1(j) \wedge \sigma_2(i) < \sigma_2(j) \vee \sigma_1(i) > \sigma_1(j) \wedge \sigma_2(i) > \sigma_2(j))\}, \quad (3)$$

where D is still the set of discordant pairs, as previously defined in Equation 4. Note the strict $<$ and $>$ in the definition of D, and that a tie within either permutation is not a discordant pair. E is the set of pairs that are ties in one permutation, but not the other (i.e., one ranking considers the objects equivalent, but the other does not). Therefore, E is defined as:

$$E = \{(i,j) \in S \times S \mid i < j \wedge (\sigma_1(i) = \sigma_1(j) \wedge \sigma_2(i) \neq \sigma_2(j) \vee \sigma_1(i) \neq \sigma_1(j) \wedge \sigma_2(i) = \sigma_2(j))\}. \quad (9)$$

Fagin *et al.* (2006) showed that $K^{(p)}$ is a metric when $0.5 \leq p \leq 1$, and that it is what they termed a "near metric" when $0 < p < 0.5$, and that it is not a distance when $p = 0$. We do not use their "near metric" concept here so we leave it to the interested reader to consult Fagin *et al.* (2006).

2.5. Partial ranking Kendall tau distance ≠ adjacent swap edit distance

As a distance metric on partial rankings, the Kendall distance with penalty parameter p of Fagin *et al.* (2006) is an effective choice, and commonly used in the context of comparing partial rankings. However, it is not adjacent swap edit distance. Consider the following illustrative example. Let $\sigma_1 = [1,2,3,1,1,2,2]$ and $\sigma_2 = [3,2,1,2,1,2,1]$. In this case, the set of discordant pairs is $D = \{(1,2),(1,3),(1,6),(1,7),(2,3),(3,4),(3,6),(4,7)\}$, and the set $E = \{(1,4),(1,5),(2,4),(2,7),(3,5),(3,7),(4,5),(4,6),(5,7),(6,7)\}$. Thus, $K^{(p)}(\sigma_1,\sigma_2) = 8 + 10p$ (Equation 8).

You can compute $|D|$ and $|E|$ without actually computing the sets D and E via the approach of Knight (1966) based on sorting. Let $T = [(1,3),(2,2),(3,1),(1,2),(1,1),(2,2),(2,1)]$. Sort T by first component of tuples, breaking ties via the second components, and obtain: $T' = [(1,1),(1,2),(1,3),(2,1),(2,2),(2,2),(3,1)]$. You can finally sort T' via mergesort (or another $O(n \lg n)$ sort), with the sort modified to count inversions. In this case, there are 8 inversions in T', which is equal to $|D|$. It is also straightforward enough to compute $|E|$.

The $|D|$ in this example is the minimum number of adjacent swaps necessary to sort T'. However, it is not the minimum number of adjacent swaps necessary to transform σ_1 into σ_2. That can be done with fewer than eight adjacent swaps. Specifically, it can be done via the following sequence of six adjacent swaps: $\sigma_1 = [1,2,3,1,1,2,2]$, $[1,3,2,1,1,2,2]$, $[3,1,2,1,1,2,2]$, $[3,2,1,1,1,2,2]$, $[3,2,1,1,2,1,2]$, $[3,2,1,2,1,1,2]$, $[3,2,1,2,1,2,1] = \sigma_2$.

Now, previously in Section 2.3, we saw that with full rankings (i.e., permutations) Kendall tau distance is equal to the minimum number of adjacent swaps to transform p_1 into p_2 (i.e., an adjacent swap edit distance on the p notation, where $p(r)$ yields the object with rank r). With partial rankings, we don't have the equivalent of p since multiple objects may have the same rank. One attempt might be to allow $p(r)$ to map to the set of objects with rank r. Thus, for the example of the prior paragraph, we'd have $p_1 = [\{1,4,5\},\{2,6,7\},\{3\}]$, and $p_2 = [\{3,5,7\},\{2,4,6\},\{1\}]$. Transforming p_1 to p_2 via adjacent swaps (if we define an adjacent swap in this context as swapping two elements in adjacent sets) can be done with four such swaps.

Also in Section 2.3, for full rankings, we saw that Kendall tau distance is equal to the minimum number of applications of an operation that exchanges the ranks of two elements whose ranks differ by 1. For this example, a sequence of four such operations can transform $\sigma_1 = [1,2,3,1,1,2,2]$ into $\sigma_2 = [3,2,1,2,1,2,1]$. That sequence is as follows: $\sigma_1 = [1,2,3,1,1,2,2]$, $[1,2,3,2,1,2,1]$, $[1,3,2,2,1,2,1]$, $[2,3,1,2,1,2,1]$, $[3,2,1,2,1,2,1] = \sigma_2$. This is equivalent to our redefinition of $p(r)$ to the set of elements with rank r.

There is no interpretation where $K^{(p)}$ or any other partial ranking variation of Kendall tau distance that is based on the number of discordant pairs is equivalent to an adjacent swap edit distance. The example of this section illustrates this in that there are eight discordant pairs (thus $K^{(p)} \geq 8$ unless p is negative) while less than eight adjacent swaps is sufficient for sorting the permutation (either 6 or 4 depending upon the interpretation of "adjacent swap" and the representation to which it is applied).

2.6. Positions of elements in a sequence are not ranks

If the sequences we are comparing do not define rankings, then the partial ranking variants of Kendall tau distance are not applicable as it would be arbitrary to impose a ranking interpretation upon them, and also likely to lead to a nonsensical interpretation. For example, consider the string s: "abacab". It would be arbitrary to impose a lexicographical order of the characters as if they are ranks (e.g., "a" as 1, "b" as 2, etc), such as transforming s to $\sigma = [1,2,1,3,1,2]$. Or, if you consider position in the sequence to be an element's rank, then you'd have something meaningless like "a" is simultaneously ranked first, third, and fifth.

3. Kendall tau sequence distance

3.1. Notation and Assumptions

Let s be a sequence of length n, where $s(i) \in \Sigma$ for some alphabet Σ and $i \in \{0,1,\ldots,n-1\}$. The alphabet Σ can be a character set for some language, but can also be the set of integers, the set of real numbers, the set of

```
τ_S(s_1, s_2)
1.    if |s_1| ≠ |s_2|
2.       return error: unequal length sequences
3.    Let n = |s_1|
4.    Let S be a sorted copy of s_1
5.    Let M be a new array of length n
6.    M[0] ← 0
7.    for i = 1 to n − 1 do
8.       if S[i] = S[i − 1]
9.          M[i] ← M[i − 1]
10.      else
11.         M[i] ← M[i − 1] + 1
12.   Let B_1 and B_2 be arrays of length M[n − 1] + 1 of initially empty queues
13.   for i = 0 to n − 1 do
14.      Let j be an index into S, such that S[j] = s_1[i].
15.      Let k be an index into S, such that S[k] = s_2[i].
16.      if k is undefined
17.         return error: sequences contain different elements
18.      Add i to the tail of queue B_1[M[j]].
19.      Add i to the tail of queue B_2[M[k]].
20.   Let P be an array of length n
21.   for i = 0 to M[n − 1] do
22.      if lengths of queues B_1[i] and B_2[i] are different
23.         return error: sequences contain different number of copies of an element
24.      while queue B_1[i] is not empty do
25.            Remove the head of queue B_1[i] storing it in h_1.
26.            Remove the head of queue B_2[i] storing it in h_2.
27.            P[h_1] ← h_2
28.   Let I be the number of inversions in P.
29.   return I
```

Figure 1. Algorithm for computing τ_S

represents character $s_1[i]$. This requires a search of S in step 14, which can be implemented with binary search in $O(f_c(m) \lg n)$ time since S is in sorted order. The buckets are represented with queues to easily maintain the order that duplicate copies of an element appear in the original sequence. Adding to the tail of a queue is a constant time operation. B_1 is an array of the buckets for s_1. In a similar manner, a bucket sort of s_2 is performed, and B_2 is an array of the corresponding buckets. The block in lines 12–19 has a total cost of $O(f_c(m)\, n \lg n)$ since the loop of line 13 iterates n times and the binary searches in lines 14 and 15 have a runtime of $O(f_c(m) \lg n)$.

Lines 20–27 iterates over the buckets, mapping the elements of s_2 to the corresponding elements of s_1. The resulting mapping is a permutation P of the integers in $\{0, 1, \ldots, n-1\}$. Where there are duplicates of a specific character of the alphabet Σ, they are mapped in the order of appearance. For example, if character c appears in positions 2, 5, 18 of s_1 and in positions 4, 7, 22 of s_2, then the permutation P will have the following corresponding entries: $P[2] = 4, P[5] = 7, P[18] = 22$. The nested loops in lines 21 and 24 iterate exactly one time for each sequence index, i.e., a total of n executions of the body (lines 25–27) of the nested loops. The body of which contains only constant time operations. Thus, the runtime of lines 20–27 is $O(n)$.

Counting permutation inversions (line 28) is done in $O(n \lg n)$ time with a modified mergesort.

The runtime of this first algorithm is therefore $O(f_c(m)\, n \lg n)$ due to the sort in line 4, and the block of lines 12–19. This is worst case as well as average case. If the sequences contain values of a primitive type, such as ASCII or Unicode characters, primitive integers, primitive floating-point numbers, etc, then $f_c(m) = O(1)$, and thus the runtime of the algorithm simplifies to $O(n \lg n)$.

Algorithm 2. Our second algorithm for computing τ_S is found in Figure 2. It is similar in function to the first algorithm, but generates the mapping from unique sequence elements to integers differently. Specifically, it uses a hash table, H (initialized in line 4). Lines 5–9 populates that hash table. The loop in that block iterates

should be no worse than linear in the size of the objects. Thus, the runtime for Algorithm 1 is no worse than $O(mn \lg n)$, which is higher order than the runtime of Algorithm 2. However, it is possible that a comparison of objects of size m may run faster than a hash of an object of size m since a comparison may short circuit on an object attribute difference found early in the comparison. Therefore, Algorithm 1 may be the preferred algorithm for sequences of large objects. We explore this experimentally in the next section.

Furthermore, the runtime, $O(f_h(m)\, n + n \lg n)$, of Algorithm 2 is no worse than the runtime, $O(f_c(m)\, n \lg n)$, of Algorithm 1 provided that $\frac{f_h(m)}{f_c(m)} = O(\lg n)$. So any advantage Algorithm 1 may have on sequences of large objects diminishes for large sequence lengths.

4. Experiments

In this section, we experimentally explore the relative performance of the two algorithms for computing Kendall tau sequence distance. In Section 4.1 we describe our reference implementations of the two algorithms, and explain our experimental setup in Section 4.2. Then, in Section 4.3, we experimentally compare the two algorithms on sequences of primitive values, such as strings of Unicode characters, arrays of integers, and arrays of floating-point values. Section 4.4 compares the performance of the algorithms on arrays of objects of varying sizes..

4.1. Reference Implementations in Java

We provide reference implementations of both algorithms from the previous section in an open source Java library available at: `https://jpt.cicirello.org`. Specifically, the class KendallTauSequenceDistance, in the package org.cicirello.sequences.distance, implements both algorithms. The implementations support computing the Kendall tau sequence distance between Java String objects, arrays of any of Java's primitive types (i.e., char, byte, short, int, long, float, double, boolean), as well as computing the distance between arrays of any object type.

For arrays of objects, the implementation of Algorithm 1 requires the objects to be of a class that implements Java's Comparable interface, since the sort step requires comparing pairs of elements for relative order; while Algorithm 2 requires the objects to be of a class that overrides the hashCode and equals methods of Java's Object class since it relies on a hash table.

To compute the distance between arrays of objects, our implementation of Algorithm 2 uses Java's HashMap class for the hash table, and the default maximum load factor of 0.75. To eliminate the need to rehash to maintain that load factor, we initialize the HashMap's size to $\lceil \frac{n}{0.75} \rceil$, where n is the sequence length. In this way, even if every element is unique, no rehashing will be needed.

For computing the distance between arrays of primitive values, as well as for computing the distance between String objects, our implementation of Algorithm 2 uses a set of custom hash table classes (one for each primitive type). All of these hash tables (except the one for bytes) use chaining with single-linked lists for the buckets. The size of the hash table is set, as above, based on the length of the array to ensure that the load factor is no higher than 0.75. Additionally, we use a table size that is a power of two to enable using a bitwise-and operation rather than a mod to compute indexes. However, we limit the table size to no greater than 2^{16} for the two 16-bit primitive types (char and short), and to no greater than 2^{30} for all other types. The integer primitive types are hashed in the obvious way for each of the three such types that use 16 to 32 bits (char, short, int). Specifically, char and short values are cast to 32-bit int values. We hash long values with an xor of the right and left 32-bit halves. We hash a float using its 32 bits as an int. We hash a double with an xor of its left and right 32-bit halves, using the result as a 32-bit int. Java's Float and Double classes provide methods for converting the bits of float and double values to int and long values, respectively. We otherwise do not use Java's wrapper classes for the primitive types.

In the case of arrays of bytes, our implementation of Algorithm 2 uses a simple array of length 256 as the hash table, one cell for each of the possible byte values, regardless of byte sequence length. In this way, there are never any hash collisions when computing the distance between arrays of byte values.

For arrays of booleans, we handle the mapping to integers differently regardless of algorithm choice, since it is straightforward to map all false values to 0 and all true values to 1 in linear time.

The KendallTauSequenceDistance class can be configured to use either of the two algorithms. The default is Algorithm 2, since as we will see in Sections 4.3 and 4.4, it is always faster for sequences of primitives and nearly always faster for arrays of objects.

4.2. Experimental Setup

Our experiments are implemented in Java 1.8, and we use the Java HotSpot 64-Bit Server VM, on a Windows 10 PC. Our test system has 8GB RAM, with a quad-core AMD A10-5700 APU processor with 3.4 GHz clock speed.

4.3. Results on Sequences of Primitives

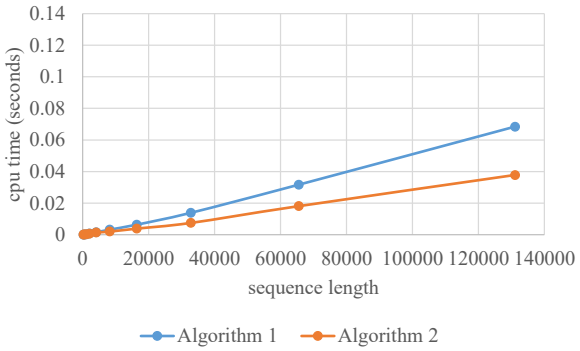
Strings. Our first set of results is on computing Kendall tau sequence distance between Java String objects.

Algorithm 2 is consistently faster than Algorithm 1 for computing Kendall tau sequence distance between arrays of 32-bit integers, independent of alphabet size and array length.

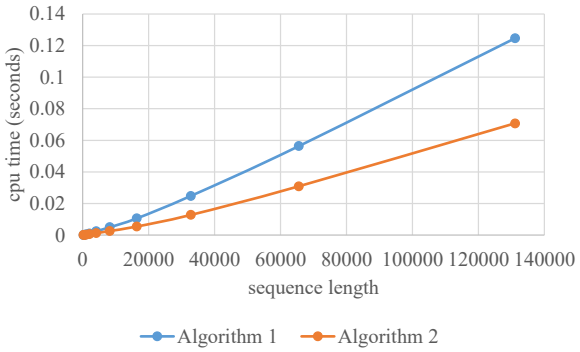
Just as in the case of Strings, both algorithms run faster with the smaller alphabet size than with a larger alphabet size. The explanation is the same: smaller alphabet means more duplicate copies of elements, which means sorting is faster (Algorithm 1) and hash table lookups are faster due to reduced load factor (Algorithm 2).

Arrays of Floating-Point Numbers. In this last case of sequences of primitives, we consider arrays of 64-bit double-precision floating point numbers, Java's double type. We consider the same array lengths and alphabet sizes as the previous cases, but now the alphabet is a set of floating-point values. Specifically, the alphabet Σ contains $1.0x$ where x is the first $|\Sigma|$ non-negative integers.

Figure 5 shows the results for two of the alphabet sizes: 256 and 65536. Just as in the previous two cases, Algorithm 2 is consistently faster than Algorithm 1 for computing Kendall tau sequence distance between arrays of 64-bit double-precision floating-point numbers, independent of alphabet size and array length. And again, runtime is longer for both algorithms with larger alphabet size for the same reasons as before.



(a) $|\Sigma| = 256$



(b) $|\Sigma| = 65536$

Figure 5. Average CPU time for sequences of 64-bit doubles from varying size alphabets.

4.4. Results on Sequences of Objects

In this section, we explore the performance of the algorithms on computing distance between sequences of objects. Specifically, we use arrays of Java String objects. For example, consider sequences s_1 and s_2 as follows:

$$s_1 = [\text{“hello”}, \text{“world”}, \text{“hello”}, \text{“blue”}, \text{“sky”}], \quad (12)$$

$$s_2 = [\text{“hello”}, \text{“blue”}, \text{“sky”}, \text{“hello”}, \text{“world”}]. \quad (13)$$

These sequences are a Kendall tau sequence distance of 5 from each other. One sequence of adjacent swaps of length five that transforms s_1 into s_2, starts by swapping "blue" to the left twice, then swaps "sky" twice to the left, and finally swaps "world" with the right most of the two copies of "hello."

We use String objects for this set of experiments because it is easy to vary the size of a String object; and it is also relatively easy to create a case where both a hash and a comparison have cost $O(m)$ where m is object size (in this case length) as well as a case where a comparison costs significantly less than a hash.

We consider array lengths $L \in \{2^8, 2^9, \ldots, 2^{14}\}$, and alphabet size $|\Sigma| = 256$, where the alphabet is a set of String objects. We consider the following object sizes $m \in \{2^0, 2^1, \ldots, 2^{11}\}$. Computing a hash of a String of length m has cost $O(m)$ regardless of String content. We consider two cases of String formation. In the first case, each of the 256 Strings in Σ begin with $m-1$ copies of Unicode character 0, and only differ in the last character. In this case, all comparisons also cost $O(m)$ since linear iteration over the entire String object is required to determine how they differ. We will refer to this case as *high cost comparisons (HCC)*. In the second case, each of the 256 Strings in Σ is m copies of the same character, but each of the 256 Strings use a different character. Comparisons in this case either immediately short circuit on the first character (if they are different) or require linear iteration if they are identical. We will refer to this case as *low cost comparisons (LCC)*. For each combination of L, m, and HCC vs LCC, we generate 10 pairs of sequences. Each pair contains the same set of objects, but in different random orders. We compute average CPU time across the 10 pairs of sequences.

In Figures 6 and 7, we show average CPU time as a function of sequence length for arrays of String objects 32 characters and 2048 characters in length, respectively. Part (a) of each figure is the HCC case, and part (b) is the LCC case. For the small objects (Figure 6), Algorithm 2 is consistently faster for all sequence lengths in both the HCC and LCC cases,

object comparison, Algorithm 2 is still the preferred algorithm.

We provide reference implementations of both algorithms in the Java language. These implementations have been made available in an open source library. Our experiments confirm that Algorithm 2 is the faster algorithm under most circumstances. The code to replicate our experimental data is also available as open source.

References

Campos, V., Laguna, M. and Martí, R. (2005) Context-independent scatter and tabu search for permutation problems. *INFORMS Journal on Computing* **17**(1): 111–122. doi:10.1287/ijoc.1030.0057.

Cicirello, V.A. (2016) The permutation in a haystack problem and the calculus of search landscapes. *IEEE Transactions on Evolutionary Computation* **20**(3): 434–446. doi:10.1109/TEVC.2015.2477284.

Cicirello, V.A. (2018) JavaPermutationTools: A java library of permutation distance metrics. *Journal of Open Source Software* **3**(31): 950. doi:10.21105/joss.00950.

Cicirello, V.A. (2019) Classification of permutation distance metrics for fitness landscape analysis. In *Proceedings of the 11th International Conference on Bio-inspired Information and Communications Technologies* (ICST). doi:10.1007/978-3-030-24202-2_7.

Cicirello, V.A. and Cernera, R. (2013) Profiling the distance characteristics of mutation operators for permutation-based genetic algorithms. In *Proceedings of the 26th International Conference of the Florida Artificial Intelligence Research Society* (AAAI Press): 46–51.

Fagin, R., Kumar, R., Mahdian, M., Sivakumar, D. and Vee, E. (2006) Comparing partial rankings. *SIAM Journal on Discrete Math* **20**(3): 628–648.

Fagin, R., Kumar, R. and Sivakumar, D. (2003) Comparing top k lists. *SIAM Journal on Discrete Mathematics* **17**(1): 134–160.

Kendall, M.G. (1938) A new measure of rank correlation. *Biometrika* **30**(1/2): 81–93.

Knight, W.R. (1966) A computer method for calculating kendall's tau with ungrouped data. *Journal of the American Statistical Association* **61**(314): 436–439.

Martí, R., Laguna, M. and Campos, V. (2005) Scatter search vs. genetic algorithms: An experimental evaluation with permutation problems. In *Metaheuristic Optimization via Memory and Evolution* (Springer), 263–282.

Meilă, M. and Bao, L. (2010) An exponential model for infinite rankings. *Journal of Machine Learning Research* **11**: 3481–3518.

Ronald, S. (1995) Finding multiple solutions with an evolutionary algorithm. In *Proceedings of the IEEE Congress on Evolutionary Computation* (IEEE Press): 641–646.

Ronald, S. (1997) Distance functions for order-based encodings. In *Proceedings of the IEEE Congress on Evolutionary Computation* (IEEE Press): 49–54.

Ronald, S. (1998) More distance functions for order-based encodings. In *Proceedings of the IEEE Congress on Evolutionary Computation* (IEEE Press): 558–563.

Sevaux, M. and Sörensen, K. (2005) Permutation distance measures for memetic algorithms with population management. In *Proceedings of the Metaheuristics International Conference (MIC2005)*: 832–838.

Sörensen, K. (2007) Distance measures based on the edit distance for permutation-type representations. *Journal of Heuristics* **13**(1): 35–47. doi:10.1007/s10732-006-9001-3.

Wagner, R.A. and Fischer, M.J. (1974) The string-to-string correction problem. *Journal of the ACM* **21**(1): 168–173.