Computing Distances Between Partial Rankings

Mukul S. Bansal, David Fernández-Baca

Dept. of Computer Science, Iowa State University, Ames, IA 50011, USA

Abstract

We give two efficient algorithms for computing distances between partial rankings (i.e. rankings with ties). Given two partial rankings over n elements, and with b and c equivalence classes, respectively, our first algorithm runs in $O(n \log n / \log \log n)$ time, and the second in $O(n \log \min\{b, c\})$ time.

Key words: Algorithms, partial rankings, p-ratings, top-m lists, Kendall tau distance

1 Introduction

A *ranking* is an ordering of a set of elements indicating some sort of preference relationship among them. *Rank aggregation* is the problem of combining multiple rankings into a single, *aggregate*, ranking. Perhaps because it arises naturally in a range of diverse settings, including voting and internet search, the rank aggregation problem has a long history (see, for example, [1–3]).

In many important applications, rankings are only *partial*, in the sense that ties are allowed. This happens, for example, when only the top, say m, elements are ordered, while all the remaining elements are assumed to have rank $m + 1$. Another scenario that leads to rankings with ties arises when a set of elements, say hotels, are rated by assigning them an integer score from a finite range, say 0 through 5. The aggregation of partial rankings has received significant attention in recent years [1,4–6], and several distance measures have been proposed to compare partial rankings. Two such distance measures, $K^{(p)}$ and K_{Haus}, have been extensively studied and shown to have especially nice mathematical and algorithmic properties, particularly with regard to rank aggregation [1,5,6]. Both $K^{(p)}$ and K_{Haus} are generalizations of the well known Kendall tau distance; their definitions appear in

Email addresses: `bansal@cs.iastate.edu` (Mukul S. Bansal), `fernande@cs.iastate.edu` (David Fernández-Baca).

Section 2. The Kendall tau distance between two full rankings (over the same set of n elements) counts the number of pairwise disagreements between the two rankings, and can be computed in $O(n \log n / \log \log n)$ time by counting inversions [7].

Computing $K^{(p)}$ and K_{Haus} is useful not only for comparing partial rankings against each other, but also for aggregating them into a single ranking. Indeed, computing a median ranking (i.e., a ranking with minimum total distance to the input rankings) provides a mathematically sound way of combining multiple partial rankings into a single one [3,5]. Both $K^{(p)}$ and K_{Haus} can be computed naively in $O(n^2)$ time; however, the problem of computing these values efficiently has not been addressed so far. Here, we give two algorithms for computing $K^{(p)}$ and K_{Haus} between partial rankings: One of these runs in $O(n \log n / \log \log n)$ time, and the other in $O(n \log \min\{b, c\})$ time, where b, c are the number of equivalence classes (i.e. *buckets*) in the two partial rankings.

2 Basic Notation and Preliminaries

By and large, we follow the terminology of Fagin et al. from [5]. A *bucket order* is a linear order with ties. More formally, a bucket order is a transitive binary relation $\prec$ for which there are non-empty sets $B_1, \ldots, B_b$ (called buckets) that form a partition of the domain such that $x \prec y$ if and only if there are i, j with $i < j$ such $x \in B_i$ and $y \in B_j$. We say that bucket B_i *precedes* bucket B_j if $i < j$.

We associate a *partial ranking* with each bucket order, by letting $\sigma(x) = i$ when $x \in B_i$. We assume that all partial rankings have the same domain, denoted D. Without any loss in generality, we shall assume that $D = \{1, \ldots, n\}$. We say that x and y are *tied in* σ if $\sigma(x) = \sigma(y)$.

Let σ_1 and σ_2 be two partial rankings on the domain D with bucket sets $B_1, B_2, \ldots, B_b$, and $C_1, C_2, \ldots, C_c$ respectively (where bucket B_i precedes bucket B_{i+1} for $1 \leq i < b$, and bucket C_i precedes bucket C_{i+1} for $1 \leq i < c$). Let $\mathcal{P} = \{\{i, j\} \colon i \neq j \text{ and } i, j \in D\}$ be the set of unordered pairs of distinct elements from D. We partition $\mathcal{P}$ into the following three sets:

1. $\mathcal{D}(\sigma_1, \sigma_2)$ is the set of all $\{i, j\} \in \mathcal{P}$ such that i and j appear in different order in σ_1 and σ_2; that is, either $\sigma_1(i) < \sigma_1(j)$ and $\sigma_2(i) > \sigma_2(j)$ or vice versa.
2. $\mathcal{R}_1(\sigma_1, \sigma_2)$ is the set of all $\{i, j\} \in \mathcal{P}$ such that i and j are tied in σ_1 but not tied in σ_2; that is, $\sigma_1(i) = \sigma_1(j)$ and $\sigma_2(i) \neq \sigma_2(j)$.
3. $\mathcal{R}_2(\sigma_1, \sigma_2)$ is the set of all $\{i, j\} \in \mathcal{P}$ such that i and j are tied in σ_2 but not tied in σ_1; that is, $\sigma_2(i) = \sigma_2(j)$ and $\sigma_1(i) \neq \sigma_1(j)$.

Fagin et al. in [5] defined the following two distance measures between partial rankings. The *Kendall distance with penalty parameter p*, denoted by $K^{(p)}$, is defined to be $|\mathcal{D}(\sigma_1, \sigma_2)| + p \cdot (|\mathcal{R}_1(\sigma_1, \sigma_2)| + |\mathcal{R}_2(\sigma_1, \sigma_2)|)$ where p is a real number in the interval $[0, 1]$. The *Hausdorff distance based on Kendall distance*, denoted by K_{Haus}, is equal to $|\mathcal{D}(\sigma_1, \sigma_2)| + \max\{|\mathcal{R}_1(\sigma_1, \sigma_2)|, |\mathcal{R}_2(\sigma_1, \sigma_2)|\}$. [1]

The values of both $K^{(p)}(\sigma_1, \sigma_2)$ and $K_{Haus}(\sigma_1, \sigma_2)$ are easily obtained if we first compute the values $|\mathcal{D}(\sigma_1, \sigma_2)|$, $|\mathcal{R}_1(\sigma_1, \sigma_2)|$ and $|\mathcal{R}_2(\sigma_1, \sigma_2)|$. In the next section we show how to compute the values $|\mathcal{R}_1(\sigma_1, \sigma_2)|$ and $|\mathcal{R}_2(\sigma_1, \sigma_2)|$ in $O(n)$ time. In Section 4 we present two algorithms that compute the value $|\mathcal{D}(\sigma_1, \sigma_2)|$.

3 Preprocessing

Our algorithms assume that the following three preprocessing steps have been executed: (i) Relabel the domain D so that in σ_1 bucket B_1 is $\{1, \ldots, |B_1|\}$, B_2 is $\{|B_1| + 1, \ldots, |B_1| + |B_2|\}$, and so on. (ii) For each $i \in \{1, \ldots, c\}$, sort C_i in ascending order (according to the relabeled domain). (iii) Compute $|\mathcal{R}_1(\sigma_1, \sigma_2)|$ and $|\mathcal{R}_2(\sigma_1, \sigma_2)|$.

We claim that all three steps can be done in $O(n)$ time. This is easy to see for Step (i). To implement Step (ii), we rely on the fact that the buckets C_i, $1 \leq i \leq c$, partition $D = \{1, \ldots, n\}$ and that, given any element in D, we know (in $O(1)$ time) the bucket C_i to which it belongs. This allows us to obtain sorted versions of the sets C_i, $1 \leq i \leq c$, in $O(n)$ time simply by adding the elements of D in ascending order one at a time to the correct ordered set. We now show how to compute the value $|\mathcal{R}_2(\sigma_1, \sigma_2)|$ in $O(n)$ time; thus, by symmetry, $|\mathcal{R}_1(\sigma_1, \sigma_2)|$ can also be computed in $O(n)$ time. Our algorithm relies on the following result.

Lemma 3.1 *For every two partial rankings σ_1 and σ_2,*

$$|\mathcal{R}_2(\sigma_1, \sigma_2)| = \sum_{i=1}^{c} \left(\binom{|C_i|}{2} - \sum_{j=1}^{c_i} \binom{|C_{ij}|}{2} \right), \qquad (1)$$

where, for $i \in \{1, \ldots, c\}$, $C_{i1}, \ldots, C_{ic_i}$ is a partition of C_i into c_i maximal subsets such that, for $j \in \{1, \ldots, c_i\}$, every element of C_{ij} occurs in the same bucket of σ_1.

Proof: Let R denote the right-hand side of Equation (1). Consider each term of the outermost sum in R: The value of $\binom{|C_i|}{2}$ is the number of unordered pairs in bucket C_i, while the value of $\sum_j \binom{|C_{ij}|}{2}$ is the number of unordered pairs from the bucket C_i that appear in the same bucket in σ_1. Since each term of the outer sum considers different buckets of σ_2, R counts each pair in $\mathcal{R}_2(\sigma_1, \sigma_2)$ at most once. Therefore, $R \leq |\mathcal{R}_2(\sigma_1, \sigma_2)|$. Now consider some element $x = \{a, b\} \in \mathcal{R}_2(\sigma_1, \sigma_2)$. Then,

[1] The actual definition of K_{Haus} is more technical, but the measure was shown to be equal to $|\mathcal{D}(\sigma_1, \sigma_2)| + \max\{|\mathcal{R}_1(\sigma_1, \sigma_2)|, |\mathcal{R}_2(\sigma_1, \sigma_2)|\}$ in [5].

there exists some $i \in \{1, \ldots, c\}$ such that $a, b \in C_i$ and there is no $j \in \{1, \ldots, c_i\}$ such that $a, b \in C_{ij}$. Thus, x contributes at least one to R and, therefore, $R \geq |\mathcal{R}_2(\sigma_1, \sigma_2)|$. Altogether, this implies that $R = |\mathcal{R}_2(\sigma_1, \sigma_2)|$. ■

Theorem 3.1 *The value of $|\mathcal{R}_2(\sigma_1, \sigma_2)|$ can be computed in $O(n)$ time.*

Proof: We show that the right-hand side of Equation (1) can be evaluated in $O(n)$ time. For each $i \in \{1, \ldots, c\}$, the partition $C_{i1}, \ldots, C_{ic_i}$ of C_i can be built in $O(|C_i|)$ time as follows: Because of preprocessing Steps (i) and (ii), the elements in C_i are given as a list in increasing order. Hence, each C_{ij} is a single contiguous block of integers. The partition $C_{i1}, \ldots, C_{ic_i}$ can therefore be created by traversing through the elements of C_i in order, and assigning them to consecutive groups. Given $C_{i1}, \ldots, C_{ic_i}$, each term of the sum on the right-hand side of Equation (1) can be computed in $O(|C_i|)$ time. Hence, the total time to evaluate Equation (1) and to obtain the value of $|\mathcal{R}_2(\sigma_1, \sigma_2)|$ is $O(\sum_{i=1}^{c} |C_i|)$, which is $O(n)$. ■

4 Computing $|\mathcal{D}(\sigma_1, \sigma_2)|$

Our algorithms for computing $|\mathcal{D}(\sigma_1, \sigma_2)|$ are related to algorithms for the classical inversion counting problem [8]. Informally, the inversion counting problem is to find the number of pairs that are out of order in a given permutation on the numbers $1, \ldots, n$. The fastest known algorithm for inversion counting uses the subset rank data structure [7] and runs in $O(n \log n / \log \log n)$ time. In Section 4.1 we develop this idea further and show how to compute $|\mathcal{D}(\sigma_1, \sigma_2)|$ in $O(n \log n / \log \log n)$ time. Another approach to solve the inversion counting problem uses divide and conquer, yielding an $O(n \log n)$ algorithm (see, for example, [8]). In Section 4.2 we extend this approach to obtain an $O(n \log \min\{b, c\})$ algorithm to compute $|\mathcal{D}(\sigma_1, \sigma_2)|$.

4.1 An $O(n \log n / \log \log n)$ algorithm

This algorithm uses a data structure Ψ for the *subset rank* problem [7]. Such a data structure allows one to maintain a subset $A \subseteq \{1, \ldots, n\}$ under the following operations: *Insert*(i, Ψ), which inserts i into Ψ, *Delete*(i, Ψ), which deletes i from Ψ, and *Rank*(i, Ψ), which, given some $i \in A$, returns the number of elements in A that are greater than i.[2] It is known that each of these operations can be performed in $O(\log n / \log \log n)$ time [7].

[2] We note that the *Rank*(i, Ψ) operation is customarily defined as returning the number of elements in A that are less than or equal to i. However, it is easy to see that this version and the one we need are computationally equivalent.

Given any $x \in D$, let $f(x)$ denote the largest element in the bucket of x in σ_1. We write $\Gamma(x)$ to denote the set $\{z \in \bigcup_{1 \leq p < q} C_p \colon z > f(x), \text{ and } q = \sigma_2(x)\}$. The main idea behind our algorithm is captured in the following lemma.

Lemma 4.1 *For any two elements $x, y \in D$, $\sigma_1(x) < \sigma_1(y)$, and $\sigma_2(y) < \sigma_2(x)$ if and only if $y \in \Gamma(x)$.*

Proof: ($\Rightarrow$) By definition, if $y \in \Gamma(x)$ then $\sigma_2(y) < \sigma_2(x)$. Similarly, since the elements in σ_1 are in sorted order, $y > f(x)$ implies that $\sigma_1(x) < \sigma_1(y)$.

($\Leftarrow$) Since the elements of σ_1 are in sorted order, we must have $y > f(x)$. Similarly, since $\sigma_2(y) < \sigma_2(x)$, y must be an element of $\bigcup_{1 \leq p < q} C_p$ where $q = \sigma_2(x)$. Thus, y must be in $\Gamma(x)$. ■

Lemma 4.2 $|\mathcal{D}(\sigma_1, \sigma_2)| = \sum_{x \in D} |\Gamma(x)|$.

Proof: Consider any pair $\{x, y\} \in |\mathcal{D}(\sigma_1, \sigma_2)|$. Since $\{x, y\}$ is an unordered pair, we may assume, without any loss of generality, that $\sigma_2(y) < \sigma_2(x)$. Thus, by Lemma 4.1, this pair $\{x, y\}$ is counted in the sum $\sum_{x \in D} |\Gamma(x)|$. Additionally, since $\{x, y\}$ is counted exactly once, we must have $|\mathcal{D}(\sigma_1, \sigma_2)| \leq \sum_{x \in D} |\Gamma(x)|$. Similarly, by Lemma 4.1, each term counted in the expression $\sum_{x \in D} |\Gamma(x)|$ must be an element of $\mathcal{D}(\sigma_1, \sigma_2)$. Moreover, since each term in the expression $\sum_{x \in D} |\Gamma(x)|$ represents a distinct element of $\mathcal{D}(\sigma_1, \sigma_2)$, we must have $|\mathcal{D}(\sigma_1, \sigma_2)| \geq \sum_{x \in D} |\Gamma(x)|$. The lemma follows. ■

By Lemma 4.2, to compute $|\mathcal{D}(\sigma_1, \sigma_2)|$ our algorithm computes $|\Gamma(x)|$ for each $x \in D$. We use the subset rank data structure to compute $|\Gamma(x)|$ efficiently for each $x \in D$. In particular, given $x \in D$, if Ψ consists of the elements in $\bigcup_{1 \leq p < q} C_p$, where $q = \sigma_2(x)$, then we must have *Rank*$(f(x), \Psi) = |\Gamma(x)|$. Thus, the algorithm effectively reduces to maintaining Ψ by incrementally inserting elements into it and making appropriate *Rank* queries. A detailed description appears below.

Algorithm *OppositePairs*

```
 1: for $i$ from 1 to $n$ do
 2:     Let $f(i)$ be the largest element in $i$'s bucket in $\sigma_1$.
 3: Initialize a counter count $\leftarrow 0$.
 4: Create an empty data structure $\Psi$ for the subset rank problem.
 5: Insert all the elements of $C_1$ to $\Psi$.
 6: for $i$ from 2 to $c$ do
 7:     for each element $j \in C_i$ do
 8:         count $\leftarrow$ count $+$ Rank$(f(j), \Psi)$.
 9:     Add all the elements of $C_i$ to $\Psi$.
10: return count.
```

Remark: Strictly speaking, the subset rank problem requires the element being ranked to be actually present in the subset. However, this issue is easy to deal with by inserting (if necessary) and subsequently deleting the elements being ranked.

Theorem 4.1 $|\mathcal{D}(\sigma_1, \sigma_2)|$ *can be computed in* $O(n \log n / \log\log n)$ *time.*

Proof: We study the correctness and time complexity of Algorithm *OppositePairs*.

Correctness: Observe that if Ψ consists of the elements in $\bigcup_{1 \le p < q} C_p$ where $q = \sigma_2(x)$, then *Rank*$(f(x), \Psi) = |\Gamma(x)|$. Lemma 4.2 now immediately establishes the correctness of Algorithm *OppositePairs*.

Complexity: The first step in the algorithm is to compute the values $f(i)$, for each $1 \le i \le n$, and these are easily pre-computed in $O(n)$ time. Subsequently, the complexity of the algorithm is dominated by the two nested *for* loops in Steps 6 and 7. In particular, each iteration of Step 8 takes $O(\log n / \log\log n)$ time, and the total number of iterations is $O(n)$, yielding a total complexity of $O(n \log n / \log\log n)$ for these steps. Similarly, Step 9 requires a total of $O(n \log n / \log\log n)$ time as well. Thus, the time complexity of Algorithm *OppositePairs* is $O(n \log n / \log\log n)$. ■

4.2 An $O(n \log \min\{b, c\})$ algorithm

The algorithm we now present is motivated by the fact that in most real-life situations the number of buckets in at least one of the rankings is either quite small or fixed. In such cases, our $O(n \log \min\{b, c\})$ algorithm has linear or near-linear time complexity. This algorithm uses a divide-and-conquer strategy similar to that of the classical $O(n \log n)$ merge sort based algorithm for inversion counting (see, for example, [8]).

Our algorithm treats σ_1 and σ_2 as fully ranked lists. Thus, σ_1 simply becomes the sorted list $1, \ldots, n$. We then run a merge sort like algorithm on the list σ_2, which not only sorts the list but also computes the value $|\mathcal{D}(\sigma_1, \sigma_2)|$ by maintaining a counter and carefully incrementing it during the merging process. The typical "merge" procedure in merge sort thus becomes a "merge-and-count" procedure in our algorithm. A key difference in our algorithm is that our divide-and-conquer strategy works at the level of buckets, i.e. it does not break up buckets in σ_2 into smaller parts. Thus, there are only $O(\log c)$ levels of recursion. This is because for each $i \in \{1, \ldots, c\}$, the set C_i is already stored as a list in ascending order, and moreover, the pairs of elements we wish to count must appear in different buckets in σ_2. Thus, we gain nothing by breaking up the buckets into smaller parts.

The input for procedure *MergeAndCount* consists of two mutually disjoint lists $P = \langle P_1, \ldots, P_{|P|} \rangle$ and $Q = \langle Q_1, \ldots, Q_{|Q|} \rangle$ containing elements from the domain D, such that for any $p \in P$ and $q \in Q$, $\sigma_2(p) < \sigma_2(q)$. We will also assume that each element in P and Q is tagged with the bucket number it belongs to in σ_2 as well as in σ_1. As in merge sort, the input lists P, Q are both in sorted order, and the output is a sorted list L obtained by merging P and Q. Additionally, the *MergeAndCount* procedure also counts the number of unordered pairs (p, q), where $p \in P$ and $q \in Q$, such that $\sigma_1(q) < \sigma_1(p)$. This is done as follows: As in merge sort, assume that in the current step we are at element P_i in P and Q_j in Q. If $P_i < Q_j$ then we simply append P_i to L and continue. However, if $Q_j < P_i$, then in addition to appending Q_j to L, we increment *count* (which is a counter that is initialized to 0 at the beginning of the algorithm) by the number of elements in the list $P_i, \dots, P_{|P|}$ that are not in the same bucket as Q_j in σ_1. When procedure *MergeAndCount* terminates, it outputs the value of *count* along with the list L.

Lemma 4.3 *Procedure MergeAndCount on parameters (P, Q) computes the number of unordered pairs $\{p, q\}$, where $p \in P$ and $q \in Q$, such that $\sigma_1(q) < \sigma_1(p)$ and $\sigma_2(p) < \sigma_2(q)$.*

Proof: Let Π be the set of unordered pairs (p, q), where $p \in P$ and $q \in Q$, such that $\sigma_1(q) < \sigma_1(p)$ and $\sigma_2(p) < \sigma_2(q)$. During the merging process we increment *count* by the number of elements in the list $P_i, \dots, P_{|P|}$ that are not in the same bucket as Q_j in σ_1. Observe that (i) each element in P belongs to a different bucket than any element of Q in σ_2, (ii) we ensure that we count the pair $\{P_x, Q_j\}$, for $i \leq x \leq |P|$, only if P_x and Q_j are in different buckets in σ_1, and (iii) since both P and Q are sorted, element Q_j must be smaller than each of the elements $P_i, \dots, P_{|P|}$. In addition, since element Q_j is considered at most once during the merging, each unordered pair is counted only once. Therefore, when procedure *MergeAndCount* terminates, *count* $\leq |\Pi|$.

Now consider some pair $\{u, v\} \in \Pi$ where $u < v$. Then, by definition, $\sigma_1(u) < \sigma_1(v)$, and $\sigma_2(v) < \sigma_2(u)$. This implies that $v \in P$ and $u \in Q$. Based on procedure *MergeAndCount*, it immediately follows that the pair $\{u, v\}$ will be counted. Therefore, when the procedure terminates, *count* $\geq |\Pi|$. Thus, we have *count* $= |\Pi|$. ■

Lemma 4.4 *After an $O(n)$ preprocessing step, procedure MergeAndCount on parameters (P, Q) can be executed in $O(|P| + |Q|)$ time.*

Proof: In the preprocessing step, let us create an array A of size b, initialized to all 0's. This takes $O(n)$ time. Consider the merging step when Q_j is appended to L in procedure *MergeAndCount*. It is not hard to see that if we knew exactly how many elements in $P_i, \dots, P_{|P|}$ are in the same bucket as Q_j in σ_1, then this could be executed in $O(1)$ time. This number can be computed as follows: During each execution of procedure *MergeAndCount*, we first traverse through the list P and update array A so that A_i is the number of elements in P that belong to bucket i in σ_1. This takes $O(|P|)$ time. Later, during a merging step, if the bucket of P_i in σ_1 is p, we decrement the value at A_p by one. Now, it is easy to see that if the bucket of Q_j in σ_1 is q, then A_q equals the number of elements from $P_i, \dots, P_{|P|}$ that are in the same bucket as Q_j in σ_1. Finally, re-initializing A to 0 for subsequent reuse takes $O(|P|)$ time. The lemma follows. ■

Theorem 4.2 *The value* $|\mathcal{D}(\sigma_1, \sigma_2)|$ *can be computed in* $O(n \log \min\{b, c\})$ *time.*

Proof: Recall that our divide and conquer algorithm has $O(\log c)$ levels of recursion. Therefore, Lemmas 4.3 and 4.4 imply that $|\mathcal{D}(\sigma_1, \sigma_2)|$ can be computed in $O(n \log c)$ time. Since the algorithm also works if we switch σ_1 and σ_2, we get a total time complexity of $O(n \log \min\{b, c\})$. ■

References

[1] D. E. Critchlow, Metric Methods for Analyzing Partially Ranked Data, Vol. 34 of Lecture Notes in Statist., Springer-Verlag, Berlin, 1980.

[2] H. P. Young, Condorcet's theory of voting, American Political Science Review 82 (2) (1988) 1231–1244.

[3] C. Dwork, R. Kumar, M. Naor, D. Sivakumar, Rank aggregation methods for the web, in: Tenth International World Wide Web Conference, Hong Kong, 2001, pp. 613–622.

[4] R. Fagin, R. Kumar, D. Sivakumar, Comparing top k lists, SIAM J. Discrete Math. (2003) 134–160.

[5] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, E. Vee, Comparing partial rankings, SIAM J. Discrete Math. 20 (3) (2006) 628–648.

[6] N. Ailon, Aggregation of partial rankings, p-ratings and top-m lists, in: SODA '07, 2007, pp. 415–424.

[7] P. F. Dietz, Optimal algorithms for list indexing and subset rank, in: WADS, 1989, pp. 39–46.

[8] J. Kleinberg, E. Tardos, Algorithm Design, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.