Computing Similarity Distances Between Rankings

Lili Su · Farzad Hassanzadeh · Olgica Milenkovic

Received: date / Accepted: date

Abstract We address the problem of computing distances between rankings that take into account similarities between candidates. The need for evaluating such distances is governed by applications as diverse as rank aggregation, bioinformatics, social sciences and data storage. The problem may be summarized as follows. Given two rankings and a positive cost function on transpositions that depends on the similarity of the candidates involved, find a smallest cost sequence of transpositions that converts one ranking into another. Our focus is on costs that may be described via special metric-tree structures and on full rankings modeled as permutations. The presented results include a quadratic-time algorithm for finding a minimum cost transform for simple cycles; and a linear time, $5/3$-approximation algorithm for permutations that contain multiple cycles. In addition, for permutations with digraphs represented by non-intersecting cycles embedded in trees, we present a polynomial-time transform algorithm. The proposed methods rely on investigating a newly introduced balancing property of cycles embedded in trees, cycle-merging methods, and shortest-path optimization techniques.

Keywords Digraphs of permutations · rankings · shortest paths · transposition distance

1 Introduction

Meta-search engines, recommenders, social data aggregation centers as well as many other data processing systems are centered around the task of ranking distinguishable objects according to some predefined criteria [2,18,19]. Rankings are frequently provided by different experts or generated according to different criteria. In order to perform comparative studies of such rankings or in order to aggregate them, one needs to be able to assess how much they agree or disagree. This is most easily accomplished by assuming that data is of the form of full rankings - i.e., of the form of permutations - and that one ranking may be chosen as a reference sample (identity). In this case, the problem of evaluating the agreement between permutations essentially reduces to the problem of sorting permutations.

The problem of sorting distinct elements according to a given criterion has a long history and was studied in mathematics, computer science, and social choice theory alike [10,11,14]. One volume of the classical text in computer science – Knuth’s The Art of Computer Programming – is almost entirely devoted to the study of sorting permutations. The solution to the problem is straightforward and well known if the sorting steps are...
swaps (transpositions) of two elements: One has to first perform a cycle decomposition of the permutation and then swap elements in the same cycle until all cycles have unit length.

Sorting problems naturally introduce the need for studying distances between permutations. There are many different forms of distance functions on permutations, with the two most frequently used being the Cayley distance/Kendall distance [5]. In this case, each transposition/adjacent transposition contributes equally to the overall distance. Although many generalizations of Kendall and other distances are known [13], until our recent companion work [9], no attempt was made to study the problem in a more general framework where one may assign to every basic rearrangement step a cost (weight) that may be proportional to its likelihood of being performed. Examples where such cost-constrained problems arise are social sciences [9] (in the context of constrained vote aggregation), bioinformatics [3,21] (in the context of the fragile DNA breakage models or of gene prioritization), and logistics [12].

Most practical problems call for positive costs (weights) on transpositions, and costs that capture some constraint imposed by the comparative study performed on the permutations. The problem at hand may then be described as follows: For a given set of positive costs assigned to transpositions of distinct elements, find a smallest cost sequence of transpositions (henceforth termed transform) converting a given permutation to the identity.

In our subsequent analysis, we focus on constraints that take into account that candidates may be similar and that transposing similar candidates induces a smaller cost than transposing dissimilar candidates. We refer to the underlying family of distance measures as similarity distances. The notion of similarity distance is not to be confused with the metric used in [20], where the goal was to rank similar items close to each other in the aggregate.

To illustrate the practical utility of the similarity distance, we next present a number of illustrative examples.

The first example comes from social choice theory. When ranking politicians and assessing voters’ opinion dynamics, one often needs to take into account that the candidates come from different parties. Swapping candidates from the same party may be perceived as having a smaller impact on the overall diversity of the ranking or outcome of an election than doing otherwise. As an example, consider the following three rankings of politicians:

$$\pi_1 = (\text{Clinton, Obama, Bush, Kerry, Romney}) = (D, D, R, D, R),$$
$$\pi_2 = (\text{Obama, Clinton, Bush, Kerry, Romney}) = (D, D, R, D, R),$$
$$\pi_3 = (\text{Clinton, Bush, Obama, Kerry, Romney}) = (D, R, D, D, R).$$

Notice that $\pi_2$ and $\pi_3$ differ from $\pi_1$ only in one (adjacent) transposition. In the first case, the swap involves members of the same party, while in the second case, the transposed candidates belong to two different parties.

Throughout the paper, we use the words cost and weight interchangeably, depending on the context of the exposition.
parties. It would hence be reasonable to assume that the distance between \(\pi_1\) and \(\pi_2\) be smaller than the distance between \(\pi_1\) and \(\pi_3\) because it induces a change in the overall ordering of the parties.

To capture this similarity, candidates may be arranged into a tree-structure with each edge having a certain weight, so that the transposition cost of two candidates equals the weight of the unique path between them. An illustrative example involving three parties is shown in Fig. 1 where the tree has only one vertex of degree larger than two, corresponding to the political center. Republicans, Democrats and Greens are all arranged on different branches of the tree, and in order of their proximity to the political center. Note that two republicans are generally closer in the tree compared to a republican and a democratic candidate, implying that transpositions involving Republicans are, on average, less costly than those involving candidates of two different parties.

Another application of metric-tree weight distances is in assignment aggregation and rank aggregation [2, 6, 7, 8, 15, 17]. In the former case, one considers a committee of \(m\) members with the task of distributing \(n\) jobs to \(n\) candidates. Each committee member provides her suggestion of full assignment of candidates to jobs. The goal is to aggregate the assignments given by individual committee members into one assignment. If a measure of similarity between the candidates is available, one can use the similarity distance to aggregate the assignments by finding the best compromise in terms of swapping candidates of similar qualifications, age, gender, working hour preferences, etc. This is achieved by computing the median of the rankings under a suitable similarity distance, such as the metric-path cost [9].

The third application, and the one that has received most attention in the areas of computer science and search engines is related to overcoming biases of search engines [2, 6, 23]. As an example, when trying to identify the links most closely associated with a query, many different search engines can be utilized, including Google, Yahoo!, Ask, Bing, IBM Sangan, etc. One may argue that the most objective, and hence least biased, rankings are produced by aggregating the rankings of these different search engines. Many search queries are performed with the goal of identifying as many diverse possibilities on the first or first two listed pages. Hence, such problem also motivates the need for identifying similarity distances on trees, as many search items may be naturally arranged in such a structure. Simulation results suggesting similarity distances may lead to more diverse solutions can be found in our companion paper [9].

Finally, similarity distances may be used as valuable tools in gene prioritization studies. Gene prioritization is a method for identifying disease-related genes based on diverse information provided by linkage studies, sequence structure, gene ontology and other procedures [1]. Since testing candidate genes is experimentally costly, one is often required to prioritize the list by arranging the genes in descending order of likelihood for being involved in the disease. Different prioritization methods produce different lists, and similarity of the lists carries information about which genes may be important under different selection criteria. In addition, since genes are usually clustered into family trees according to some notion of similarity, finding lists that prioritize genes while at the same time making sure that all families of genes are tested is of great interest.

The contributions of this work are three-fold. First, we introduce Y-tree cost functions and the notion of similarity distance between permutations. In this setting, the cost of transposing two elements equals the weight of the shortest path in a Y-tree, i.e. a tree with at most one node of degree three. Second, we describe an exact quadratic-time decomposition algorithm for cycle permutations with Y-tree costs. Third, we develop a quadratic-time, constant-approximation method for computing the similarity distance between arbitrary permutations. In addition, for permutations whose functional digraphs consist of cycles that may be embedded in the Y-tree in a planar fashion, we describe an exact algorithm for computing the similarity distance. Note that in the above context, the term “embedding” differs from the standard notion of graph embedding and is related to the work in [4].

The paper is organized as follows. Section 2 introduces the notation and definitions used throughout the paper. Section 3 contains a brief review of prior work as well as some relevant results used in subsequent derivations. This section also presents a quadratic-time algorithm for computing the Y-tree similarity distance between cycle permutations. This algorithm is extended in Section 4 to general permutations via cycle-merging strategies that provide polynomial-time, constant-approximation guarantees. Section 5 contains our concluding remarks.

Clearly, one could also argue that changes at the top of the list are more relevant than changes at the bottom, in which case the comment about the pairwise distances should be reversed. An overview of positional distances may be found in [9], and the related work on generalized Borda counts in [19, 22] and references therein.
2 Mathematical Preliminaries

A permutation \( \pi : [n] \to [n] \) is a bijection, where we use the notation \([n] \triangleq \{1, 2, \cdots, n\}\). The collection of all permutations on \([n]\) – the symmetric group of order \(n!\) – is denoted by \(S_n\). There are several ways to represent a permutation. The two line representation has the domain written out in the first line and the corresponding image in the second line. For example, the following permutation is given in the two line form

\[
\pi = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
6 & 1 & 2 & 5 & 4 & 3 
\end{pmatrix}.
\]

The one-line representation is more succinct, and basically equals to the second line of the two-line representation; the above permutation may also be written as \((6, 1, 2, 5, 4, 3)\). Sometimes we find it useful to describe a permutation in terms of elements and their images; in this case, a third description of the aforementioned permutation is \(\pi(1) = 6, \pi(2) = 1, \pi(3) = 2, \pi(4) = 5, \pi(5) = 4, \text{ and } \pi(6) = 3\). With this representation at hand, the product of two permutations \(\pi, \sigma \in S_n, \mu = \pi \sigma\), can be defined by \(\mu(i) = \pi(\sigma(i))\), for all \(i \in [n]\).

For \(k > 1\), a \(k\)-cycle, denoted by \(c = (i_1 \cdots i_k)\), is a permutation that acts on \([n]\) in the following way:

\[
i_1 \to i_2 \to \cdots \to i_k \to i_1,
\]

where \(x \to y\) is used to denote \(y = c(x)\). In other words, the permutation cyclically shifts all entries in \(\{i_1, \cdots, i_k\}\) and keeps all other elements fixed. The set \(\{i_1, \cdots, i_k\}\) is defined as the support of the cycle \(c\), and is denoted by \(\text{supp}(c)\). For example, the support of the cycle \((1632)\) is \([1, 2, 3, 6]\). We refer to a cycle of length two as a transposition and denote it by \((ab)\). An adjacent transposition is the transposition of two adjacent elements. The symbol \(e\) is reserved for the identity permutation \((1, 2, \cdots, n)\).

In general, for \(a, b \in [n]\), \(\pi(ab) \neq (ab)\pi; \pi(ab)\) corresponds to swapping elements of \(\pi\) in positions \(a\) and \(b\) while \((ab)\pi\) corresponds to swapping elements \(a\) and \(b\) in \(\pi\). For instance, \((6, 1, 2, 5, 4, 3)(23) = (6, 2, 1, 5, 4, 3)\), while \((23)(6, 1, 2, 5, 4, 3) = (6, 1, 3, 5, 4, 2)\). Note that in the former example, we used \(\pi(ab)\) to denote the product of a permutation and a transposition, which is not to be confused with the image of an element under \(\pi\).

Two cycles are said to be disjoint if the intersection of their supports is empty; on the other hand, adjacent cycles have only one common element in their supports. It is well known that a permutation can be written as a product of disjoint cycles, which is often referred to as cycle decomposition or cycle representation. For example, the cycle decomposition of the permutation \((6, 1, 2, 5, 4, 3)\) equals \((1632)(45)\), where one can freely choose the order in which to multiply \((1632)\) and \((45)\). We notice that a cycle can be decomposed into a product of shorter cycles, representing a combination of disjoint and adjacent cycles. This procedure is termed adjacent cycle decomposition. The distinction between the aforementioned two cycle decompositions is that in adjacent cycle decompositions, the order of multiplication matters; \((1632)\) equals \((216)(36)\) but not \((36)(216)\). Contrary to the uniqueness of the disjoint cycle decomposition, there are potentially multiple adjacent cycle decompositions.

The functional digraph of a function \(f : [n] \to [n]\), denoted by \(\mathcal{G}(f)\), is a directed graph with vertex set \([n]\) and an edge from \(i\) to \(f(i)\) for each \(i \in [n]\). Specifically, for a permutation \(\pi, \mathcal{G}(\pi)\) is a collection of disjoint cycles; hence, the cycles of a permutation correspond to the cycles of its functional digraph.

A transposition weight function \(\varphi\) is a function that assigns to each transposition \(\tau\) a positive weight \(\varphi_{\tau}\). Let \(G_{\varphi} = (V, E)\), with \(V = [n]\) be a connected, undirected, edge-weighted graph, where all weights are positive. Assume that \(\varphi_{ab} = \varphi_{\tau}\) equals the minimum weight among all paths between vertices \(a\) and \(b\), for all \(a, b \in [n]\). Then \(\varphi\) is a graph metric, and we refer to \(G_{\varphi} = (V, E)\) as the defining graph of the weight function \(\varphi\). An arbitrary chosen minimum weight path between vertices \(a\) and \(b\) will be consequently denoted by \(p_{\varphi}(a, b)\).

The weight of a sequence of transpositions is defined as the sum of the weights of its constituent elements. That is, the weight of the sequence of transpositions \(T = (\tau_1, \cdots, \tau_T)\) equals

\[
\text{wt}(T) = \sum_{i=1}^{T} \varphi_{\tau_i},
\]

\footnote{This is not to be confused with the one line representation where we use commas between entries.}
Computing Similarity Distances Between Rankings

1) Defining graph $G_\phi$ corresponding to a general metric-tree weight function $\phi$. From the graph, one reads $\phi(17) = \phi(13) + \phi(34) + \phi(47) = 2 + 3 + 4 = 9$.

2) Defining graph $G_\phi$ of a Y-tree with all, except for one, edge costs equal to one. From the graph, one reads $\phi(17) = \phi(18) + \phi(36) + \phi(67) = 1 + 1 + 1 = 3$.

Fig. 2: Examples of defining graphs.

where $|T|$ denotes the number of transpositions in the sequence $T$.

If $\sigma = \pi \tau_1 \tau_2 \cdots \tau_{|T|}$, we refer to $T = (\tau_1, \cdots, \tau_{|T|})$ as a transform, converting $\pi$ into $\sigma$. The set of all such transforms is denoted by $A(\pi, \sigma)$. Clearly, $A(\pi, \sigma)$ is non-empty for any $\pi, \sigma \in S_n$. The Cayley and Kendall distances are defined to be the lengths of the shortest transform including arbitrary and adjacent transpositions only, respectively.

The $\phi$-weighted transposition distance between $\pi$ and $\sigma$ is defined by

$$d_\phi(\pi, \sigma) = \min_{T \in A(\pi, \sigma)} \text{wt}(T).$$

Computing $d_\phi(\pi, \sigma)$ represents a minimization problem over $A(\pi, \sigma)$, namely that of finding a minimizing transform $T^\ast$ such that $d_\phi(\pi, \sigma) = \text{wt}(T^\ast)$. We henceforth focus on the previously introduced family of graph metric weights, satisfying the triangle inequality

$$\phi(ab) \leq \phi(ac) + \phi(cb), \quad \text{for all } a, b, c \in [n].$$

In particular, a weight function $\phi$ is a metric-tree weight function if it has a tree-structured defining graph. For such defining graphs, there clearly exists a unique minimum cost path between any two vertices, and for $a, b \in [n]$, $\phi(ab)$ is the sum of the weights of the edges on the unique path between $a$ and $b$ in $G_\phi$. If $G_\phi$ is a path, then $\phi$ is called a metric-path weight function. Furthermore, if there exists a unique vertex in a tree structured $G_\phi$ of degree larger than or equal to three, the graph is called a star metric-tree. The vertex with highest degree is referred to as the central vertex. In particular, if the central vertex has degree three, the defining graph is called a Y-tree. Examples of the aforementioned defining graphs are shown in Fig. 2.

The following function, termed the displacement, is of crucial importance in our analysis of $d_\phi$:

$$D_\phi(\pi, \sigma) = \sum_{i=1}^n \text{wt}(p_\phi^*(\pi^{-1}(i), \sigma^{-1}(i)));$$

Similarly, we refer to $\text{wt}(p_\phi^*(\pi^{-1}(i), \sigma^{-1}(i)))$ as the displacement of the element $i$ in the permutations $(\pi, \sigma)$. All closed form expressions for $d_\phi$ derived in subsequent sections, as well as their approximations, will rely on the displacement function $D_\phi$.

It is easy to verify that for every positive weight function, the weighted transposition distance $d_\phi$ and $D_\phi$ are both pseudo-metrics and left-invariant (i.e., $d_\phi(\pi, \sigma) = d_\phi(\omega \pi, \omega \sigma)$ and $D_\phi(\pi, \sigma) = D_\phi(\omega \pi, \omega \sigma)$, for all $\pi, \sigma, \omega \in S_n$). As a result, we henceforth focus on analyzing $d_\phi(\pi, e)$ and $D_\phi(\pi, e)$, where before, $e$ denotes the identity permutation. We refer to the problem as the (weighted) decomposition problem. The main part of the paper is devoted to studying the decomposition problem when $\pi$ is a cycle, given that a logical manner to decompose a permutation with multiple cycles is via individual and/or joint decomposition of cycles.

For ease of exposition, we draw the digraph of a permutation and the undirected defining Y-tree graph of the given weight function on the same vertex set, as shown in Fig. 3. In this case, we say that the permutation...
Fig. 3: Defining Y-tree and the cycle (1 2 5 8 7). Thin lines are used to represent the defining Y-tree $\mathcal{G}$, while directed bold lines are used to represent the permutation.

is embedded in the defining graph. This graphical way of viewing both the cost function and the cycle decomposition of a permutation allows us to gain intuition about the algorithms involved in the decomposition approach.

Denote the branches of a star metric-tree, which are paths starting from the central vertex and extending to a leaf, excluding the central vertex, as $B_1, \ldots, B_{n^*}$, $n^* \leq n$. First, we formalize the notion of a cycle path on the Y-tree as a cycle that has support contained in $B_i \cup B_j$, for some $i, j$ not necessarily distinct. In other words, a cycle lies on a path if its support is contained in at most tow of the three branches. Furthermore, for a branch pair $(B_i, B_j)$, $i \neq j$, let $l_{ij}$ be the number of directed edges from $a_i \in B_i$ to $a_j \in B_j$; similarly, let $l_{ji}$ be the number of directed edge from $a_j \in B_j$ to $a_i \in B_i$. For a cycle permutation, if $l_{ij} = l_{ji}$, we say that the branch pair $(B_i, B_j)$ is balanced. Furthermore, if $l_{ij} = l_{ji}$ for all $i, j \in \{1, \ldots, n^*\}$, we say that the cycle is balanced.

A transposition $(ab)$ is efficient with respect to the permutation $\pi$ if
\[ D(\pi, e) - D(\pi(ab), e) = 2\varphi(ab). \]

The inefficiency of a transposition $(ab)$ with respect to the permutation $\pi$, denoted by $\sigma_{(ab)}$, equals
\[ \sigma_{(ab)} = 2\varphi(ab) - (D(\pi, e) - D(\pi(ab), e)). \]

The proposed algorithm for finding a minimum cost decomposition of a permutation under Y-tree weights or star weights consists of two steps:
1. First, we derive a closed form expression for the minimum cost decomposition of a single cycle. The presented algorithm is exact and it can find a minimizing transform $T^*$ in time $O(n^2)$.
2. Second, for general permutations with multiple cycles, we develop a linear time, $5/3$-approximation algorithm that merges cycles into one single cycle and then uses the first step. For permutations which may be embedded on the defining graphs as nonintersecting cycles, we provide an exact optimal merging algorithm of complexity $O(n^2)$.

3 Similarity Distances on Y-trees: Single Cycle

The analysis of decomposition algorithms, as already pointed out, relies on using the displacement functions and properties of the cycle embedding in the tree, such as the balancing property. We hence start our analysis by providing useful characterizations of efficient transpositions. To accomplish this task, we notice that for any cycle embedded in a Y-tree, two vertices $a, b$ may assume only one of the four possible configurations shown in Fig. 3. Furthermore, since the defining graph $G_\varphi$ is a tree, there is a unique path between any two vertices $a, b$ of $G_\varphi$, denoted as $a$-$b$. The next lemma shows that only in case of the vertices $a$ and $b$ assuming configuration 1), the transposition $(ab)$ is efficient.

**Lemma 1** Let $G_\varphi$ be the defining graph of a metric-tree weight function and let $\pi$ be a permutation, so that $a, b \in \text{supp}(\pi)$. Then transposition $(ab)$ is efficient if and only if vertex $a$ lies on the $b$-$\pi(b)$ path in $G_\varphi$ and vertex $b$ lies on the $a$-$\pi(a)$ path in $G_\varphi$.

*It is easy to see that the inefficiency of a transposition is nonnegative.*
Fig. 4: Four types of local structures used to assess the efficiency of a transposition, where \( a' = \pi(a) \), and \( b' = \pi(b) \).

Proof Sufficient Condition. There exists a unique \( a' - \pi(a) \) path, and thus
\[
\text{wt}(p^{\pi^{-1}}(a), e(a)) = \text{wt}(p^\pi(a, \pi(a))) \\
= \text{wt}(p^\pi(a, b)) + \text{wt}(p^\pi(b, \pi(a))) \\
= \phi_{(ab)} + \text{wt}(p^\pi(b, \pi(b))) 
\]  
(1)
where the first equality follows from the left-invariance property of the transposition distance, the second equality holds due to the fact that \( b \) lies on the \( a' - \pi(a) \) path and the third equality is verified by the definition of metric-tree weight functions. A similar expression holds for \( b' \). Upon applying the transposition \( (ab) \), the displacement of \( a \) decreases by \( \phi_{(ab)} \), while the displacement of \( b \) decreases by \( \phi_{(ab)} \). As the transposition \( (ab) \) can only affect the displacement of vertices \( a \) and \( b \), its total reduction of displacement equals \( 2\phi_{(ab)} \). This completes the proof of the sufficient condition.

Necessary Condition. Without loss of generality, suppose that \( (ab) \) is efficient but \( a \) does not lie on the \( b' - \pi(b) \) path. Then, the net displacement of \( b \) reduced by \( (ab) \) is strictly less than \( \phi_{(ab)} \), implying that the total reduction in displacement is strictly less than \( 2\phi_{(ab)} \). This contradicts the assumption that \( (ab) \) is efficient.

In general, the following Lemma holds.

Lemma 2 Let \( \phi \) be a metric-tree weight function, and let \( \pi \) be a permutation. The distance between \( \pi \) and \( e \) is bounded below by one half of the total displacement, i.e.,
\[
d_\phi(\pi, e) \geq \frac{1}{2}D_\phi(\pi, e). 
\]

The lower bound is achieved for metric-path weight functions \( \phi \), i.e., weights for which the defining graph is a path,
\[
d_\phi(\pi, e) = \frac{1}{2}D_\phi(\pi, e). 
\]
The proof of the previous equality pertaining to metric-paths can be found in our companion paper [9], and the result follows by induction on the number of elements in the support of \( \pi \). An algorithm which describes how to find a minimum cost transform \( T^* \) under the given scenario can be easily derived using the idea behind the proof, and is presented in what follows.

Without loss of generality, label the vertices in the defining path from left to right as \( 1, 2, \cdots, n \). Suppose the given cycle is \( \pi = (a_1 a_2 \cdots a_{|\text{supp}(\pi)|}) \), where \( a_1 = \min \text{supp}(\pi) \). If this is not the case, we just rewrite
\begin{algorithm}
\textbf{Algorithm 1: Path Case Decomposition}

\begin{algorithmic}
\State \textbf{Input:} A cycle $\pi = (a_1 a_2 \ldots a_{\text{supp}(\pi)})$
\State \textbf{Output:} A minimum cost decomposition $\pi = t_1 t_2 \ldots t_k$
\While{$|\text{supp}(\pi)| > 2$}
\If{$a_i \neq a_{i+|\text{supp}(\pi)|}$}
\State $\pi = \pi \tau$, where $\pi_1 = (a_1 a_2 \ldots a_i)$ and $\pi_2 = (a_{i+1} a_{i+2} \ldots a_{|\text{supp}(\pi)|})$;
\State repeat procedure for $\pi_1$ and $\pi_2$;
\Else
\State $\pi = \pi \tau$, where $\tau = (a_{|\text{supp}(\pi)|} a_{|\text{supp}(\pi)|-1} \ldots a_1)$;
\State repeat procedure for $\pi$.
\EndIf
\EndWhile
\end{algorithmic}

\end{algorithm}

\section{Algorithm 1: Path Case Decomposition}

\begin{algorithm}
\textbf{Algorithm 1: Path Case Decomposition}

\begin{algorithmic}
\State \textbf{Input:} A cycle $\pi = (a_1 a_2 \ldots a_{\text{supp}(\pi)})$
\State \textbf{Output:} A minimum cost decomposition $\pi = t_1 t_2 \ldots t_k$
\While{$|\text{supp}(\pi)| > 2$}
\If{$a_i \neq a_{i+|\text{supp}(\pi)|}$}
\State $\pi = \pi \tau$, where $\pi_1 = (a_1 a_2 \ldots a_i)$ and $\pi_2 = (a_{i+1} a_{i+2} \ldots a_{|\text{supp}(\pi)|})$;
\State repeat procedure for $\pi_1$ and $\pi_2$;
\Else
\State $\pi = \pi \tau$, where $\tau = (a_{|\text{supp}(\pi)|} a_{|\text{supp}(\pi)|-1} \ldots a_1)$;
\State repeat procedure for $\pi$.
\EndIf
\EndWhile
\end{algorithmic}

\end{algorithm}

\section{3.1 Case 1: The support of the cycle contains the central vertex}

Denote the central vertex as $a_0$, and use $B_1, B_2$ and $B_3$ to denote both the three branches of our Y metric-tree and their corresponding vertex sets.

The decomposition procedure is described in Algorithm 2. We use $k$ to track the index of subcycles in the decomposition generated at each iteration of the procedure. The indicator functions $\mathcal{I}$ are applied to conditions of the form $\text{supp}(\pi) \cap B_i \neq \emptyset$, where $i \in \{1, 2, 3\}$. Note that $a_1$ may be used to index different vertices from iteration to iteration. The algorithm terminates when all subcycles $\pi_j$ have supports that lie on a path of the defining Y-tree.

\begin{lemma}
For a cycle $\pi$ containing the central vertex, the distance between $\pi$ and $e$ equals one half of their total displacement, i.e.,
\[ d(\pi, e) = \frac{1}{2} \phi(\pi, e). \]
\end{lemma}

\begin{proof}
It suffices to show that $d(\pi, e) \leq \frac{1}{2} \phi(\pi, e)$.
\end{proof}
1) We have $a_t = 2$ and $d_{\supp(\pi)} = 3$, so that $a_t \neq d_{\supp(\pi)}$. Consequently, the directed edge $(3, 1)$ is split into two parts. The resulting two cycles are represented with solid and dashed directed lines, respectively.

2) We have $a_t = a_{\supp(\pi)} = 3$. Consequently, the directed edge $(1, 4)$ is split into two parts. The resulting two cycles are represented with solid and dashed directed lines, respectively.

Fig. 5: In 1), we rewrote the cycle $(1\ 4\ 2\ 6\ 5\ 3)$ as $(1\ 4\ 2\ 6\ 5\ 3) = (1\ 4\ 2)(2\ 6\ 5\ 3)$; in 2), we used $(1\ 4\ 6\ 5\ 3) = (1\ 3)(3\ 4\ 6\ 5)$.

Fig. 6: The weight $\varphi$ is defined via a Y-tree with all edges of weight one; the cycle equals $(1\ 2\ 3)$.

Let $\pi = \pi_{k+1} \cdots \pi_1$ be the cycle decomposition of $\pi$ generated by Algorithm 2. Then,

$$d_\varphi(\pi, e) \leq \sum_{i=1}^{k+1} d_\varphi(\pi_i, e)$$

$$= \frac{1}{2} \sum_{i=1}^{k+1} D_\varphi(\pi_i, e)$$

(2)
\textbf{Algorithm 2: Adjacent Cycle Decomposition I}

\begin{verbatim}
\textbf{Input:} A cycle $\pi = (a_0a_1 \cdots a_{|\text{supp}(\pi)|-1})$
\textbf{Output:} An adjacent cycle decomposition $\pi = \pi_0 \cdots \pi_t$ with supports on $\pi_j$, $j = 1, 2 \cdots, k$, lying on paths of the Y-tree

1) Initialize: $k \leftarrow 0$;
2) while $|\mathcal{I}(\text{supp}(\pi))| \cdot |\mathcal{I}(\text{supp}(\pi))| \cdot |\mathcal{I}(\text{supp}(\pi))| = 1$ do
3) \hspace{1em} $k \leftarrow k+1$;
4) Assuming that $a_i \in B_1$, let $t = \min\{i \mid a_i \in B_1 \}$;
5) $\pi_0 \leftarrow (a_0a_1 \cdots a_t)$ and $\pi' \leftarrow (a_0a_{t+1} \cdots a_{|\text{supp}(\pi)|-1})$;
6) $\pi \leftarrow \pi'$;
7) end
8) $\pi_{k+1} \leftarrow \pi$.
\end{verbatim}

Fig. 7: 1) A Y-tree and input cycle (7 4 6 5 2 3). After the first iteration of Algorithm 2, the directed edge (4,6) is broken into two edges, decomposing the original cycle (7 4 6 5 2 3) into a product of two adjacent cycles, i.e., (7 4 6 5 2 3)=(7 4)(7 6 5 2 3), as shown in 2).

where the first bound holds due to the triangle inequality, while the second equality is satisfied since all adjacent cycles have supports on some paths in the Y-tree. From Lemma[2] we know that for metric paths, one has $d_\phi(\pi_i, e) = \frac{1}{2}D_\phi(\pi_i, e)$.

To complete the proof, we only need to show that

\[ D_\phi(\pi, e) = \sum_{i=1}^{k+1} D_\phi(\pi_i, e). \]

At each iteration of Algorithm 2, we may write

\[ \pi = (a_0a_1 \cdots a_{|\text{supp}(\pi)|-1})(a_0a_1 \cdots a_t). \]

Given that the supports of the subcycles lie on paths of the Y-tree, we may write

\[ D_\phi((a_0a_1 \cdots a_t), e) = \sum_{i=0}^{t-1} \text{wt}(p_\phi(a_1, a_{i+1})) + \text{wt}(p_\phi(a_0, a_0)), \]

and

\[ D_\phi((a_0a_1 \cdots a_{|\text{supp}(\pi)|-1}), e) = \sum_{i+1}^{|\text{supp}(\pi)|-2} \text{wt}(p_\phi(a_1, a_{i+1})) + \text{wt}(p_\phi(a_0, a_{t+1})) + \text{wt}(p_\phi(a_{|\text{supp}(\pi)|-1}, a_0)). \]

By the definition of the Y-tree weight function, there exists a unique path between $a_t$ and $a_{t+1}$. As the parameter $t$ is chosen so that $a_t \in B_1$ and $a_{t+1} \notin B_1$, we have

\[ \text{wt}(p_\phi(a_t, a_{t+1})) = \text{wt}(p_\phi(a_t, a_0)) + \text{wt}(p_\phi(a_0, a_{t+1})). \]
Combining the above equations leads to
\[ D_\phi(((a_0 a_1 \cdots a_t)_e) + D_\phi(((a_0 a_t \cdots a_{|\text{supp}(\pi)|-1})) \]
\[ = \sum_{i=0}^{t-1} \text{wt}(p_\phi^*(a_i, a_{i+1})) + \text{wt}(p_\phi^*(a_t, a_0)) + \sum_{i=t+1}^{t-1} \text{wt}(p_\phi^*(a_t, a_{i+1})) \]
\[ + \text{wt}(p_\phi^*(a_0, a_{i+1})) + \text{wt}(p_\phi^*(a_t, a_0)) \]
\[ = \sum_{i=0}^{t-1} \text{wt}(p_\phi^*(a_i, a_{i+1})) + \sum_{i=t+1}^{t-1} \text{wt}(p_\phi^*(a_t, a_{i+1})) + \text{wt}(p_\phi^*(a_t, a_{i+1})) \]
\[ + \text{wt}(p_\phi^*(a_0, a_{i+1})) + \text{wt}(p_\phi^*(a_0, a_{i+1})) \]
\[ = \sum_{i=0}^{t-1} \text{wt}(p_\phi^*(a_i, a_{i+1})) + \text{wt}(p_\phi^*(a_t, a_0)) \]
\[ = D_\phi(\pi, e) \]
Hence, inequality (2) becomes
\[ d_\phi(\pi, e) \leq \sum_{i=1}^{k+1} d_\phi((\pi_i, e) = \frac{1}{2} \sum_{i=1}^{k+1} D_\phi(\pi_i, e) = \frac{1}{2} D_\phi(\pi, e), \]
which completes the proof.

3.2 Case 2: Balanced cycles

Given that a cycle containing the central vertex is covered in Case 1 of our exposition, we henceforth tacitly assume that the balanced cycles and unbalanced cycles of interest do not contain the central vertex. The description and analysis of the algorithm proceed along similar lines as the case described in Subsection 3.1.

Note that in Algorithm 2, we start with the directed edge emanating from the central vertex and check if the decomposition conditions are met. Unfortunately, when \( a_0 \notin \text{supp}(\pi) \), it is not clear which edge to start with in order to get to a proper adjacent decomposition. Hence, we need to introduce a search procedure to identify good starting edge(s).

A push-down stack data structure is used to assist in this search. Denote the stack as \( S \), and the element on the top of the stack as \( s_0 \). We follow the closed walk induced by \( \pi \), starting from an arbitrary vertex \( \pi \) in the support until encountering an edge between branches. Such an edge is pushed into stack \( S \) and temporarily assumes the role of \( s_0 \). We keep following the closed walk while pushing edges in or out of the stack \( S \). Only edges between branches may be added to the stack. Once an edge crossing branches in the opposite direction from \( s_0 \) is encountered, the current \( s_0 \) is paired up with this edge and removed from the stack. The paired edges are then used to split the current cycle. The procedure is repeated until all the vertices of the cycle are visited exactly once.

The decomposition procedure is described in Algorithm 3. The vertices in \( B_1 \) are indexed in increasing order, starting from the vertex closest to the central vertex. A similar indexing is performed for vertices on \( B_2 \), starting from label \( |B_1| + 1 \), and for vertices on \( B_3 \) starting from label \( |B_1| + |B_2| + 1 \). Note that this form of indexing is chosen for ease of exposition, as the performance of the algorithm does not depend on the particular labeling scheme. As before, assume that the given cycle \( \pi = (a_1 a_2 \cdots a_{|\text{supp}(\pi)|}) \) is written so that \( a_1 = \min \text{supp}(\pi) \).

Note that the decomposition strategy at each iteration of Algorithm 3 depends on whether the condition \( \text{wt}(p_\phi^*(a_0, b_1)) < \text{wt}(p_\phi^*(a_0, a_{i+1})) \)

\footnote{Although the procedure works for an arbitrarily chosen vertex, for ease of demonstration, in Algorithm 3, we simply fix this initial vertex.}

\footnote{In contrast to Algorithm 2, for which the order of shorter cycles generated at each iteration is fixed, Algorithm 3 may produce an arbitrary order of the cycles. Thus, indices \( i_k \)’s appear in the subscript to emphasize that the output is some ordering of shorter cycles which cannot be deduced beforehand.}
Algorithm 3: Adjacent Cycle Decomposition 2

Input: A cycle \( \pi = (a_1 a_2 \cdots a_{|\text{supp}(\pi)|}) \)

Output: An adjacent cycle decomposition \( \pi = \pi_{k+1} \cdots \pi_1 \), with supports of \( \pi_j \), \( j = 1, 2 \cdots k+1 \), lying on paths of the Y-tree

1. Initialize: \( k \leftarrow 0 \), \( S \leftarrow \emptyset \).
2. Find an edge \((a_l, a_{l+1})\) by following the closed walk induced by \( \pi \) starting from \( a_1 \), with \( t = \min_{i \in [1, \cdots |\text{supp}(\pi)|]} \{ a_l \in \text{supp}(\pi), a_i \in B_j, a_{l+1} \notin B_j \} \) for some \( j \).
3. if such an edge \((a_l, a_{l+1})\) does not exist then
   4. Stop;
   5. else
      6. Add \((a_l, a_{l+1})\) to stack \( S \);
      7. end
8. while \( J_{(\text{supp}(\pi) \setminus \{a_l, a_j\})} \cdot J_{(\text{supp}(\pi) \setminus \{a_j\})} \cdot J_{(\text{supp}(\pi) \setminus \{a_l\})} = 1 \) do
    9. \( k \leftarrow k + 1 \);
    10. Assume the head \( b_1 \) of \( s_0 \) lies on \( B_1 \), and that the tail \( b_2 \) of \( s_0 \) lies on \( B_2 \). Let \( l = \min_{b \in [1, \cdots |\pi|]} \{ a_i \in B_2, a_{i+1} \notin B_2, a_i \text{ not previously visited} \} \);
    11. if \( a_{l+1} \in B_1 \) then
        12. if \( wt(p_\phi^*(a_0, b_1)) < wt(p_\phi^*(a_0, a_{l+1})) \) then
             13. \( \pi_{l} \leftarrow (b_1 \cdots a_l) \) and \( \pi' \leftarrow (b_1 a_{l+1} \cdots a_{|\text{supp}(\pi)|}) \);
             14. else
                 15. \( \pi_{l} \leftarrow (b_2 \cdots a_l a_{l+1}) \) and \( \pi' \leftarrow (b_1 a_{l+1} \cdots a_{|\text{supp}(\pi)|}) \);
             16. end
        17. end
    18. else
        19. Add \((a_l, a_{l+1})\) to the stack \( S \);
    20. end
21. \( \pi \leftarrow \pi' \);
22. if \( S \) is empty then
    23. Continue to follow the directed walk from the current vertex and go to step 2;
24. end
25. \( \pi_{k+1} \leftarrow \pi \).

Fig. 8: An example of the decomposition procedure when \( wt(p_\phi^*(a_0, b_1)) < wt(p_\phi^*(a_0, a_{l+1})) \). The cycle equals \( \pi = (1352748) \), with \( a_1 = 1 \in B_1 \). The first visited edge between branches is \((1, 3)\), i.e., \( a_l = 1 \in B_1, a_{l+1} = 3 \in B_2 \); the second visited edge between branches is \((5, 2)\), i.e., \( a_l = 5 \in B_2, a_{l+1} = 2 \in B_1 \). As \( a_{l+1} = 2 \in B_1 \), we decompose \((1352748)\) into two shorter cycles \((12748)\) and \((135)\), i.e., \((1352748) = (12748)(135)\).

or the condition
\[
wt(p_\phi^*(a_0, b_1)) > wt(p_\phi^*(a_0, a_{l+1}))
\]
is satisfied. Examples illustrating the difference are depicted in Fig. 8 and Fig. 9.

**Lemma 4** For a balanced cycle \( \pi \), the distance between \( \pi \) and \( e \) equals one half of their total displacement, i.e.,
\[
d_\phi(\pi, e) = \frac{1}{2}D_\phi(\pi, e).
\]

**Proof** The proof of the result follows along the same line as the proof of Lemma 5 and is therefore omitted.
Lemma 5 were false, and that one could actually have $T$ equal to $\pi \supseteq \cup_{l} \in B_1$; the second visited edge between branches is $(8, 4)$, i.e., $a_i = 8 \in B_3, a_{i+1} = 4 \in B_2$. As $a_{i+1} = 4 \notin B_1$, we add $(8, 4)$ to stack $S$ and move on to edge $(4, 7)$. As $a_{i+1} = 7 \in B_3$, we decompose $(1847235)$ into two shorter cycles $(47)$ and $(187235)$, i.e., $(1847235)=(47)(187235)$.

3.3 Case 3: Unbalanced Cycles

We start our exposition with a result that shows that if for some $i, j \in \{1, 2, 3\}$ one has $l_{ij} \neq l_{ji}$, i.e., the underlying cycle is unbalanced, then equality in the bound of Lemma 2 cannot be achieved.

**Lemma 5** For any unbalanced cycle $\pi$ we have

$$d_\varphi(\pi, e) \geq \frac{1}{2}D_\varphi(\pi, e).$$

**Proof** Since by Lemma 2 we have that $d_\varphi \geq \frac{1}{2}D_\varphi$, it suffices to show $d_\varphi(\pi, e) \neq \frac{1}{2}D_\varphi(\pi, e)$. Suppose that Lemma 5 were false, and that one could actually have

$$d_\varphi(\pi, e) = \frac{1}{2}D_\varphi(\pi, e).$$

Without loss of generality, assume $l_{12} \neq l_{21}$ and that $T^* = (\tau_1, \cdots, \tau_{|T^*|})$, with $\tau_j = (a_j, b_j)$, is a minimum weight transform converting $\pi$ into $e$. Next, define $\pi_j = \pi_{j-1} \tau_j$, for all $1 \leq j \leq |T^*|$, with $\pi_0 = \pi$. In our subsequent derivation, we occasionally use the notation $T'_{\pi} = (T_{\pi_{12}})$, with $\pi_{12}$ in the superscript, to emphasize that we are referring to the number of directed edges from $B_1$ to $B_2$ ($B_2$ to $B_1$) for a given permutation $\pi_j$.

The preceding claim, $d_\varphi(\pi, e) = \frac{1}{2}D_\varphi(\pi, e)$, implies that every transposition in a minimum weight transform $T^*$ is efficient, i.e., that for all $1 \leq j \leq |T^*|$, $D_\varphi(\pi_{j-1}, e) - D_\varphi(\pi_j, e) = 2\pi(\pi_j, e)$ holds.

$$= 2\varphi(a_j, b_j). \quad (3)$$

To arrive at a contradiction, we need the following two results.

**Claim 1** If $d_\varphi(\pi, e) = \frac{1}{2}D_\varphi(\pi, e)$ holds, then any minimum weight transform $T^* = (\tau_1, \cdots, \tau_{|T^*|})$ exclusively consists of transpositions of elements within the cycle support; in addition, each element in $supp(\pi)$ has to be swapped at least once, i.e.,

$$\cup_{j=1}^{|T^*|} supp(\tau_j) = supp(\pi).$$

**Proof** Given that in order to decompose a permutation $\pi$ each element in $supp(\pi)$ has to be swapped at least once, it holds $\cup_{j=1}^{|T^*|} supp(\tau_j) \supseteq supp(\pi)$. Suppose there exists an $a_j$ such that $a_j \notin supp(\pi)$, and $a_j \in \cup_{j=1}^{|T^*|} supp(\tau_j)$. This implies that there exists a transposition that introduces $a_j$ into the cycle support, thereby increasing the displacement of vertex $a_j$ from 0 to some positive value. Such transposition is inefficient, contradicting the fact that every transposition in the minimum weight transform has to be efficient.
Claim 2 An efficient transposition does not change the balancing property of branch pairs.

Proof For an efficient transposition \( \tau_j = (a_j b_j) \) in \( T^* \), we need to consider two cases separately: when \( a_j \) and \( b_j \) lie on the same branch; or when \( a_j \) and \( b_j \) lie on different branches of the defining tree.

When \( a_j \) and \( b_j \) both lie on the same branch, say \( B_1 \), then the transposition \( (a_j b_j) \) can neither change the number of directed edges from \( B_1 \) to \( B_2 \) nor the number of directed edges from \( B_2 \) to \( B_1 \). In other words, \( l_{12}^{\tau_{j-1}} = l_{12}^{\tau_j} \) and \( l_{21}^{\tau_{j-1}} = l_{21}^{\tau_j} \). Thus, if \( l_{12}^{\tau_{j-1}} = l_{21}^{\tau_{j-1}} \), then \( l_{12}^{\tau_j} = l_{21}^{\tau_j} \).

When \( a_j \) and \( b_j \) lie on different branches, say \( a_j \in B_1 \) and \( b_j \in B_2 \), then \( l_{12}^{\tau_j} = l_{12}^{\tau_{j-1}} - 1 \) and \( l_{21}^{\tau_j} = l_{21}^{\tau_{j-1}} - 1 \). Consequently, if \( l_{12}^{\tau_{j-1}} \neq l_{21}^{\tau_{j-1}} \), then \( l_{12}^{\tau_j} \neq l_{21}^{\tau_j} \).

By an inductive argument, we conclude that if every transposition in \( T^* \) is efficient, \( l_{12}^{\tau} \neq l_{21}^{\tau} \) implies \( l_{12}^{\tau_j} \neq l_{21}^{\tau_j} \), for all \( 0 \leq j \leq |T^*| \). This proves Claim 2.

Given that \( l_{12}^{\tau} \neq l_{21}^{\tau} \), based on Claim 2, we know that \( l_{12}^{\tau_{j}} \neq l_{21}^{\tau_{j}} \). In addition, if \( \pi_1 \cdots \pi_{|T^*|} = \pi_{|T^*|} = e \), then \( l_{12}^{\tau_{j}} \neq l_{21}^{\tau_{j}} \) reduces to \( l_{12} \neq l_{21} \). Since for the identity permutation \( e \), \( l_{12}^{\tau} = l_{21}^{\tau} = 0 \), we arrive at a contradiction. And this completes the proof.

We show next how to characterize the effect of inefficient transpositions on the gap between \( d_\varphi(\pi, e) \) and \( \frac{1}{2} D_\varphi(\pi, e) \) for the case of unbalanced cycles.

Lemma 6 For an unbalanced cycle \( \pi \), we have

\[
d_\varphi(\pi, e) = \frac{1}{2} D_\varphi(\pi, e) + \min_{a_i \in \text{supp}(\pi)} \varphi(a_0 a_i).
\]

Proof To prove Lemma 6, we first derive a lower bound on \( d_\varphi(\pi, e) \), which we subsequently show in a constructive manner to be tight for unbalanced cycles.

The idea behind the proof is to consider two types of transforms: 1) Transforms that only swap elements in \( \text{supp}(\pi) \), i.e., transforms for which \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) = \text{supp}(\pi) \); 2) transforms that also involve some element not in \( \text{supp}(\pi) \), i.e., transforms for which \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi) \).

In the subsequent derivation, we use the same setup as that in the proof of Lemma 5 but investigate both the scenario \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) = \text{supp}(\pi) \) and \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi) \). In addition, without loss of generality, we assume that \( l_{12} > l_{21} \).

Case 1) \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) = \text{supp}(\pi) \).

From the proof of Lemma 5 we know that there exists an inefficient transposition \( \tau_{j'} = (a_{j'} b_{j'}) \) in \( T^* \), where \( a_{j'} \in B_1 \) and \( b_{j'} \in B_2 \), that changes the balancing property of branch pairs when \( \bigcup_{i=1}^{|T^*|} \text{supp}(\tau_i) = \text{supp}(\pi) \). Without loss of generality, suppose that \( a_{j'} \in B_1 \), \( a_{j'} = \pi_{j'}(a_{j'}) \in B_2 \), and that \( b_{j'} \in B_2 \), \( b_{j'} = \pi_{j'}(b_{j'}) \notin B_1 \).

If \( b_{j'} \in B_3 \), then the inefficiency of \( \tau_{j'} \) equals

\[
\sigma_{\tau_{j'}} \geq 2 \varphi(a_{j'} b_{j'}) - \varphi(a_{j'} b_{j'}) - \varphi(a_0 b_{j'}) + \varphi(a_0 a_{j'}) = 2 \varphi(a_{j'} a_0) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0 a_i).
\]

Otherwise, if \( b_{j'} \in B_2 \), then the inefficiency of \( \tau_{j'} \) equals

\[
\sigma_{\tau_{j'}} \geq 2 \varphi(a_{j'} b_{j'}) - \varphi(a_{j'} b_{j'}) + (\varphi(b_{j'} a_0) - \varphi(b_{j'} b_{j'}) \cdot \mathcal{I}(\varphi(b_{j'} a_0) \leq \varphi(b_{j'} b_{j'}))) + \varphi(a_{j'} a_{j'}) \geq 2 \varphi(a_0 a_{j'}) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0 a_i).
\]

\[
\sigma_{\tau_{j'}} \geq 2 \varphi(a_{j'} b_{j'}) - \varphi(a_{j'} b_{j'}) + (\varphi(b_{j'} a_0) - \varphi(b_{j'} b_{j'}) \cdot \mathcal{I}(\varphi(b_{j'} a_0) > \varphi(b_{j'} b_{j'})) + \varphi(a_{j'} a_{j'}) \geq 2 \varphi(a_0 a_{j'}) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0 a_i).
\]
Thus,
\[
D_\varphi(\pi_{j-1}, e) - D_\varphi(\pi_j, e) \leq 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)} - 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)}.
\]
\[
\leq 2 \varphi_{(a_jb_j)} - 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)}.
\]

On the other hand, we know that for all 0 ≤ j ≤ |T*|,
\[
D_\varphi(\pi_{j-1}, e) - D_\varphi(\pi_j, e) \leq 2 \text{wt}(p_\varphi^*(a_j, b_j)).
\]
By summing up the terms in inequality (5) over 0 ≤ j ≤ j*, j* + 1 ≤ j ≤ |T*|, and adding to the resulting sum inequality (4), we obtain
\[
D_\varphi(\pi, e) - D_\varphi(\pi, e) \leq 2 \sum_{j=1}^{|T^*|} \text{wt}(p_\varphi^*(a_j, b_j)) - 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)}
\]
\[
\leq 2 \sum_{j=1}^{|T^*|} \varphi_{(a_jb_j)} - 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)},
\]
i.e.,
\[
\sum_{j=1}^{|T^*|} \varphi_{(a_jb_j)} \geq \frac{1}{2} D_\varphi(\pi, e) + \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)},
\]
which, as desired, establishes the claimed lower bound on the displacement for unbalanced cycles.

**Case 2:** \( \cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi) \)

We start our analysis by bounding the inefficiency of a minimum weight transform.

**Claim 3** For \( \cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi) \), the total inefficiency of a minimum weight transform is bounded as
\[
\sigma(T^*) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)}.
\]

**Proof** The proof is by inducting on \( |\cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi)| \).

**Base Case** Let \( |\cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi)| = 1 \). Assume that \( |\cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi)| = \{x_1\}, k_1 = \min\{i : x_1 \in \text{supp}(\tau_i), 1 \leq i \leq |T^*|\} \), and \( \tau_{k_1} = (x_1b_{k_1}) \), i.e., that \( x_1 \) is introduced into the cycle at the \( k_1^{th} \) transposition step of \( T^* \). The cycle \( \pi_{k_1} \) containing \( x_1 \) either remains unbalanced, or reduces to one of the two other cycle cases we analyzed – a cycle containing the central vertex or a balanced cycle.

Note that it is impossible for \( \pi \) to be converted into a balanced cycle via some other inefficient transpositions occurring before \( \tau_{k_1} \). To see this, suppose instead that \( \pi_{k_1} \) was balanced. From the analysis in Subsection 3.2, every transposition indexed by a value larger than \( k_1 \) is efficient, thus \( \tau_{k_1} \notin T^* \), which would contradict the starting assumption \( \cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi) \).

The inefficiency of \( \tau_{k_1} \) equals \( \sigma(\tau_{k_1}) = 2 \varphi_{\tau_{k_1}} - \ell_1 - \ell_2 \), where \( \ell_1, \ell_2 \) are the net reductions in the inefficiency of \( \pi_{k_1}^{-1}(x_1) \) and \( \pi_{k_1}^{-1}(b_{k_1}) \), respectively. Note that \( \ell_1 \) and \( \ell_2 \) are both upper bounded by \( \varphi_{\tau_{k_1}} \).

Let us first consider the case when \( \pi_{k_1} \) contains the central vertex, i.e., when \( x_1 = a_0 \). Since we know that \( \ell_1 = -\varphi_{\tau_{k_1}} \), it follows that\( \sigma(T^*) \geq \sigma(\tau_{k_1}) \geq 2 \varphi_{\tau_{k_1}} + \varphi_{\tau_{k_1}} - (-\varphi_{\tau_{k_1}}) \)
\[
= 2 \varphi_{\tau_{k_1}} \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi_{(a_0a_i)},
\]

One may argue that \( |\cup_{i=1}^{|T^*|} \text{supp}(\tau_i) \supset \text{supp}(\pi)| = 2 \) should also be considered as a base case for induction, given that it is possible to introduce two vertices outside \( \text{supp}(\pi) \) via a single transposition. This can be accomplished by first creating a 2-cycle which intersects \( \pi \), and then merging these two cycles in an efficient manner. However, it is not hard to see that such transpositions cannot appear in a minimum weight transform \( T^* \) for the scenario under consideration.
For the case when $\pi_k$ is balanced, without loss of generality, assume that $l_{12}^{B_k-1} > l_{21}^{B_k-1}$, and $x_1 \in B_1, b_{k_j} \in B_2$. Following the same line of argument as in Equations (8), we arrive at

$$\sigma(T^*) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i).$$

The most involved case is when $\pi_k$ is unbalanced. To facilitate the derivation, without loss of generality assume that $x_1 \in B_1$. Then,

$$\sigma(T^*) \geq \sigma_{k_1} + \sum_{i=k_1-1}^{T^*} \sigma_{\tau_i} \geq 2 \min_{a_i \in B_1 \cup \text{supp}(\pi)} \varphi(x_1a_i) + 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i) + 2 \min\{\varphi(x_1a_1), \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i)\} \geq 2 \min_{a_i \in B_1 \cup \text{supp}(\pi)} \varphi(x_1a_1) + \min_{a_i \in B_1 \cup \text{supp}(\pi)} \varphi(a_0a_i) \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i),$$

This completes the proof of the basis of induction.

**Induction Hypothesis:** Let $\bigcup_{i=1}^{T^*} \text{supp}(\tau_i) \setminus \text{supp}(\pi) = \{x_1, x_2, \ldots, x_m\}$, and assume that $k_j = \min\{i : x_j \in \text{supp}(\tau_i), 1 \leq i \leq m\}$, for all $1 \leq j \leq m$. In addition, assume that inequality (8) holds for $1 \leq m < |T^*|$. Inequality (6) also holds for the case $|\bigcup_{i=1}^{T^*} \text{supp}(\tau_i) \setminus \text{supp}(\pi)| = m + 1$, since

$$\sum_{i}^{m+1} \sigma_{\tau_i} = \sum_{i}^{m} \sigma_{\tau_i} + \sigma_{\tau_{m+1}} \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i) + \sigma_{\tau_{m+1}} \geq 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i),$$

where the first inequality follows from the induction hypothesis, while the second inequality follows from the fact that $\sigma_{\tau_{m+1}} \geq 0$. This proves Claim 3.

As a result of Claim 3, the following is true for all $1 \leq j \leq |T^*|:

$$D_\varphi(\pi_{j-1}, e) - D_\varphi(\pi_j, e) = 2 \text{wt}(p^*_\varphi(a_jb_j)) - \sigma_{\tau_j} = 2 \varphi(a_jb_j) - \sigma_{\tau_j},$$

(9)

where $\sigma_{\tau_j}$ is the inefficiency of $\tau_j$, and $\sigma_{\tau_j} = 0$ if $\tau_j$ is an efficient transposition. Telescoping equation (8) over $j$, where $1 \leq j \leq |T^*|$, gives

$$D_\varphi(\pi, e) - D_\varphi(\pi, e) \leq \sum_{j=1}^{|T^*|} 2 \text{wt}(p^*_\varphi(a_jb_j)) - \sum_{j=1}^{|T^*|} \sigma_{\tau_j} = 2 \sum_{j=1}^{|T^*|} \varphi(a_jb_j) - \sigma(T^*) \leq 2 \sum_{j=1}^{|T^*|} \varphi(a_jb_j) - 2 \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i),$$

i.e.,

$$\sum_{j=1}^{|T^*|} \varphi(a_jb_j) \geq \frac{1}{2} D_\varphi(\pi, e) + \min_{a_i \in \text{supp}(\pi)} \varphi(a_0a_i).$$
Fig. 10: 1) The defining Y-tree and cycle $\pi = (357)$; 2) Since $\varphi(58) < \min\{\varphi(78), \varphi(38)\}$, and since the vertices 5 and 7 belong to different branches, we can swap 5 and 8 to include the central vertex into the cycle support. However, as illustrated in 3) and 4), when the cycle equals $\pi = (2357)$ and when $\varphi(28) < \min\{\varphi(58), \varphi(78)\}$, we have to first swap vertices 2 and 3 before swapping vertices 2 and 8.

where $\sum_{j=1}^{T}[\varphi(a_{j}b_{j})]$ is the minimum transposition cost.

Consequently, we have

$$d_{\varphi}(\pi, e) \geq \frac{1}{2}D_{\varphi}(\pi, e) + \min_{a_{i} \in supp(\pi)} \varphi(a_{0}a_{i}),$$

(11)

independent of $\cup_{j=1}^{n}supp(\tau_{j}) = supp(\pi)$ or $\cup_{j=1}^{n}supp(\tau_{j}) \supset supp(\pi)$.

The bound in inequality (11) is tight – this claim can be proved in a constructive manner via an application of Algorithm 4 and Algorithm 1. Therefore, we conclude that

$$d_{\varphi}(\pi, e) = \frac{1}{2}D_{\varphi}(\pi, e) + \min_{a_{i} \in supp(\pi)} \varphi(a_{0}a_{i}).$$

Algorithm 4: Adjacent Cycle Decomposition 3

| Input: A cycle $\pi = (a_{1}a_{2}\cdots a_{|supp(\pi)|})$ |
|----------------------------------------------------------|
| Output: An adjacent cycle decomposition $\pi = \left\{ \begin{array}{ll} \pi_{k+1}\cdots\pi_{1}(a_{0}a_{j})(a_{j}a_{j+1}), & \text{if } a_{j} \text{and } a_{j+1} \text{lie on the same branch;} \\ \pi_{k+1}\cdots\pi_{1}(a_{0}a_{j}), & \text{if } a_{j} \text{and } a_{j+1} \text{lie on different branches}, \end{array} \right. $ |
| with the supports of $\pi_{i}, i = 1, \cdots, k + 1$, lying on paths of the Y-tree |
| $a_{j} \leftarrow \text{argmin}_{a_{i} \in supp(\pi)} \varphi(a_{i}a_{0});$ |
| if $a_{j}$ and $a_{j+1}$ lie on the same branch then |
| $\pi^{*} \leftarrow \pi_{j}(a_{j}a_{j+1});$ |
| else |
| $\pi^{*} \leftarrow \pi;$ |
| end |
| $\pi^{*} \leftarrow \pi^{*}(a_{0}a_{j});$ |
| Call Algorithm 2 to obtain $\pi^{*} = \pi_{k+1}\cdots\pi_{1}.$ |

Observe that Algorithm 4 behaves differently depending on the relative positions of $a_{j}$ and $a_{j+1}$. An illustrative example, describing how $a_{j}$ and $a_{j+1}$ lying on the same branch influences the cycle decomposition in Algorithm 4, is depicted in Fig. 10.

The underlying computational steps of the complete decomposition algorithm are presented in Algorithm 5. The focal point of the procedure are the adjacent cycle decomposition routines that decompose a cycle into products of adjacent subcycles with supports on paths of the Y-tree. For each subcycle, a minimum weight transform can be efficiently found via Algorithm 1. Given that adjacent cycles may only share one element, multiplying the minimum weight path-cost decompositions in the correct order produces a minimum Y-tree weight transform of the input cycle.

As a final remark, note that the exposition in Subsection 3.1, Subsection 3.2 and Subsection 3.3 implicitly assumes that certain properties (such as balancedness) of a specific cycle are known beforehand. However, if this is not the case, an additional procedure has to be designed to test for such properties, as given in Algorithm 5.
Algorithm 5: Minimum Weight Transform of a Cycle

Input: A cycle $\pi$
Output: A minimum weight transform from $\pi$ to $e$

1. if $a_0$ is in the cycle support then
   2. Call Algorithm 2;
else
   4. Initialize $l_{ij} \leftarrow 0$, for all $i, j \in \{1, 2, 3\}$ and follow the closed walk induced by $\pi$ starting from an arbitrary vertex in $\text{supp}(\pi)$;
   5. while there exists a vertex in $\text{supp}(\pi)$ that has not been visited do
      6. if an edge bridges $B_i$ to $B_j$, for $i, j \in \{1, 2, 3\}$, then
         7. $l_{ij} \leftarrow l_{ij} + 1$;
      end
   end
   9. if $l_{ij} = l_{ji}$ for all $i, j \in \{1, 2, 3\}$ then
      10. Call Algorithm 3;
   else
      11. Call Algorithm 4;
   end
end
16. Apply Algorithm 1 for each subcycle in the decomposition of $\pi$ returned by the preceding if-else command.

We summarize our findings in the following theorem.

Theorem 1 Let $\phi$ be a Y metric-tree weight function and $\pi$ be a cycle permutation. If $\pi$ is unbalanced, then

$$d_\phi(\pi, e) = \frac{1}{2}D_\phi(\pi, e) + \min_{a_i \in \text{supp}(\pi)} \phi(a_0a_i).$$

Otherwise,

$$d_\phi(\pi, e) = \frac{1}{2}D_\phi(\pi, e).$$

Note that it is easy to show that the results of Theorem 1, as well as Algorithm 2, Algorithm 3 and Algorithm 4, are all valid under the more general star metric-tree weights. The proofs in support of these results are omitted since they follow along the same lines as the proofs in Theorem 1.

3.4 Computational Complexity

Careful examination of Algorithm 5 reveals that three major computational steps are involved in finding a minimum weight transposition decomposition, including: 1) Identifying the type of cycle; 2) conducting an adjacent cycle decomposition; 3) solving the individual subcycles decomposition problems with supports on paths. From a complexity point of view, step 1) is the least costly procedure as it requires $O(n)$ operations for checking whether the central vertex $a_0$ is in the cycle or not. If the central vertex belongs to the cycle, the decomposition calls for Algorithm 2, which requires $O(n)$ operations. Otherwise, in order to check whether the given cycle is balanced or unbalanced, one has to traverse the cycle to count the number of edges crossing branches and to store/compare the values of $l_{ij}$ for all pairs of $i, j \in \{1, 2, 3\}$. This counting procedure requires less than $O(n^2)$ operations.

Since Step 2) calls for different decomposition approaches based on the type of the cycle, we consider two cases:

1. If Algorithm 2 and Algorithm 3 are used, we follow the given cycle and at each vertex we check whether the optimal decomposition conditions are met. As each check requires constant computational time, and as Algorithm 2 and Algorithm 3 terminate when each vertex in the cycle is checked exactly once, their time complexities are $O(n)$.

2. Algorithm 4 is similar to Algorithm 2 except for the fact that an additional minimization problem is involved, which requires $O(n^2)$ operations.
Fig. 11: Merging two cycles creates a balanced cycle: In 1) there are two cycles (1 4 5) and (2 6 3); the merged cycle after applying transposition (1 3) is presented in 2).

In the third step, we have to solve multiple path cycle decompositions individually. It can be shown inductively that at most \( m - 2 \) operations are needed to find a minimum weight transform for such a cycle decomposition problem, where \( m \) is the length of the cycle. In addition, we know that \( \sum_{i=1}^{k} |\text{supp}(\pi_i)| = |\text{supp}(\pi)| + (k - 1) \), where \( k \) is the number of path-case cycles in an adjacent cycle decomposition of \( \pi \). Thus, solving all the path cycle decompositions may be accomplished with \( O(n^3) \) operations.

Therefore, Algorithm 5 has time complexity \( O(n^2) \).

4 General Permutations

Computing the weighted transposition distance between permutations with multiple cycles under the Y-tree weights model is significantly more challenging than computing this distance for arbitrary permutations. Nevertheless, for special classes of permutations whose cycle embedding in a fixed and predefined Y-tree is “planar” (i.e., the cycles do not intersect when embedded in the Y-tree, although an individual cycle may cross itself), the weighted transposition distance can be computed exactly in time \( O(n^2) \). In addition, we develop a polynomial-time, constant-approximation algorithm for general permutations.

Recall the solution to the decomposition problem, when all transposition weights are equal: perform the disjoint cycle decomposition and then sort each cycle independently. However, this independent cycle decomposition strategy does not work for general weight functions, as illustrated in Fig. 11. There, the permutation \( \pi = (4,6,2,5,1,3,7) \) can be decomposed by first swapping the vertices 1 and 3, thereby merging the cycles (1 4 5) and (2 6 3). As the resulting cycle is balanced, it can be subsequently sorted via a sequence of efficient transpositions. Since the transposition (1 3) is efficient as well, the resulting transform has cost \( d_{\phi}(\pi, e) = \frac{1}{2}D_{\phi}(\pi, e) \). But there also exist examples involving non-uniformly weighted transpositions where merging cycles does not lead to minimum sorting cost (see Fig. 12).

We now focus on the family of permutations that have a planar embedding in the Y-tree. To simplify our exposition, we also assume that there are no cycles in the functional digraph of the permutation that can be embedded on paths in the Y-tree. Furthermore, we suppose that one particular Y-tree representation for the metric weights is fixed and not changed throughout the process. The problem of identifying if a set of cycles of a permutation may be embedded without intersection in some Y-tree defining the metric weights will not be considered.

The decomposition steps are listed in Algorithm 6. We next prove the validity of the method in terms of producing a minimum cost decomposition and then show that the derived results hold for the more general case of planar embeddings without constraints.

The key observation behind the proof is that if \( c_1 \) and \( c_2 \) are cycles with nonintersecting functional digraphs, then their minimum merging cost equals

\[
\min_{v_1 \in \text{supp}(c_1), w_j \in \text{supp}(c_2)} \phi(v_1, w_j).
\]

Now, given that some of the non-intersecting cycles can be unbalanced, there may exists a gap between the weighted distance and one half of the displacement \( \frac{1}{2}D(c, e) \) of the cycle. Thus, the problem of finding a minimum weight transform reduces to minimizing this gap for all cycles simultaneously. Observe that the
Fig. 12: An example of merging two cycles leads to suboptimal solution: For permutation $\pi = (4, 8, 3, 9, 2, 7, 6, 1, 5, 10)$, via exhaustive search we know that the minimum decomposition cost is $\frac{1}{2}D(\pi, e)$ instead of $\frac{1}{2}D(\pi, e) + \phi(56)$, which is obtained via merging cycles.

Algorithm 6: Minimum Weight Transform for “planar” permutations

Input: A permutation $\pi$ with nonintersecting cycles
Output: A minimum weight transform $T^*$ from $\pi$ to $e$

1. Label cycles in order of increasing “distance” to the central vertex $a_0$;
2. while there is more than one cycle do
3.  if the furthest cycle is balanced then
4.  Decompose the cycle;
5.  else
6.  Merge it with the second-furthest cycle with minimum merging cost;
7. end
8. end
9. Call Algorithm 5 for the resulting single cycle.

Theorem 2 Algorithm 6 outputs a minimum weight transform for permutation whose functional digraph is planar and without cycles that may be embedded on paths.

Proof The proof follows by induction on the number of cycles of the permutation.

Base Case: We show that when the input permutation has a single cycle or two cycles, Algorithm 6 outputs a minimum weight transform.

When the input permutation $\pi$ contains only one cycle, optimality of the output trivially holds. When the input permutation contains two cycles, denoting the inner cycle as $c_1$ and the outer cycle as $c_2$, the properties of both of these two cycles needed to be considered: if $c_2$ is balanced, it is decomposed separately and the problem reduces to the single cycle case; otherwise, three separate sub-scenarios involving $c_1$ have to be investigated, as depicted in Fig. 13.

To prove the optimality of the algorithm, we compare the minimum costs of decomposition when cycles are decomposed individually and when they are merged. Under scenario 1), on the one hand, the minimum cost given that the cycles are merged equals

$$\frac{1}{2}D_\phi(c_1c_2, e) + \min_{w_j \in \text{supp}(c_1), v_i \in \text{supp}(c_2)} \phi(v_iw_j),$$

where the second term is the minimum merging cost. On the other hand, from the analysis in Section 3, we know that if $c_1$ and $c_2$ are decomposed separately, the minimum cost equals

$$\frac{1}{2}D_\phi(c_1c_2, e) + \min_{v_i \in \text{supp}(c_2)} \phi(v_ia_0).$$

It is straightforward to see that

$$\min_{w_j \in \text{supp}(c_1), v_i \in \text{supp}(c_2)} \phi(v_iw_j) \leq \min_{v_i \in \text{supp}(c_2)} \phi(v_ia_0).$$

The closest point to the central vertex on a cycle $e$ is defined as $\text{argmin}_{v_i \in \text{supp}(e)} \phi(a_0v_i)$. 

---

8 The closest point to the central vertex on a cycle $e$ is defined as $\text{argmin}_{v_i \in \text{supp}(e)} \phi(a_0v_i)$. 

Fig. 13: 1) The inner cycle $c_1$ (in the example, $c_1 = (1648)$) contains the central vertex in its support; 2) The inner cycle $c_1$ (in the example, $c_1 = (184725)$) is balanced; 3) The inner cycle $c_1$ (in the example, $c_1 = (184275)$) is unbalanced.

Therefore, a minimum cost decomposition can be obtained by merging cycles. Similarly, under scenarios 2) and 3), the minimum cost under cycle-merging is at most

$$\frac{1}{2} D_\phi(c_1, c_2; e) + \min_{w_j \in \text{supp}(c_1), v_i \in \text{supp}(c_2)} \phi(v_i w_j) + \min_{v_i \in \text{supp}(c_2)} \phi(v_i a_0),$$

where the third term is only to be included if the cycle obtained by merging $c_1$ and $c_2$ is unbalanced.

Consequently, the output of Algorithm 6 is optimal for the base case of induction.

**Induction Hypothesis:** If there are less than $k + 1$ cycles in the permutation $\pi$, Algorithm 6 gives an optimal solution.

**Induction Step:** When there are $k + 1$ cycles in $\pi$, by the induction hypothesis, one only needs to consider the innermost cycle and the resulting cycle obtained by processing the $k$ outermost cycles. This reduces to the base case, which shows that Algorithm 6 gives an optimal solution.

The result of Theorem 2 may be readily generalized to the case when the input permutation contains non-intersecting cycles embedded on paths. The key observation is that the existence of such cycles may reduce the merging cost of two cycles that include edges between branches. In each iteration, one starts by checking the balancing property of the outermost cycle containing edges between branches, which requires $O(n^2)$ operations; when cycle-merging is considered, assume that there are $h$ path-embedded cycles that lie “between” two cycles spanning across branches. Exhaustive search for finding an optimal path-embedded cycle based merging requires $O(2^h)$ operations. A randomly selected permutation of $n$ elements has an expected number of cycles equal to $H_n$, the $n$-th harmonic number, which is asymptotically equal $\log n$. Hence, the expected complexity of merging two cycles is linear in $n$ and only $\log n$ iterations are needed to complete the process. Therefore, the average running time of the generalized form of algorithm Algorithm 6 equals $O(n^2 \log n)$. Note that the worst-case complexity may still be exponential in $n$.

For general permutations which may contain intersecting cycles, no efficient optimal decomposition procedure is currently known. Nevertheless, we present next a straightforward linear-time $5/3$-approximation algorithm, described in Algorithm 7.

To explain the motivation behind the steps of Algorithm 7, consider the illustrative example in Fig. 14. There, a permutation $\pi$ is depicted which can be embedded in the defining graph such that there exists a cycle containing the central vertex. The support of the cycle lies on a path and the edge emanating from $a_0$ intersects all cycles across branches. In this case, one can merge all the cycles branches efficiently using the aforementioned intersecting edge.

To reduce an arbitrary permutation to the desired structure depicted in Fig. 14 part 1), in the first step of the algorithm, we decompose all cycles with supports on paths. This procedure requires $O(n)$ computational steps. Subsequent steps 5 and 6 introduce the central vertex $a_0$ into the decomposition procedure, which in the
Algorithm 7: Constant-Approximation Algorithm for General Permutations

Input: A arbitrary permutation \( \pi \)
Output: A constant-approximation transform \( T^{\pi} \) from \( \pi \) to \( e \)
1. Decompose all the cycles with supports lying on paths, as well as the cycle containing the central vertex independently and denote the resulting permutation by \( \pi^* \);
2. if \( \pi^* = e \) then
   3. Stop;
4. end
5. Define \( a_1, a_2, a_3 \) to be the farthest nonfixed points with outgoing edges across branches on \( B_1, B_2, B_3 \), respectively; in addition, assume \( \phi(0; (a_1), 1) = \min(\phi_1(0; (a_1), 1), \phi_2(0; (a_1), 1)) \);
6. \( \pi^* \leftarrow \pi^*(0; a_1) \);
7. Invert \( \pi^* \) to obtain \( (\pi^*)^{-1} \);
8. Suppose there are \( m \) cycles in \( (\pi^*)^{-1} \) with edges across branches, and let \( b_1, b_2, \ldots, b_m \in B_1 \) denote the heads of outgoing edges across branches, where the labels indicate their proximity to the central vertex \( a_0 \). In addition, let \( b_0 = a_0 \);
9. for \( i = 1 : m \) do
10.    \( \pi^* \leftarrow \pi^*(b_i b_{i-1}) \);
11. end
12. Call Algorithm 5.

Fig. 14: Cycle structure behind Algorithm 7.

worst case takes \( O(n) \) operations. Note that introducing \( a_0 \) may require some preprocessing steps as described in Algorithm 4.

To this end, it is important to note that we only have the incoming edge into \( a_0 \), rather than the desired outgoing edge from \( a_0 \) intersecting all cycles across branches. Thus, the inverse of \( \pi^* \) is needed to complete the construction of the desired structure for subsequent cycle merging. As described in steps 8–11 of Algorithm 7, cycles are merged via the transpositions \( (b_i b_{i-1}) \), for \( i = 1, 2, \ldots, m \). The computational cost of this merging process is \( O(n) \). In step 12, the resulting cycle is processed by Algorithm 5. Note that the running time of step 12 equals \( O(n) \) even though the time complexity of Algorithm 5 itself is \( O(n^2) \). This is due to the fact that the input cycle to Algorithm 5 will always contain the central vertex, and as a result, the most computationally expensive steps in Algorithm 5 are not executed. Therefore, Algorithm 7 has complexity linear in \( n \).

As for the performance guarantee, it is easy to see that the only step that introduces inefficiency in the decomposition is the introduction of the central vertex in step 6. Thus \( d(\pi, e) \) may be bounded as

\[
\frac{1}{2} D(\pi, e) \leq d(\pi, e) \leq \frac{1}{2} D_\phi(\pi, e) + \varphi(0; (a_0 a_1)),
\]

where the upper bounded is the cost of the output of Algorithm 7. In addition, we notice that \( \varphi(0; (a_0 a_1)) \leq \frac{1}{3} D(\pi, e) \). Thus the approximation factor is \( 5/3 \), as claimed.
5 Conclusion

We introduced the notion of similarity distance between rankings under Y-tree weights and presented a polynomial-time algorithm for computing the distance between cycle permutations in terms of the displacement function. The algorithm was centered around the idea of adjacent cycle decomposition, i.e., rewriting a cycle as a product of adjacent/disjoint shorter cycles, where the support of each cycle can be embedded on a path in the defining graph of the Y-tree.

We also described an exact polynomial-time decomposition algorithm for permutations that may be embedded in the Y-tree as non-intersecting cycles, and the procedure reduced to finding the shortest path between two non-intersecting cycles. As for general permutations, we developed a linear time, 5/3-approximation algorithm which is governed by the fact that if there exists a directed edge emanating from the central vertex that intersects all cycles across branches, then all cycles across branches can be merged efficiently.

References

1. Aerts, S., Lambrechts, D., Maity, S., Van Loo, P., Coessens, B., De Smet, F., Tranchevent, L.C., De Moor, B., Marynen, P., Hassan, B., et al.: Gene prioritization through genomic data fusion. Nature biotechnology 24(5), 537–544 (2006)
2. Ailon, N., Charikar, M., Newman, A.: Aggregating inconsistent information: Ranking and clustering. J.ACM 55(5), 23:1–23:27 (2008)
3. Bafna, V., Pevzner, P.: Sorting by transpositions. SIAM Journal on Discrete Mathematics 11(2), 224–240 (1998)
4. Bienstock, D.: On embedding graphs in trees. Journal of Combinatorial Theory, Series B 49(1), 103 – 136 (1990)
5. Diaconis, P., Graham, R.L.: Spearman's footrule as a measure of disarray. Journal of the Royal Statistical Society. Series B (Methodological) 39(2), 262–268 (1977)
6. Dwork, C., Kumar, R., Naor, M., Sivakumar, D.: Rank aggregation methods for the web. ACM (2001). DOI 10.1145/371920.372165
7. Fagin, R., Kumar, R., Sivakumar, D.: Efficient similarity search and classification via rank aggregation. ACM (2003). DOI 10.1145/872757.872795
8. Farnoud, F., Touri, B., Milenkovic, O.: Nonuniform vote aggregation algorithms. CoRR abs/1206.5343, 1–5 (2012)
9. Farnoud (Hassanzadeh), F., Milenkovic, O.: Sorting of permutations by cost-constrained transpositions. IEEE Trans. Inf. Theor. 58(1), 3–23 (2012)
10. Goulden, I.P., Jackson, D.M.: Combinatorial enumeration. Dover Publications (2004)
11. Hofri, M.: Analysis of algorithms: Computational methods and mathematical tools. Oxford University Press (1995)
12. Huth, T., Mattfeld, D.C.: Integration of vehicle routing and resource allocation in a dynamic logistics network. Transportation Research Part C: Emerging Technologies 17(2), 149 – 162 (2009)
13. Kumar, R., Vassilvitskii, S.: Generalized distances between rankings. ACM (2010). DOI 10.1145/1772690.1772749
14. van Lint, J.H., Wilson, R.M.: A course in combinatorics. Cambridge University Press (2001)
15. Liu, Y.T., Liu, T.Y., Qin, T., Ma, Z.M., Li, H.: Supervised rank aggregation. ACM (2007). DOI 10.1145/1242572.1242638
16. Marchant, T.: Valued relations aggregation with the borda method. Journal of Multi-Criteria Decision Analysis 5(2), 127–132 (1996)
17. Pihur, V., Datta, S., Datta, S.: Rankaggreg, an r package for weighted rank aggregation. BMC Bioinformatics 10(1), 62 (2009)
18. Renda, M.E., Straccia, U.: Web metasearch: rank vs. score based rank aggregation methods. ACM (2003). DOI 10.1145/952532.952698
19. Ricci, F., Werthner, H.: Case base querying for travel planning recommendation. Information Technology and Tourism 4(3-4), 215–226 (2001)
20. Sculley, D.: Rank aggregation for similar items. pp. 587–592. Citeseer (2007)
21. Tranchevent, L.C., Barriot, R., Yu, S., Van Vooren, S., Van Loo, P., Coessens, B., De Moor, B., Aerts, S., Moreau, Y.: Endeavour update: a web resource for gene prioritization in multiple species. Nucleic acids research 36(suppl 2), W377–W384 (2008)
22. Young, H.P.: Condorcet's theory of voting. The American Political Science Review 82(4), 1231–1244 (1988)
23. van Zuylen, A., Williamson, D.P.: Deterministic pivoting algorithms for constrained ranking and clustering problems. Mathematics of Operations Research 34(3), 594–620 (2009)