Rank Aggregation: Together We're Strong

Frans Schalekamp* Anke van Zuylen†

Abstract

We consider the problem of finding a ranking of a set of elements that is "closest to" a given set of input rankings of the elements; more precisely, we want to find a permutation that minimizes the Kendall-tau distance to the input rankings, where the Kendall-tau distance is defined as the sum over all input rankings of the number of pairs of elements that are in a different order in the input ranking than in the output ranking. If the input rankings are permutations, this problem is known as the Kemeny rank aggregation problem. This problem arises for example in building meta-search engines for Web search, aggregating viewers' rankings of movies, or giving recommendations to a user based on several different criteria, where we can think of having one ranking of the alternatives for each criterion. Many of the approximation algorithms and heuristics that have been proposed in the literature are either *positional*, *comparison sort* or *local search* algorithms. The rank aggregation problem is a special case of the (weighted) feedback arc set problem, but in the feedback arc set problem we use only information about the preferred relative ordering of pairs of elements to find a ranking of the elements, whereas in the case of the rank aggregation problem, we have additional information in the form of the complete input rankings. The positional methods are the only algorithms that use this additional information. Since the rank aggregation problem is NP-hard, none of these algorithms is guaranteed to find the optimal solution, and different algorithms will provide different solutions. We give theoretical and practical evidence that a combination of these different approaches gives algorithms that are superior to the individual algorithms. Theoretically, we give lower bounds on the performance for many of the "pure" methods. Practically, we perform an extensive evaluation of the "pure" algorithms and combinations of different approaches. We give three recommendations for which (combination of) methods to use based on whether a user wants to have a very fast, fast or reasonably fast algorithm.

1 Introduction

We consider the problem of finding a ranking of a set of elements that best represents a given set of input rankings of the elements. This is a classical problem from social choice and voting theory, in which each voter gives a preference on a set of alternatives, and the system outputs a single preference order on the set of alternatives based on the voters' preferences.

There has been a lot of interest in this problem in the computer science community in recent years. The rank aggregation problem arises when building meta-search engines for Web search, where we want to combine the rankings obtained by different algorithms into a representative ranking. For example, Dwork, Kumar, Naor and Sivakumar [9] propose combining the rankings of individual search engines to get more robust rankings that are not sensitive to the various shortcomings and biases of individual search engines. Other applications arise in ranking movies, hotels, etc. based on the ratings given by users, or giving recommendations to a user based on several different criteria, where we can think of having one ranking of the alternatives for each criterion.

In the social choice literature, a widely accepted objective for aggregating voters' preferences, if each voter gives a complete ranking of the candidates, is Kemeny rank aggregation (see for example [21, 22]): Given two permutations π, σ of $\{1, \ldots, n\}$, the Kendall-tau distance is defined as

(1.1)
$$\mathcal{K}(\sigma, \pi) = \sum_{i=1}^{n} \sum_{j=i}^{n} \mathbf{1}\{(\pi(i) < \pi(j)) \ \&\ (\sigma(i) > \sigma(j))\},$$

where $\mathbf{1}\{\cdot\}$ is the indicator function. Given a set of permutations $\pi_1, \ldots, \pi_k$ of $\{1, \ldots, n\}$, the Kemeny rank aggregation problem seeks a permutation σ that minimizes the number of pairwise disagreements with the input permutations, i.e. $\frac{1}{k} \sum_{\ell=1}^{k} \mathcal{K}(\pi_\ell, \sigma)$.

We also consider the *partial* rank aggregation problem. A partial ranking of V is a function $\pi : V \to \{1, \ldots, |V|\}$, where the function π does not have to be

*Institute for Theoretical Computer Science, Tsinghua University, Beijing, China. frans@mail.tsinghua.edu.cn. Research performed in part while the author was at Nature Source Genetics, Ithaca, NY.

†Institute for Theoretical Computer Science, Tsinghua University, Beijing, China. anke@mail.tsinghua.edu.cn. Research partly supported by NSF grant CCF-0514628 and performed in part while the author was at the School of Operations Research and Information Engineering at Cornell University, Ithaca, NY.

one-to-one; in other words, a partial ranking is a ranking with ties. Fagin et al [10, 11, 12] proposed a set of different distance measures between two partial rankings and a permutation and a partial ranking. We will follow Ailon [1] and define the distance $\mathcal{K}(\pi,\sigma)$ between two partial rankings π,σ as in Equation (1.1), and following Ailon we let the partial rank aggregation problem be the problem of finding a *permutation* that minimizes the sum of the distances to given input partial rankings. We note that, if we require the output ranking to be a permutation, then the other extensions of the Kendall-tau distance that were proposed by Fagin et al. [11, 10] just add a constant (that depends on the input rankings but not on the output permutation) to the distance $\frac{1}{k}\sum_{\ell=1}^{k}\mathcal{K}(\pi_\ell,\sigma)$.

In this paper we will use the term full rank aggregation if the input rankings are known to be permutations, and partial rank aggregation if the input rankings are partial rankings.

A problem related to rank aggregation is the feedback arc set problem in tournaments. A tournament is a complete directed graph $G = (V, A)$. The feedback arc set problem in tournaments asks for the smallest set of arcs A' such that $(V, A\backslash A')$ is acyclic. Equivalently, we want to find a permutation of the vertices π that minimizes the number of "back-arcs"; the arcs that go from right to left if we order the vertices according to π. A generalization of this is the weighted feedback arc set problem, where given a set of vertices V and a nonnegative weight $w_{(i,j)}$ for every ordered pair (i,j), we want to find a permutation π that minimizes $\sum_{i,j:\pi(i)<\pi(j)} w_{(j,i)}$. We say the weights satisfy probability constraints if $w_{(i,j)} + w_{(j,i)} = 1$ for all $i,j \in V$, and the triangle inequality if $w_{(i,j)} + w_{(j,k)} \geq w_{(i,k)}$ for every $i,j,k \in V$.

Rank aggregation is a special case of the weighted feedback arc set problem, since we can let $w_{(i,j)} = \frac{1}{k}\sum_{\ell=1}^{k}\mathbf{1}\{\pi_\ell(i) < \pi_\ell(j)\}$, and then the weighted feedback arc set problem then seeks σ that minimizes $\sum_{i,j:\sigma(i)<\sigma(j)}\frac{1}{k}\sum_{\ell=1}^{k}\mathbf{1}\{\pi_\ell(i) < \pi_\ell(j)\} = \frac{1}{k}\sum_{\ell=1}^{k}\mathcal{K}(\pi_\ell,\sigma)$. It is easily checked that the weights satisfy the triangle inequality. In the case of full rank aggregation, the weights also satisfy the probability constraints.

Finding the Kemeny optimal ranking is NP-hard, even when the number of input permutations is only 4 ([9]). However, in recent years, many algorithms have been proposed in the theoretical computer science community that give provably good solutions. An α-approximation algorithm is an algorithm that runs in polynomial time and produces a solution for which the objective value is within a factor α of the optimal value. The *performance guarantee* α of an approximation algorithm is thus (an upper bound on) the worst case ratio of the objective value of the solution returned by the algorithm, and the objective value of the optimal solution. Several algorithms are known for rank aggregation with performance guarantees of 2 or less, see for example [2], [1], [7], [20], [19]. For full rank aggregation, there is even a Polynomial-Time Approximation Scheme (PTAS) [16]: A PTAS is an algorithm that for any fixed $\epsilon > 0$ finds a solution with performance guarantee $(1+\epsilon)$ in time polynomial in the size of the input, but not necessarily polynomial in $\frac{1}{\epsilon}$. In particular, the running time of the PTAS in [16] is doubly exponential in $\frac{1}{\epsilon}$.

Roughly speaking, many of the algorithms fall into one of four categories: positional methods, comparison sort methods, local search algorithms and "hybrid" methods, that combine ideas from the previous three categories. A positional method for full rank aggregation seeks a permutation in which the *position* of each element is "close to the average position" of the element in the input permutations. Examples of positional methods are Borda's method [3, 7], Footrule aggregation [8] and Pick-a-Perm [2]. Borda's method finds a permutation that minimizes the distances between the elements' positions and their mean positions in the input permutations, Footrule aggregation finds a permutation σ that minimizes $\sum_{i\in V}\frac{1}{k}\sum_{\ell=1}^{k}|\pi_\ell(i)-\sigma(i)|$, and Pick-a-Perm chooses one of the input permutations at random, thus returning a permutation in which the expected position of each element is its mean position. Comparison sort algorithms use a comparison relation to sort the elements, where the comparison relation is not necessarily transitive in this context. The result of a comparison sort algorithm hence depends on the comparison relation and the sorting algorithm. Examples of comparison sort methods are QuickSort, MergeSort and InsertionSort. The local moves in the local search algorithms we consider are "single vertex moves", where we may move one element at a time to a different position in the ranking. An example of a "hybrid" method is Copeland's method, which sorts the elements based on the number of elements they would beat in a pairwise majority contest.

Both theoretically and practically there is evidence that combinations of positional, comparison sort and local search methods outperform the "pure" methods. For example, Ailon, Charikar and Newman [2] show that taking the best of the best input permutation, and the permutation obtained by running QuickSort, is a $\frac{11}{7}$-approximation algorithm. In the implementation study done by Dwork et al. [9], the best performing method was inspired by Copeland's method [6] followed by InsertionSort, where Copeland's method can itself be seen as a hybrid method.

In this paper we compare these different types of algorithms and combinations of the different approaches. We give lower bounds on the performance guarantees of most of the existing methods. However, a bad example for one type of algorithm is typically not a bad example for the other types of algorithm. We therefore propose and evaluate new and existing combinations of the different types of algorithms. In our experiments, we find that these give an excellent trade-off in running time and performance.

The remainder of our paper is organized as follows. In Section 2 we discuss some related papers that evaluate different algorithms for rank aggregation and related problems. In Section 3 we will discuss the positional, comparison sort, local search and hybrid algorithms that have been proposed in the literature, and we propose ways of combining the different approaches. In Section 4 we show bad examples for the "pure" algorithms, and we show that algorithms that combine different approaches perform much better on these examples. In Section 5 we evaluate the different algorithms on real data sets. We find that by combining different methods we get an excellent trade-off in running time and performance. In particular, we give three recommendations of which combination of methods to use based on whether a user wants to have a very fast, fast or reasonably fast algorithm.

2 Related work

There has been a lot of work in the algorithms community on the rank aggregation problem and the feedback arc set problem in tournaments. Since we are comparing many of these algorithms, we defer the discussion of these works to the discussion of the algorithms we are considering. Here we mention some other studies that compare different algorithms for the rank aggregation problem or related problems.

Dwork et al. [9] propose aggregating the results from different search engines as a way to combat spam and get more robust search results. They investigate some traditional rank aggregation methods such as Footrule aggregation and Borda's method, and they propose algorithms similar to PageRank, where they define a Markov chain on the search results and order the results based on their respective probabilities in the stationary distribution.

Gionis, Mannila, Puolamäki and Ukkonen [13] consider algorithms for a problem closely related to partial rank aggregation. They refer to a partial ranking as a bucket order, and the problem they consider is slightly different: they define the distance between two partial rankings π, σ as the number of pairs i, j such that $\pi(i) < \pi(j)$ and $\sigma(i) > \sigma(j)$ plus one half times the number of pairs that are tied in one of π, σ and not in the other. The goal is to find a partial ranking that minimizes the distance to given input partial rankings.

Coleman and Wirth [5] recently investigated the performance of different algorithms for the feedback arc set problem in tournaments. Although they consider rank aggregation as a special case, the algorithms they consider do not exploit the additional structure that rank aggregation problems have, and they do not consider any positional methods. They also show that some of the algorithms have very poor performance guarantees, but their work is different from ours in two aspects. First of all, the bad examples they give are not instances that could arise from a rank aggregation problem. Secondly, for the randomized algorithms they consider the bad examples given are only bad *conditional* on certain random choices of the algorithm, and *not in expectation* over the choices made by the algorithm. One of the data sets on which we test our algorithms was also used by Coleman and Wirth. Some of the algorithms we consider were considered by Coleman and Wirth as well: local search and Chanas, MergeSort, InsertionSort and QuickSort. However, since their other data sets are not rank aggregation instances, their conclusions are quite different from ours (for instance, QuickSort performed poorly on the data sets which are not rank aggregation instances).

Finally, the problem of aggregating different clusters is closely related. In the consensus clustering problem, the input consists of k clusterings of a set of elements V, and the goal is to find a clustering of V that minimizes the number of pairwise disagreements with the input clusterings, i.e. the number of pairs that are in the same cluster in an input clustering but in different clusters in the output clustering or vice versa. A variation of the QuickSort algorithm also gives an approximation algorithm for consensus clustering [2]. Some papers that investigate the theoretical and practical performance of algorithms for consensus clustering are [14, 15].

3 Rank aggregation algorithms

We assume without loss of generality that the elements are numbered $1, \ldots, n$. We think of a ranking $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ as a list of the elements $\{1, \ldots, n\}$ in order of preference (possibly with ties). We will thus say that a ranking π *prefers* i to j, or ranks i *higher* than j, if $\pi(i) < \pi(j)$. If π is a permutation, we will sometimes write it as $\texttt{list}(\pi) = (\pi^{-1}(1), \pi^{-1}(2), \ldots, \pi^{-1}(n))$.

We assume that the input is provided in the form of k different permutations or partial rankings $\pi_1, \ldots, \pi_k$, and multiplicities $\mu_1, \ldots, \mu_k$ (so that π_ℓ occurs μ_ℓ

times). We let $M = \sum_{\ell=1}^{k} \mu_\ell$. Given such an input, we define an n-by-n matrix w, where $w_{(i,j)} = \sum_{\ell=1}^{k} \frac{\mu_\ell}{M} \mathbf{1}\{\pi_\ell(i) < \pi_\ell(j)\}$.

3.1 Positional algorithms We call an algorithm for full rank aggregation *positional* if it seeks a permutation in which the *position* of each element is "close to the average position" of the element in the input permutations.

3.1.1 Borda The Borda count of an element is its mean position in the input permutations, ie. $\texttt{Borda}(i) = \sum_{\ell=1}^{k} \frac{\mu_\ell}{M} \pi(i)$. The Borda algorithm ranks the elements in order of increasing Borda counts. Coppersmith, Fleischer and Rudra [7] show that the Borda algorithm finds a permutation σ that minimizes $\sum_{i \in V} |\sum_{\ell=1}^{k} \frac{\mu_\ell}{M} \pi_\ell(i) - \sigma(i)|$, i.e. the sum of the distances from the elements' position to their mean positions.

Note that in the corresponding instance to the weighted feedback arc set problem in tournaments, ordering by Borda count is equivalent to ranking the vertices by increasing (weighted) indegree. For partial rank aggregation, we therefore define the Borda count of an element to be its weighted indegree in the corresponding tournament. It was shown in [7] that this is a 5-approximation algorithm for the weighted feedback arc set problem in tournaments if the weights satisfy the probability constraints; hence this holds for full rank aggregation.

3.1.2 Footrule The Spearman's Footrule distance between two permutations π, σ is defined as

$$\mathcal{F}(\pi, \sigma) = \sum_{i=1}^{n} |\pi(i) - \sigma(i)|.$$

It was shown by Diaconis and Graham [8] that

$$\mathcal{K}(\pi, \sigma) \leq \mathcal{F}(\pi, \sigma) \leq 2\mathcal{K}(\pi, \sigma).$$

Hence a permutation σ that minimizes the average Footrule distance $\frac{1}{k} \sum_{\ell=1}^{k} \mathcal{F}(\sigma, \pi_\ell)$ to a given set of permutations $\pi_1, \pi_2, \ldots, \pi_k$ is a 2-approximation for the Kemeny rank aggregation problem. Finding a permutation σ that minimizes $\frac{1}{k} \sum_{\ell=1}^{k} \mathcal{F}(\sigma, \pi_\ell)$ can be done in polynomial time by solving a bipartite matching problem (see Dwork et al. [9]). Dwork et al. also showed that if the median positions of the elements form a permutation, then this permutation is an optimal Footrule aggregation.

To extend Footrule aggregation to partial rank aggregation, we need to define the Footrule distance between a partial ranking and a full ranking. Given a partial ranking π, and an element i, let $\pi_{\min}(i) = \#\{j : \pi(j) < \pi(i)\}$ and let $\pi_{\max}(i) = n - \#\{j : \pi(j) > \pi(i)\}$, in other words, $\pi_{\min}(i)$ and $\pi_{\max}(i)$ are the minimum and maximum position that i could take in a full ranking σ such that $\mathcal{K}(\pi, \sigma) = 0$. We then define the Footrule distance between permutation σ and partial ranking π as

$$\mathcal{F}(\pi, \sigma) = \sum_{i=1}^{n} \min_{p \in [\pi_{\min}(i), \pi_{\max}(i)]} |p - \sigma(i)|$$

We note that our definition of Footrule distance between a permutation and a partial ranking is different from the one proposed in Dwork et al. [9], since they use a different definition of a partial ranking. As in the case of full rank aggregation, we can find an optimal Footrule aggregation for partial rank aggregation by solving a bipartite matching problem.

LEMMA 3.1. *Footrule aggregation is a 2-approximation algorithm for partial rank aggregation.*

Proof. We will show that for a permutation σ and a partial ranking π, $\mathcal{F}(\pi, \sigma) \leq 2\mathcal{K}(\pi, \sigma)$.

Since we know that for a permutation π', $\mathcal{F}(\pi', \sigma) \leq 2\mathcal{K}(\pi', \sigma)$, it suffices to show that there exists a permutation π' such that $\mathcal{K}(\pi', \sigma) = \mathcal{K}(\pi, \sigma)$ and $\mathcal{F}(\pi', \sigma) \geq \mathcal{F}(\pi, \sigma)$.

We let π' be the unique permutation that has $\pi'(i) < \pi'(j)$ if $\pi(i) < \pi(j)$ or if $\pi(i) = \pi(j)$ and $\sigma(i) < \sigma(j)$. Clearly, $\mathcal{K}(\pi', \sigma) = \mathcal{K}(\pi, \sigma)$. On the other hand, $\pi'(i) \in [\pi_{\min}(i), \pi_{\max}(i)]$ for any i, and hence $\mathcal{F}(\pi', \sigma) \geq \mathcal{F}(\pi, \sigma)$. □

3.1.3 Pick-a-Perm The Pick-a-Perm algorithm for full rank aggregation returns an input permutation at random. It is easily shown that this is a 2-approximation algorithm, see for example Ailon et al. [2]. This algorithm has a deterministic variant, where the input permutation that gives the smallest objective value is chosen. We call this the Best-of-k algorithm.

Ailon [1] showed how to generalize the Pick-a-Perm algorithm to an algorithm for partial rank aggregation: We think of a partial ranking as a bucket order, where elements with the same rank are in one bucket. To construct output ranking σ, we start with σ having all elements in the same bucket. We repeatedly choose a partial ranking at random from the input rankings, and order the elements within each bucket of σ according to this partial ranking, until each bucket of σ is of size 1. Ailon shows that this algorithm is also a 2-approximation algorithm and can be derandomized to give a deterministic 2-approximation algorithm.

3.2 Comparison sort algorithms We now describe three comparison sort algorithms. We have a relation $\preceq$ on the elements, where $i \preceq j$ if the majority of the

input rankings has ranked i before j (in other words, if $w_{(i,j)} \geq w_{(j,i)}$). We will say that $i \prec j$ if $i \preceq j$ and $j \not\preceq i$. We assume that the numbering of the elements is used to break ties, so that if both $i \preceq j$ and $j \preceq i$, and $i < j$ then we say that $i \prec j$.

Note that the relation $\prec$ is not transitive, which is exactly Condorcet's paradox. The fact that the relation is not transitive implies that different comparison sort algorithms can produce different rankings.

3.2.1 QuickSort The QuickSort algorithm recursively sorts the elements by choosing a vertex i as pivot, and ordering vertex j to the left of (higher than) i if $j \prec i$, or to the right of i if $i \prec j$ (breaking ties arbitrarily). The algorithm then recurses on the instance induced by the vertices to the left of i and those to the right of i.

Ailon et al. [2] showed that if we define $\preceq$ as above, and choose a pivot uniformly at random, then the permutation returned by the QuickSort algorithm is an expected 2-approximation algorithm.

Van Zuylen and Williamson [20] give a deterministic QuickSort algorithm, in which the "best" pivot is chosen instead of a random pivot, and show that this is also a 2-approximation algorithm. In particular, in a recursive call on $V' \subseteq V$, let $L(i)$ be the set of elements that would be ordered to the left of $i \in V'$ if i is the pivot in this iteration, i.e. $L(i) = \{j \in V' : j \preceq i\}$, and let $R(i)$ be the elements that would be ordered to the right. Then the pairs (ℓ, r) such that $r \preceq \ell$ and $\ell \in L(i), r \in R(i)$ are out of order with respect to the relation $\preceq$. For each element i they compute the ratio

$$\frac{\sum_{\ell \in L(i), r \in R(i)} w_{(r,\ell)}}{\sum_{\ell \in L(i), r \in R(i)} w_{(\ell,r)}} \tag{3.2}$$

and the element for which the ratio is smallest is chosen as pivot.

Compared to Ailon et al.'s QuickSort algorithm, the running time is approximately a factor of n slower, because in each iteration every potential pivot is evaluated. We therefore also considered an "intermediate" algorithm, in which in a recursive call on V' we compute the ratio in (3.2) for $\log(|V'|)$, or a constant number of randomly chosen elements in V', and choose the element that has the smallest ratio among these. To keep the number of results we display manageable, we will only show the result of the algorithm that evaluates $\log(|V'|)$ possible pivots. The results when evaluating 3 to 5 pivots, are comparable in performance and running time.

In Section 5 we denote by QuickSort the original randomized algorithm, DetQuickSort the fully derandomized algorithm, LogQuickSort the algorithm which takes the best among $\log(|V'|)$ pivots.

3.2.2 MergeSort MergeSort recursively sorts the elements by dividing them into two (approximately) equal parts, recursing on each part to obtain two sorted lists, and merging the two lists as follows. We refer to the two sorted lists as List 1 and List 2, and we construct a merged list, List 3. While List 1 and List 2 are not empty, let i be the top element of List 1 and j the top element of List 2. If $i \prec j$, then we remove i from List 2 and add it to the bottom of List 3. Otherwise we move j to the bottom of List 3. Once one of List 1 and List 2 is empty, we add the remainder of the other list to the bottom of List 3.

3.2.3 InsertionSort In the InsertionSort algorithm, we start with an empty list and add the elements one by one to the list. When an element i is added to the list, it is placed in the highest position so that $i \prec j$ for all elements j that are in lower positions than i. We can find i's position by adding i to the bottom of the list, and allowing i to "bubble up": while $i \prec j$ for the element j directly above i in the list, we swap i and j. We note that the Local Kemenization procedure proposed in Dwork et al. [9] is the same as InsertionSort.

3.3 Local search We consider local search algorithms that execute "single vertex moves": Given a permutation π, a single vertex move takes an element i and inserts it into another position if this improves the objective value.

To the authors' knowledge there is no known performance guarantee for a permutation that is locally optimal with respect to single vertex moves. The example given in Coleman and Wirth [5] in which the local optimum is a factor $\Omega(n)$ more expensive than the global optimum applies only to the feedback arc set problem in tournaments, and not to rank aggregation.

There is evidence that single vertex moves are very powerful: they are part of the PTAS for weighted feedback arc set in tournaments with probability constraints by Kenyon-Mathieu and Schudy [16], and have been shown to be successful in other implementation studies [5].

We implemented the local search algorithm as follows: we go through the element *positions* in random order, and when considering position i, we move the element currently in position i to the position which gives the largest improvement to the objective value. We stop if we have considered all positions, and no element has been moved. Otherwise we continue in the same fashion.

We also investigate the algorithm called Chanas in Coleman and Wirth [5], which was proposed by Chanas

and Kobylański [4]. This algorithm is a composition of sort and reverse steps. We start with a random permutation, and repeatedly go through the elements from left to right; if we can improve the objective by moving an element to the left, we do so. Once we cannot make any improvements, we reverse the permutation, and repeat. Chanas and Kobylański show that for any permutation π, and the permutation π' we obtain by reversing π and one sorting pass through the elements, the objective of π' is not more than that of π.

3.4 Hybrid algorithms We now discuss algorithms that we call "hybrid" algorithms: they combine the ideas from positional and comparison based algorithms.

3.4.1 Copeland's method We define the Copeland score of an element i to be the number of elements j such that $i \prec j$. Copeland [6] suggested sorting the elements by their Copeland score.

Note that if we define the majority tournament $G = (V, A)$ to be the directed graph that has a node for every element, and an arc from i to j if $i \prec j$, then Copeland's method sorts the elements by non-increasing indegree. Hence Copeland's method can be seen as Borda's method on the majority tournament.

3.4.2 MC4 Dwork et al. [9] propose Markov chain algorithms for rank aggregation that are similar to PageRank. We consider here the best of these type of algorithms that were considered by Dwork et al.: the MC4 algorithm. One way to think of this algorithm is to think of a random process on the set of elements. The process starts at some element i, and chooses one of the elements, say j, uniformly at random. If a majority of the input rankings prefers j to i, then we move to j, otherwise we stay at i. We then again choose an element at random and move if this element is preferred to the current element by a majority of the input rankings, etc.

This is known as a Markov process, where the transition matrix P has $P(i,j) = \frac{1}{n}$ if a majority of the input rankings prefer j to i, and $P(i,i) = 1 - \sum_{j \neq i} P(i,j)$. Under certain conditions, this process has a unique (up to scalar multiples) limiting distribution x that satisfies $x = xP$, where $x(i)$ gives the fraction of time the process spends at element i. Dwork et al. propose sorting the elements by non-increasing $x(i)$ values.

To ensure that the process has a unique limiting distribution x, we use a "random jump": with probability $\delta > 0$, we will choose a random element and move to this element (regardless of whether this element is preferred to the current element). In our experiments we have used $\delta = \frac{1}{7}$, which is the value of δ that is often chosen in the literature for PageRank implementations.

In addition to calculating x exactly, we also consider the method MC4Approx, in which we start with a random vector y, and sort the elements according to $\hat{x} = yP^n$.

As Dwork et al. point out, the MC4 algorithm is similar in flavor to Copeland's method: the diagonal entries of the transition matrix are exactly $\frac{1}{n}$ times the Copeland scores of the elements.

3.5 Combining different methods We want to further exploit the fact that the rank aggregation problem has these different approaches by combining them into new algorithms. Obviously, we can combine the local search algorithm with any given algorithm, by running the local search procedure on the outcome of the other algorithm. We now outline how we can combine the comparison sort algorithms with the other approaches.

We note that, except for the deterministic QuickSort algorithm of [20], the comparison sort algorithms make random choices, either by choosing an element to pivot on, by choosing how to divide the elements into two groups, or by choosing the order in which to insert the elements. One way to make these choices deterministic is to give a permutation as additional input to the comparison sort algorithms, and let the previously random choice be determined by the permutation instead. Hence we can use the permutation that is output by one algorithm as additional input to a comparison sort algorithm, thus obtaining a deterministic algorithm that hopefully combines the desirable qualities of the individual algorithms.

In the case of InsertionSort, it is clear how a permutation dictates the random choices made by the algorithm, since we can think of the permutation as giving the order in which to insert the elements. We also tested InsertionSort when the elements are inserted in the reverse order of the permutation. We note that the first approach was also suggested by Dwork et al. [9]. The results of the latter are omitted from this extended abstract, because the first approach tended to give better results.

In MergeSort, the algorithm repeatedly divides the elements into two approximately equal parts and recurses on each part. We use a permutation to guide how the algorithm divides the elements. We tried two different approaches here: in MS we divide the elements according to whether they are in odd positions or in even positions. In MS2 we divide the elements according to whether their position is before or after the median position of the elements in the recursive call. Because MS performed better on most instances, we omitted the results of MS2 from this extended abstract.

Finally, we can use an input permutation to determine which element to pivot on in QuickSort. In the algorithm QS we take as pivot the element that is in the median position among the elements in the recursive call.

We will use the outputs of each of the positional methods as well as the output of the Copeland and MC4 algorithm as input into the different comparison sort methods.

3.6 Optimal solution We compare the solutions generated by the heuristics against the optimal value of the following integer program (IP) and its linear programming (LP) relaxation. The optimal value of the IP is equal to the optimal value of the rank aggregation instance.

$$\begin{aligned} \min \quad & \sum_{i,j \in V} w_{(i,j)} x_{(j,i)} + w_{(j,i)} x_{(i,j)} \\ \text{s.t.} \quad & x_{(i,j)} + x_{(j,k)} + x_{(k,i)} \geq 1 \ \ \forall i,j,k \in V \\ & x_{(i,j)} + x_{(j,i)} = 1 \ \ \forall i,j \in V \\ & x_{(i,j)} \in \{0,1\} \end{aligned}$$

The integrality gap of this linear program is at most $\frac{3}{2}$ if the weights w arise from a partial rank aggregation instance [1], and at most $\frac{4}{3}$ if the weights arise from a full rank aggregation instance [2]. To the best of our knowledge, no lower bounds are known. Remarkably, all but three of the instances we used had an integer optimal solution to the LP relaxation. Otherwise, the gap between the optimal integer and optimal fractional solution was less than 0.002%.

Because our instances became too large for the CPLEX solver, we partitioned the instances into subinstances: It is not hard to show that if we can partition V into A, B such that $w_{(i,j)} \geq w_{(j,i)}$ for all $i \in A, j \in B$, then there exists an optimal solution to the integer program that has $x_{(i,j)} = 1$ for all $i \in A, j \in B$. We used this fact to break up the IP for an instance into several IPs that could be solved separately.

We note that we did not attempt to speed up the solution time by using well-known techniques such as warm start or constraint generation techniques. Our main goal in this paper is to compare fast and easily implementable heuristics, that do not require special purpose software.

4 Lower Bounds on Guarantees

In the discussion of the algorithms, we noted that some of them have known upper bounds on their performance guarantees. We now show examples in which the algorithms do not perform that well, thus giving lower bounds on their performance guarantees.

4.1 Positional methods We describe an example that can be turned into a bad example for both the Borda and Footrule algorithms. We have n elements, and our input permutations consist of n different permutations of the elements, $\pi_1, \ldots, \pi_n$, where π_ℓ occurs μ_ℓ times. The permutations are defined as $\texttt{list}(\pi_1) = (1, 2, \ldots, n)$, $\texttt{list}(\pi_2) = (n, 1, \ldots, n-1)$, $\texttt{list}(\pi_3) = (n-1, n, 1, \ldots, n-2)$, etc. up to $\texttt{list}(\pi_n) = (2, \ldots, n, 1)$, or equivalently, $\pi_\ell(i) = (i + \ell - 1) \mod n$.

Note that if $\mu_\ell = 1$ for every ℓ, then the average and median position of each element is the same. This means that the elements are indistinguishable for Borda and Footrule. It is not hard to show that in this case, both Borda and the Footrule method can return any permutation. We will use this example with a small twist to give lower bounds on the performance guarantees for the Borda and Footrule method.

LEMMA 4.1. *The performance guarantee of Borda's method is at least 2.*

Proof. Let m be an arbitrary nonnegative constant. We let $\mu_1 = m, \mu_n = m+n$ and $\mu_\ell = m+n-1$ for $\ell = 2, \ldots, n-1$.

Consider $\texttt{Borda}(i) = \sum_{\ell=1}^{n} \frac{\mu_\ell}{M} \pi_\ell(i)$, the Borda count of element i. In ranking ℓ, the element is in position $(i+\ell-1) \mod n$, hence ranking ℓ contributes $((i+\ell-1) \mod n) \times \frac{\mu_\ell}{M}$ to the indegree of element i.

We therefore get that $M\texttt{Borda}(i)$ equals $im + \sum_{\ell=2}^{n-i+1}(i+\ell-1)(m+n-1) + \sum_{\ell=n-i+2}^{n-1}(i+\ell-1-n)(m+n-1) + (i-1)(m+n)$. It is easy to verify that $M\left(\texttt{Borda}(i) - \texttt{Borda}(i+1)\right) = n-2 > 0$ for all $i = 1, \ldots, n-1$, hence Borda's method will order the elements in reverse order.

We take $m = n^2$ so that $\frac{\mu_\ell}{M} = \frac{1}{n} + O(\frac{1}{n^2})$. We compare the cost of Borda's ranking to the cost of the identity. In the Borda ranking every pair is reversed, hence the cost of the identity plus the cost of the Borda ranking should be equal to:

$$\begin{aligned} \sum_{\ell=1}^{n} \frac{\mu_\ell}{M} \frac{n(n-1)}{2} &= \sum_{\ell=1}^{n-1} \left(\frac{1}{n} + O\Big(\frac{1}{n^2}\Big)\right) \frac{n(n-1)}{2} \\ &= \frac{n^2}{2} + O(n). \end{aligned} \tag{4.3}$$

We now find the cost of the identity. Note that in π_1, no pairs are out of order with respect to the identity. In π_2, all pairs involving element n are out of order. In π_3, all pairs $\{i, j\}$ such that $i \in \{1, \ldots, n-2\}, j \in \{n-1, n\}$, are out of order. So in general, in the ℓ-th ranking, we have $(n-\ell+1)(\ell-1)$ pairs that are out of order. Hence the cost of the identity is

$$\begin{aligned} \sum_{\ell=1}^{n} \frac{\mu_\ell}{M}(n-\ell+1)(\ell-1) &= \sum_{\ell=0}^{n} \left(\frac{1}{n} + O\Big(\frac{1}{n^2}\Big)\right)(n-\ell)\ell \\ &= \frac{n^2}{6} + O(n). \end{aligned}$$

We subtract this from (4.3) to find that the objective value of the Borda ranking is $\frac{n^2}{3} + O(n)$, and hence the ratio of the cost for the Borda ranking compared to the identity tends to 2 if we let $m = n$ and $n \to \infty$. □

LEMMA 4.2. *The performance guarantee of the Footrule method is at least 2.*

Proof. We again use the permutations $\pi_1, \ldots, \pi_n$ from the proof of Lemma 4.1, where $\pi_\ell(i) = (i+\ell-1) \mod n$. It is easy to verify that $\sum_{\ell=1}^{n} \mathcal{F}(\sigma, \pi_\ell)$ is a constant that does not depend on the permutation σ. We therefore have one additional input permutation π_{n+1} which is the reverse of the identity, i.e. $\pi_{n+1}(i) = n-i+1$. Then if we set μ_ℓ to the same value for $\ell = 1, \ldots, n$ and $\mu_{n+1} > 0$, it is easy to see that the Footrule distance to the input permutations is minimized by outputting π_{n+1}.

We now let $\mu_\ell = n$ for $\ell = 1, \ldots, n$ and $\mu_{n+1} = 1$, so that $\frac{\mu_\ell}{M} = \frac{1}{n} + O(\frac{1}{n^2})$ for $\ell = 1, \ldots n$ and $\frac{\mu_{n+1}}{M} = 0 + O(\frac{1}{n^2})$. Then we know from the proof of Lemma 4.1 that the ratio of the objective value of σ compared to that of the identity tends to 2 if we let $n \to \infty$. □

The Pick-a-Perm algorithm does do well on the example given above; however, it is not hard to construct a bad example for Pick-a-Perm and Best-of-k.

LEMMA 4.3. *The performance guarantee of Pick-a-Perm and Best-of-k is at least 2.*

Proof. Let $\pi_1, \ldots, \pi_{n-1}$ be the input permutations, where $\pi_i(j) = j$ if $j \neq i, i+1$ and $\pi_i(i) = i+1, \pi_i(i+1) = i$. Then the objective value of any of the input permutations is $\frac{2(n-2)}{n-1}$, but the objective value of the identity is 1. □

4.2 Comparison sort methods With the exception of the deterministic QuickSort algorithm of Van Zuylen and Williamson [20], the comparison sort methods all need to make random choices: in (randomized) QuickSort the pivot is chosen uniformly at random from the elements, in MergeSort the elements are randomly divided into two equal sized groups, and in InsertionSort the elements are inserted in random order.

It is not difficult to divise examples together with a particular random choice for which these algorithms perform very badly, so that the algorithm's solution's objective value could be a factor $\Omega(n)$ higher than the optimal value. However, if with high probability the algorithm performs very well, one could just run the algorithm a few times, and take the best solution found.

In the case of InsertionSort, we show a much stronger result: there exists an example where, if inserting the elements in random order, the *expected* performance guarantee of the InsertionSort algorithm is $\Omega(n)$.

LEMMA 4.4. *The expected performance guarantee of InsertionSort is* $\Omega(n)$.

Proof. We consider an example with $2n+1$ elements, numbered 0 to $2n$. There are three input rankings, given by π_1, π_2, π_3, where $\texttt{list}(\pi_1)$ is the identity, $\texttt{list}(\pi_2) = (1, 2, \ldots, 2n, 0)$ and $\texttt{list}(\pi_3) = (n+1, n+2, \ldots, 2n, 0, 1, \ldots, n)$, and $\mu_1 = m, \mu_2 = m, \mu_3 = 1$ for some large constant m. We call elements $1, \ldots, n$ red elements, and $n+1, \ldots, 2n$ blue elements. InsertionSort starts with an empty list, considers the element in random order and inserts the element in the highest position so that $i \prec j$ for all elements j that are in lower positions than i.

Note that at the moment when 0 is considered, the current list has the elements considered thus far in lexicographical order. If some blue element has been considered before 0, then a blue element is at the bottom of the list, and 0 is inserted at the bottom. If a blue element is inserted next, it will be inserted in its correct position among the blue elements, but if a red element is inserted next, then it is inserted below 0. After the first red element that follows 0, all subsequent red and blue elements are inserted below 0.

We will let $\mathcal{B}$ be blue elements that are considered before element 0, and we let $\mathcal{R}$ be the red elements considered after element 0, and let B, R be the size of $\mathcal{B}$ and $\mathcal{R}$ respectively. From the previous discussion, InsertionSort ranks the elements in $\mathcal{B}$ above the elements in $\mathcal{R}$, and the cost of the permutation returned by InsertionSort is thus at least $\frac{2m}{2m+1} R \times B$.

We note that if the elements are considered in random order then B and R are independent random variables, and B and $n-B$, R and $n-R$ are identically distributed, so the probability that both B and R are at least $\frac{n}{2}$ is at least $\frac{1}{4}$. Hence $\frac{2m}{2m+1} R \times B$ is at least $\frac{2m}{2m+1} \frac{n^2}{4}$ with probability $\frac{1}{4}$, and the expectation of $\frac{2m}{2m+1} R \times B$ is thus at least $\frac{2m}{2m+1} \frac{n^2}{16}$. On the other hand, the objective value for the permutations π_1, π_2 is $\frac{m+1}{2m+1} 2n + \frac{1}{2m+1} n^2$. □

Note that the example in the proof of Lemma 4.4 gives a bad example for QuickSort if we choose 0 as the first pivot, in which case the algorithm returns the solution $(n+1, \ldots, 2n, 0, 1, \ldots, n)$, which is a factor $\Omega(n)$ from optimal. However, one can show that the expected ratio of the objective value of the QuickSort solution and the optimal value for this particular example is not more than $\frac{4}{3}$. Similarly, if we use MergeSort on this example, and in the first recursive call, we split the elements into $\{0, 1, \ldots, n\}$ and $\{n+1, \ldots, 2n\}$, then MergeSort returns

$(n+1,\ldots,2n,0,1,\ldots,n)$ but the expected ratio of the MergeSort solution's objective value for this instance and the optimal value is constant.

4.3 Local search Coleman and Wirth [5] give a nice example which shows that for feedback arc set in tournaments, there exist local optima with respect to single vertex moves that are a factor $\Omega(n)$ more expensive than the optimum. For rank aggregation, a similar example is not nearly as bad: because we need to ensure that the weights on the arcs obey the triangle inequality, a large constant is added to the objective value of any feasible solution for their example. The lower bound on the approximation ratio therefore drops from $\Omega(n)$ to only $\frac{5}{4}$.

We are not aware of examples on which the performance of the local search algorithm is worse than $\frac{5}{4}$.

4.4 Hybrid algorithms In the bad example for Borda's method in the previous section, both Copeland and the MC4 algorithm are not able to distinguish between the elements and these algorithms could thus return any permutation depending on how the algorithms break ties. We can adapt the example slightly, to make an example for which the MC4 algorithm returns a solution that costs a factor $\frac{3}{2}$ more than the optimal value.

LEMMA 4.5. *The performance guarantee of the MC4 algorithm is at least* $\frac{3}{2}$.

Proof. We take n even, and we consider the same input rankings as in the bad example for the Borda and Footrule algorithms: We have n elements, and n permutations $\pi_1,\ldots,\pi_n$, where $\texttt{list}(\pi_\ell) = (n-\ell+2, n-\ell+3,\ldots,n,1,2,\ldots,n-\ell+1)$. The weights are given by $\mu_1 = m-1, \mu_\ell = m$ for $2 \le \ell \le n$.

Consider two elements i,j, where $j > i$. Then i is ranked before j in rankings 1 up to $n-j+1$, and in rankings $n-i+2$ up to n. The number of voters that ranked i before j is thus $\sum_{\ell=1}^{n-j+1}\mu_\ell + \sum_{\ell=n-i+2}^{n}\mu_\ell = (n-(j-i))m-1$ and the number of voters that ranked j before i is $(j-i)m$. We get that for $i<j$ a majority of voters prefers j to i if $(j-i)m \ge nm-(j-i)m-1$, i.e. if $j \ge i+\frac{n}{2}-\frac{1}{2m}$ and a majority prefers i to j if $j \le i+\frac{n}{2}-\frac{1}{2m}$. If we take $m \ge 1$, then for any i,j there is a strict majority that prefers one of the two elements.

Hence the transition matrix corresponding to our instance is given by

$$P(i,j) = \begin{cases} \frac{1}{n} & \text{if } j \ge i+\frac{n}{2} \text{ or } i-\frac{n}{2} < j < i \\ \frac{1}{2} & \text{if } i=j \le \frac{n}{2} \\ \frac{1}{2}+\frac{1}{n} & \text{if } i=j > \frac{n}{2} \\ 0 & \text{otherwise.} \end{cases}$$

We show that the solution x to $x = xP$ has $x_{\frac{n}{2}+1} > x_{\frac{n}{2}+2} > \ldots > x_n > x_{\frac{n}{2}} > x_{\frac{n}{2}-1} > \ldots > x_1$, and that the MC4 algorithm thus outputs the ranking $\frac{n}{2}+1, \frac{n}{2}+2,\ldots,n,\frac{n}{2},\frac{n}{2}-1,\ldots,1$: Setting $x = xP$ gives

$$\begin{aligned} x_i &= \frac{1}{2}x_i + \frac{1}{n}x_{i+1} + \ldots + \frac{1}{n}x_{n/2-1+i}, \\ x_{n/2+i} &= \frac{1}{n}x_1 + \ldots + \frac{1}{n}x_i + \Big(\frac{1}{2}+\frac{1}{n}\Big)x_{n/2+i} \\ &\quad + \frac{1}{n}x_{n/2+i+1} + \ldots + \frac{1}{n}x_n, \end{aligned}$$

for $i = 1,2,\ldots,n/2$.

Taking differences, we get:

$$\frac{1}{2}\big(x_i - x_{i+1}\big) = \frac{1}{n}\big(x_{i+1} - x_{n/2+i}\big), \tag{4.4}$$

$$\frac{1}{2}\big(x_{n/2} - x_{n/2+1}\big) = \frac{1}{n}\big(-x_1 - x_n\big), \tag{4.5}$$

$$\Big(\frac{1}{2}-\frac{1}{n}\Big)\big(x_{n/2+i} - x_{n/2+i+1}\big) = \frac{1}{n}\big(-x_{i+1} + x_{n/2+i+1}\big), \tag{4.6}$$

for $i = 1,2,\ldots,n/2-1$.

Adding (4.4) and (4.6) gives

$$\begin{aligned} \frac{1}{2}\big(x_i - x_{i+1}\big) &= \frac{1}{n}\big(x_{n/2+i+1} - x_{n/2+i}\big) \\ &\quad - \Big(\frac{1}{2}-\frac{1}{n}\Big)\big(x_{n/2+i} - x_{n/2+i+1}\big) \\ &= \frac{1}{2}\big(x_{n/2+i+1} - x_{n/2+i}\big). \end{aligned} \tag{4.7}$$

CLAIM 1. *$x_i < x_{i+1}$ implies $x_{i+1} < x_{i+2}$ and $x_{n/2+i} > x_{n/2+i+1}$ for $i = 1,2,\ldots,n/2-2$.*

Proof. By contradiction. Assume $x_i < x_{i+1}$ and $x_{i+1} \ge x_{i+2}$. The latter inequality together with (4.4) implies $x_{i+2} \ge x_{n/2+i+1}$. We thus have $x_{i+1} \ge x_{i+2} \ge x_{n/2+i+1}$, which together with (4.6) now implies $x_{n/2+i} \le x_{n/2+i+1}$ and hence $x_{n/2+i} \le x_{n/2+i+1} \le x_{i+2} \le x_{i+1}$. However, the former inequality together with (4.4) implies $x_{n/2+i} > x_{i+1}$. The other part of the claim follows from (4.7). ⋄

CLAIM 2. *The equilibrium distribution x has $x_1 < x_2 < \ldots < x_{n/2} < x_n < x_{n-1} < \ldots < x_{n/2+1}$.*

Proof. (case 1) We will start by assuming that $x_1 < x_2$. We get $x_1 < x_2 < \ldots < x_{n/2}$ and $x_n < x_{n-1} < \ldots < x_{n/2+1}$. Equation (4.6) with $i = n/2-1$ gives us the remaining inequality $x_{n/2} < x_n$. Note that (4.5) indeed holds.
(case 2) Assume $x_1 > x_2$. Then we get in a similar way $x_1 > x_2 > \ldots > x_{n/2}$ and $x_n > x_{n-1} > \ldots > x_{n/2+1}$. (4.4) with $i = n/2-1$ gives the remaining inequality $x_{n/2} > x_n$. But now (4.5) is violated.

(case 3) Assume $x_1 = x_2$. Then we get in a similar way $x_1 = x_2 = \ldots = x_{n/2}$ and $x_n = x_{n-1} = \ldots = x_{n/2+1}$. (4.4) with $i = n/2-1$ gives $x_{n/2} = x_n$. (4.5) now tells us $x_i = 0$ for all i, so this is not a distribution. $\diamond$

We now compare the objective values of the MC4 solution and the identity. We take $m = n$ so that $\frac{\mu_\ell}{M} = \frac{1}{n} + O(\frac{1}{n^2})$ for $\ell = 1, \ldots, n$.

The only pairs that are in the same order in the MC4 solution as in the identity are pairs $\{i, j\}$ where $i, j > \frac{n}{2}$. We consider the cost for *these pairs* in the identity. In π_1, all these pairs are ordered lexicographically, in π_2 the pairs with $j = n$ are out of order, so there are $\frac{n}{2} - 1$ out of order. In π_3, the pairs with $\frac{n}{2} < i < n-1$ and $j \geq n-1$ are out of order, and there are $2(\frac{n}{2} - 2)$ of these. In general, in π_ℓ, for $\ell \leq \frac{n}{2}$, there are $(\ell-1)(\frac{n}{2} - (\ell-1))$ pairs $\{i, j\}$ with $i, j > \frac{n}{2}$ that are out of order with respect to the identity. For $\ell > \frac{n}{2}$, all pairs $\{i, j\}$ with $i, j > \frac{n}{2}$ are in lexicographical order in π_ℓ. Hence the cost incurred by the identity (and hence also by the MC4 solution) for pairs $\{i, j\}$ with $i, j > \frac{n}{2}$ is $\sum_{\ell=1}^{n/2} \frac{\mu_\ell}{M}(\ell-1)\left(\frac{n}{2} - (\ell-1)\right) =$

$$\begin{aligned}\sum_{\ell=0}^{n/2-1} \frac{\mu_\ell}{M}\ell\left(\frac{n}{2} - \ell\right) &= \sum_{\ell=0}^{n/2-1}\left(\frac{1}{n} + O\Big(\frac{1}{n^2}\Big)\right)\ell\left(\frac{n}{2} - \ell\right) \\ &= \frac{1}{n}\frac{1}{6}\left(\frac{n}{2}\right)^3 + O(n) = \frac{n^2}{48} + O(n).\end{aligned}$$

From the proof of Lemma 4.1 we know that the total cost of the identity is $\frac{1}{6}n^2 + O(n)$. Hence the cost incurred by the identity *for all other pairs* (i.e. pairs were at least one of i, j is at most $\frac{n}{2}$) is $\frac{7}{48}n^2 + O(n)$. Now, note that there are $\binom{n}{2} - \binom{n/2}{2} = \frac{3}{8}n^2 - \frac{1}{4}n$ pairs $\{i, j\}$ such that at least one of i, j is at most $\frac{n}{2}$. Since these pairs are in opposite order in the identity and the MC4 solution, the sum of the cost incurred by the two solutions for these pairs is $(\frac{3}{8}n^2 - \frac{1}{4}n)$. Therefore the cost for these pairs must be $(\frac{3}{8}n^2 - \frac{1}{4}n) - \left(\frac{7}{48}n^2 + O(n)\right) = \frac{11}{48}n^2 + O(n)$ in the MC4 solution, and the total cost of the MC4 solution is thus $\frac{1}{48}n^2 + \frac{11}{48}n^2 + O(n) = \frac{1}{4}n^2 + O(n)$. Since the objective value of the identity is $\frac{1}{6}n^2 + O(n)$, we get the result by letting $n \to \infty$. $\square$

We conclude this section by noting that none of the bad examples we considered is simultaneously a bad example for a positional method and a comparison sort method. One can verify that any of the three comparison sort algorithms performed on the output of Borda's method or Footrule aggregation for their respective bad examples will return one of the input permutations, which all have objective value very close to optimal. In the case of the bad example for Pick-a-Perm, the comparison function $\prec$ is transitive, hence the identity is returned by any comparison sort algorithm. On the other hand, our bad example for InsertionSort is not a bad example for the positional methods. We do note however, that the hybrid method BordaQS performs very badly on this example: the result of Borda's method has element 0 in the median position, and hence QS will pivot on this element first, thus returning the solution $(n+1, n+2, \ldots, 2n, 0, 1, \ldots, n)$ which is a factor $\Omega(n)$ more expensive than the optimal solution.

5 Evaluation

5.1 Description of data sets

5.1.1 Web search data We extracted search results from Ask, Google, MSN Live Search and Yahoo! using the default settings of each of these search engines. The queries we used for our experiment are the same 37 queries that were used by Dwork et al. [9]:

> affirmative action, alcoholism, amusement parks, architecture, bicycling, blues, cheese, citrus groves, classical guitar, computer vision, cruises, Death Valley, field hockey, gardening, graphic design, Gulf war, HIV, java, Lipari, lyme disease, mutual funds, National parks, parallel architecture, Penelope Fitzgerald, recycling cans, rock climbing, San Francisco, Shakespeare, stamp collecting, sushi, table tennis, telecommuting, Thailand tourism, vintage cars, volcano, zen buddhism, and Zener.

As in the experiments of Dwork et al. we say that two pages are identical if their URLs are identical (up to some canonical form); we do not use the content of page to determine if two results are identical. We extracted the top-100 results from each search engine. On average, a single query resulted in 283 different pages. We assumed that all pages for a query that are returned by some search engine, but that are not in the top-100 of a particular search engine, are ranked at position 101 by that particular engine. The average number of results per query was 283, with a standard deviation of 23.4.

5.1.2 Web Communities data set We used the Web Communities data set that was used by Coleman and Wirth [5] in their implementation study. We were not able to obtain the input rankings, but only the matrix w where $w_{(i,j)}$ is the fraction of the input rankings that prefer i to j. For this reason, the only positional method we could evaluate on this data set is Borda's method.

The Web Communities data set was obtained from

9 full rankings of 25 million documents [18]. From this data, 50 different inputs to the full rank aggregation problem were generated by choosing 50 samples of 100 documents each, and letting the input rankings be the induced 9 rankings on the 100 documents.

5.2 Results We implemented each of the algorithms in MATLAB. For the Footrule method, we implemented the Hungarian algorithm for finding a minimum cost bipartite matching, as described in Lawler [17]. The implementation in MATLAB means that matrix multiplications and eigenvector computations are very fast. This may skew our results somewhat in favor of the MC4 and MC4Approx algorithms.

We found the optimal solution to the integer program and its linear programming relaxation using CPLEX. The average solution time to find the LP optimum was 10.6 seconds for the Web Search data set, and 2.8 seconds for the Web Communities data set. Although the LP relaxation is known to have a small integrality gap, a first remarkable outcome of our experiments is the fact that the LP relaxation had an integer optimum for all instances in the Web Communities data set, and for all but 3 instances (corresponding to the queries "amusement parks", "mutual funds" and "Shakespeare") in the Web Search data set. In addition, the largest gap between the optimal objective values of the integer and linear program was only 0.002%, i.e. an integrality gap of 1.00002.

For the three instances for which the LP relaxation did not have an integer optimum, two of them solved in approximately 30 seconds. The instance corresponding to the query "Shakespeare" proved the most difficult: it took CPLEX 78 minutes to find the optimal integer solution. The LP relaxation of this instance was slow to solve as well and took approximately 90 seconds.

However, as we will see, the best heuristics we study here find solutions that cost less than 0.03% more than optimal in just a fraction of the time needed to solve the linear or integer program.

For the randomized algorithms, we took the average objective value over 500 runs. For each algorithm, and for each instance, we computed the "gap": the percentage by which the algorithm's solution value was higher than the optimum. In Table 1 in the Appendix, we give the average CPU time and average gap for the Web Search data set and the Web Communities data set. For comparison, we also included these values for a randomly generated permutation.

In Figures 1 and 2 we display the CPU time versus the percentage by which the solution value found was higher than the optimum for all the algorithms we considered (including the combinations of each algorithm with a local search clean-up). We use different symbols to show the different classes of algorithms: A ▲ denotes a positional algorithm (Borda, Footrule, Pick-a-Perm and Best-of-k), a ■ denotes a comparison sort algorithm (QuickSort, LogQuickSort, DetQuickSort, MergeSort and InsertionSort), a ◆ denotes a hybrid method (Copeland, MC4, MC4Approx), a ★ denotes a local search algorithm (either only single-vertex moves, or the Chanas algorithm, started from a random permutation), a ● denotes a combination of a positional or hybrid method with a comparison sort method, and a ◁ denotes one of the previously mentioned algorithms followed by local search with single-vertex moves.

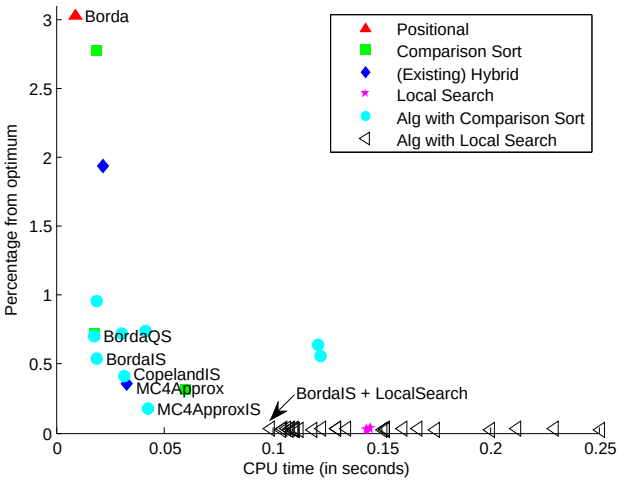

Figure 1: CPU time versus performance of the algorithms on the Web Search data.

The name of an algorithm appears in the graph if no algorithm with smaller running time performs better. Hence these graphs allow one to read off the best algorithm for a given computational budget.

We start by considering the different classes of algorithms separately.

- ▲ **Positional methods:** Because we did not have the input rankings for the Web Communities data set, we were not able to run the Pick-a-Perm, Best-of-k and Footrule algorithms on the Web Communities data set. On the Web Search data set, Borda's algorithm is a clear winner among the 4 positional methods: it finds a solution within 3.03% from the lower bound in approximately 0.01 seconds. Except for Pick-a-Perm, which has approximately the same running time as Borda's algorithm, the other methods are both slower and give worse results. We note that our implementation of the Hungarian algorithm to find the Footrule solution was very slow. Based on the fact that the re-

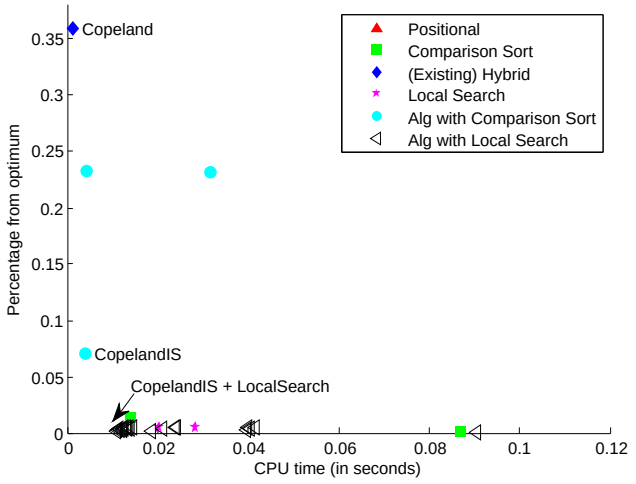

Figure 2: CPU time versus performance of the algorithms on the Web Communities data.

sulting solution is not very good, it does not seem worthwhile to investigate improvements to this algorithm.

■ **Comparison sort methods:** The QuickSort algorithm gives good results (with a gap of less than 1%) for both data sets. The deterministic QuickSort algorithm performs even better than this, but has the drawback of being quite slow. LogQuickSort is approximately 5 times slower than QuickSort, but improves the outcome considerably, especially on the Web Communities data set. The other two comparison sort algorithms are surprisingly unrobust: MergeSort performs well on the WebCommunities data set, but poorly on the Web Search data set, and InsertionSort performs reasonably well on the Web Search data set, but very poorly on the Web Communities data set. Hence the only reasonable comparison sort algorithms (if not used in combination with another algorithm) in our experimental study are those based on QuickSort.

◆ **Hybrid algorithms:** Since the MC4Approx algorithm outputs the same result as the MC4 algorithm, and is much faster than MC4, it suffices to consider only MC4Approx and Copeland. These two algorithms had similar (and very good) performance: The running times were approximately the same for MC4Approx and Copeland, and both algorithms returned solutions that were close to optimal. The performance of the Copeland algorithm was somewhat better on the Web Communities data set (0.36% vs. 0.90% for MC4Approx) but worse on the Web Search data set (1.93% vs. 0.35% for MC4Approx).

● **Combinations with comparison sort algorithms:** Whereas InsertionSort did not perform well on its own, it gave the best improvement when comparing different combinations of a (positional or hybrid) algorithm with a comparison sort algorithm. The results given by a combination of a positional or hybrid algorithm with QuickSort are very good as well, but do not improve on the results of running QuickSort alone. Running MergeSort on the result of another algorithm does not seem to be advisable, as it often gives a *worse* result.

★,◁ **Local search methods:** The single-vertex moves are extremely powerful on our instances. Regardless of the permutation that we started with, using local search we found a permutation that is within 0.03% for the Web Search data set and within 0.005% for the Web Communities data set. However, the running time did depend on the starting permutation. Local search when starting with a random permutation took considerably longer than when starting with a "good" permutation (e.g. the result of one of the other algorithms). The Chanas algorithm, which combines single-vertex moves and reversals of the permutation, came out as the strongest algorithm in the implementation by Coleman and Wirth [5]. In our experiments, the Chanas algorithm is also a very good algorithm, but it does not significantly improve on using only single-vertex moves. Much faster algorithms that are as good as the Chanas algorithm are obtained by starting with a fast heuristic such as Borda, QuickSort, Copeland, or MC4Approx and running a local search procedure on the permutation output by the heuristic.

The fastest algorithms overall in our experiments are the positional methods Pick-a-Perm and Borda. Especially Borda's algorithm, which has the additional advantage of being extremely simple, gives quite reasonable results. However we can do much better by using Copeland's method, which is only slightly more complicated and almost as fast as Borda's algorithm. Other good algorithms that are very fast are QuickSort and MC4Approx. In our experiments, Copeland, QuickSort and MC4Approx all gave results within 2% of optimal in less than 33 milliseconds on the Web Search data set, and less than 3 milliseconds on the Web Communities data set.

Based on our experiments, the best comparison sort method to use as a "second step" on top of another algorithm is InsertionSort. Combining InsertionSort with

Algorithm	WebSearch				Web Communities			
	No Local Search		With Local Search		No Local Search		With Local Search	
	% gap	time	% gap	time	% gap	time	% gap	time
▲ Pick-a-Perm	11.42	0.009	0.03	0.130				
▲ Best-of-k	5.88	0.110	0.03	0.230				
▲ **Borda**	**3.03**	**0.009**	**0.02**	**0.104**	**15.27**	**0.000**	**0.004**	**0.0212**
▲ Footrule	10.97	8.129	0.02	8.257				
■ MergeSort	16.30	0.027	0.03	0.152	1.19	0.004	0.005	0.0131
■ InsertionSort	2.77	0.019	0.03	0.112	68.34	0.003	0.005	0.0239
■ **QuickSort**	**0.72**	**0.018**	**0.03**	**0.108**	**0.67**	**0.003**	**0.004**	**0.0115**
■ DetQuickSort	0.16	2.912	0.02	2.971	0.00	0.087	0.001	0.0907
■ LogQuickSort	0.31	0.060	0.03	0.129	0.01	0.014	0.002	0.0186
★ randomPerm	67.51	0.009	0.02	0.143	188.37	0.000	0.005	0.0284
★ Chanas	0.03	0.145	0.03	0.160	0.01	0.020	0.005	0.0241
◆ **Copeland**	**1.93**	**0.021**	**0.03**	**0.118**	**0.36**	**0.001**	**0.003**	**0.0118**
◆ MC4	0.35	0.471	0.02	0.548	0.90	0.029	0.004	0.0405
◆ **MC4Approx**	**0.35**	**0.033**	**0.02**	**0.110**	**0.90**	**0.002**	**0.004**	**0.0133**
● Pick-a-PermIS	0.96	0.019	0.03	0.105				
● Pick-a-PermMS	10.41	0.028	0.03	0.152				
● Pick-a-PermQS	0.70	0.018	0.03	0.110				
● Best-of-kIS	0.55	0.122	0.03	0.200				
● Best-of-kMS	9.35	0.131	0.03	0.251				
● Best-of-kQS	0.64	0.121	0.03	0.212				
● **BordaIS**	**0.53**	**0.019**	**0.03**	**0.099**	**0.63**	**0.003**	**0.004**	**0.0138**
● BordaMS	15.27	0.028	0.03	0.151	1.24	0.004	0.004	0.0125
● BordaQS	0.70	0.018	0.03	0.106	0.99	0.003	0.003	0.0112
● FootruleIS	0.48	8.139	0.02	8.214				
● FootruleMS	16.58	8.147	0.03	8.276				
● FootruleQS	0.68	8.138	0.03	8.229				
● **CopelandIS**	**0.42**	**0.031**	**0.03**	**0.108**	**0.07**	**0.004**	**0.002**	**0.0110**
● CopelandMS	12.28	0.041	0.03	0.167	0.75	0.005	0.005	0.0138
● CopelandQS	0.72	0.030	0.03	0.122	0.43	0.003	0.004	0.0124
● MC4IS	0.18	0.480	0.03	0.548	0.23	0.032	0.005	0.0399
● MC4MS	15.28	0.489	0.02	0.613	0.94	0.032	0.005	0.0417
● MC4QS	0.74	0.479	0.03	0.572	0.41	0.031	0.003	0.0396
● **MC4ApproxIS**	**0.18**	**0.043**	**0.03**	**0.110**	**0.23**	**0.004**	**0.005**	**0.0127**
● MC4ApproxMS	15.28	0.051	0.02	0.175	0.95	0.005	0.005	0.0145
● MC4ApproxQS	0.74	0.041	0.03	0.134	0.41	0.004	0.003	0.0124

Table 1: The results of the algorithms on the data sets. "% gap" gives the objective value of the algorithm's solution minus the optimum divided by the optimum. Time gives the CPU time in seconds. The algorithms printed in boldface are recommended for a combination of ease of implementation, speed and quality.

Borda, Copeland, or MC4Approx increased the running time by not more than 10 milliseconds, and always improved the result significantly: at worst basically halving the gap between the solution found and the optimum, and often decreasing the gap by a factor 5.

Finally, in our experiments local search was a sure-fire way of finding a solution extremely close to optimal. Running local search on the output of another algorithm takes much longer than InsertionSort, but in our experiments the resulting permutation was never more than 0.03% from optimal.

The practitioner thus has a choice of algorithms for finding good solutions to the rank aggregation problem. Based on ease of implementation, we recommend using Borda for a very fast algorithm, Copeland, or a combination of Borda or Copeland with InsertionSort, for a fast algorithm that gives better results, and finally any of the four previous possibilities followed by local search for an algorithm that is still pretty fast and gives results very close to optimal.

6 Conclusion and future work

We considered positional, comparison sort and local search algorithm and algorithms that combine these different approaches. There is some theoretical indication that this would yield improved algorithms, and we find in our evaluation that hybrid methods indeed give an excellent trade-off of CPU time and performance. Based on our experimental research, we gave three recommendations for which (combination of) algorithms to use, depending on how fast you want to get the result.

Our experimental results raise some theoretical questions. First of all, no approximation guarantees are known for some of the best methods in our experiments, such as local search and MC4. But more importantly, even though the rank aggregation problem seems to be easier than the feedback arc set problem in tournaments, there is a PTAS for the latter and not for the former. Another interesting finding is the fact that the LP relaxations almost always had optimal integer solutions, and that the integrality gap was extremely small in the remaining cases. It is known that the integrality gap is not more than $\frac{3}{2}$ for any partial rank aggregation instance and not more than $\frac{4}{3}$ in the case of full rank aggregation, but it would be very interesting to have lower bounds on the integrality gap.

7 Acknowledgements

We would like to thank Emmanuel Sharef for help in extracting the search results from the four search engines, Tom Coleman, Laurence Park and Tony Wirth for sharing their data sets with us, and David Williamson for helpful discussions. We would also like to thank the anonymous referees for their comments.

References

[1] N. Ailon. Aggregation of partial rankings, p-ratings and top-m lists. In *SODA '07: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 415–424. SIAM, 2007.

[2] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: ranking and clustering. In *STOC '05: Proceedings of the 37th Annual ACM Symposium on Theory of Computing*, pages 684–693. ACM, 2005.

[3] J. Borda. *Memoire sur les elections au scrutin*. Histoire de l'Academie Royal des Sciences, 1781.

[4] S. Chanas and P. Kobylanski. A new heuristic algorithm solving the linear ordering problem. *Comput. Optim. Appl.*, 6(2):191–205, 1996.

[5] T. Coleman and A. Wirth. Ranking tournaments: Local search and a new algorithm. In *ALENEX '08: Proceedings of the Workshop on Algorithm Engineering and Experiments*. SIAM, 2008.

[6] A. Copeland. A 'reasonable' social welfare function. Seminar on Applications of Mathematics to Social Sciences, University of Michigan, USA, 1951.

[7] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *SODA '06: Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 776–782. ACM, 2006.

[8] P. Diaconis and R. L. Graham. Spearman's footrule as a measure of disarray. *J. Roy. Statist. Soc. Ser. B*, 39(2):262–268, 1977.

[9] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW '01: Proceedings of the 10th International Conference on World Wide Web*, pages 613–622. ACM, 2001.

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

[11] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.

[12] R. Fagin, R. Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 301–312. ACM, 2003.

[13] A. Gionis, H. Mannila, K. Puolamäki, and A. Ukkonen. Algorithms for discovering bucket orders from data. In *KDD '06:Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2006.

[14] A. Gionis, H. Mannila, and P. Tsaparas. Clustering aggregation. *ACM Trans. Knowl. Discov. Data*, 1(1):4, 2007.

[15] A. Goder and V. Filkov. Consensus clustering algorithms: Comparison and refinement. In *ALENEX '08: Proceedings of the Workshop on Algorithm Engineering and Experiments*. SIAM, 2008.

[16] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *STOC '07: Proceedings of the 39th Annual ACM Symposium on Theory of Computing*, pages 95–103. ACM, 2007.

[17] E. L. Lawler. *Combinatorial optimization: networks and matroids*. Holt, Rinehart and Winston, New York, 1976.

[18] L. A. F. Park and K. Ramamohanarao. Mining web multi-resolution community-based popularity for information retrieval. In *CIKM '07: Proceedings of the 2007 ACM Conference on Information and Knowledge Management*, pages 545–552, November 2007.

[19] A. van Zuylen, R. Hegde, K. Jain, and D. P. Williamson. Deterministic pivoting algorithms for constrained ranking and clustering problems. In *SODA '07: Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 405–414. SIAM, 2007.

[20] A. van Zuylen and D. P. Williamson. Deterministic algorithms for rank aggregation and other ranking and clustering problems. In *WAOA '07: Proceedings of the 5th Workshop on Approximation and Online Algorithms (WAOA '07)*, Lecture Notes in Computer Science. Springer, 2007.

[21] H. P. Young. Condorcet's theory of voting. *Math. Inform. Sci. Humaines*, (111):45–59, 1990.

[22] H. P. Young and A. Levenglick. A consistent extension of Condorcet's election principle. *SIAM J. Appl. Math.*, 35(2):285–300, 1978.