Comparing Apples and Oranges: Using Consensus Rankings for Decision Support

Jacob Baskin

May 10, 2008

Abstract

Conference program committees, graduate admissions committees, and even many social Web sites, are in the business of ranking content. However, they have too much content for everyone to review all of it. It therefore becomes necessary to order content based on partial information. Simply comparing numerical scores is error-prone because different reviewers see different portions of the content and score in different ways.

This paper describes a local-search algorithm for the Linear Ordering Problem specifically designed for rank aggregation that can be used to construct a consensus ordering of options based on reviewers' relative preferences. We will then discuss a system for using this ordering to categorize options by quality and to identify conflict among reviewers. Lastly, we will review the performance of this system on data extracted from a faculty search and a number of small academic conferences, and identify a number of features of reviewer behavior that may serve as obstacles for decision support systems in this domain.

NOTE: this may not be the latest available version of this paper. Please contact sk@cs.brown.edu to receive the most up-to-date information available.

1 Introduction

Whether deciding whom to hire, which applicants to admit to a graduate program, or which papers to accept to an academic conference, there are nu-

merous occasions when people must make decisions as a group, but when it is impractical for each group member to have reviewed all of the information. In these situations, the group members must quickly establish which applicants (or papers, or job candidates) deserve the most consideration from the entire group, and which are likely to be rejected.

Groups normally accomplish this task by having members rate options, and then comparing these ratings to determine which options are better. But how is this accomplished? One common method is averaging the numerical scores assigned to each candidate, and using these averages to represent each option's quality. However, this technique assumes that all reviewers mean the same thing when they give a score of, say, 6 out of 10. Yet in real-world situations, people may assign scores very differently from each other: one reviewer may almost never give scores as high as 6, but there may be another reviewer for whom 6 is below average. It is hard to tell whether these differences reflect underlying discrepancies in how reviewers assign scores, or whether they are the result of the options that one reviewer happens to have seen being better or worse than those that another person scored.

This problem can be circumvented by only taking into account the *relative preferences* of each reviewer: which candidates they prefer to others. Regardless of how high or low the actual scores are, it is clear that if reviewer A gives candidate 1 a higher score than candidate 2, she prefers 1 to 2. Thus, if we extract from each reviewer's score data an ordering of the options he reviewed, and find a way to aggregate these orderings, we can take all reviews into account together without having to produce a potentially problematic comparison of numerical scores.

Aggregating orderings—combining the relative preferences of multiple people into a single "consensus ordering" in as fair of a way as possible—is the very same problem addressed by voting theory. Accordingly, it has been much studied, beginning in the 18th century with Borda [4] and Condorcet [14]. More recently, Kemeny [10] proposed a rule whereby the consensus ranking should be chosen to minimize the number of *violations*: the number of cases where the consensus ordering ranks C_i as better than C_j, but some reviewer R_k prefers C_j to C_i.

The Kemeny rule has a number of desirable properties as a preference aggregation function [19, 18]. However, it has one very important undesirable property: it is NP-hard. Indeed, while there exists a polynomial-time approximation schema for finding the Kemeny-optimal ranking in tournaments [11], the general case is NP-hard to approximate better than 1.36... [9, 5]. Fur-

thermore, branch-and-bound methods to find optimal rankings cannot handle more than 80-100 options [3, 2]; as conferences, jobs, and graduate schools routinely have hundreds of applicants, a larger-scale method is required.

Even if we can find an optimal or near-optimal order, however, simply supplying it to the user is not enough. Presenting such an order as the "correct" ordering of the options would ignore number of important issues. An option that has received few reviews, for instance, may have a highly variable place in the ordering. Additionally, such an order presents only a relative picture of the options' quality: we do not know whether the 15th-best option is "good" or not, only that it is better than the 16th-best, 17th-best, and so forth. Lastly, if options are scored on multiple criteria, there may be different orderings that must be integrated with each other. Thus, it is necessary to manipulate the consensus order or orders to obtain a stable, objective, and integrated view of options' quality.

This paper presents both a novel local-search algorithm for finding near-optimal consensus orderings and a strategy for using these orderings to provide decision support to committees tasked with reviewing large numbers of submissions. We tested the local search algorithm against other heuristic algorithms for solving the Linear Ordering Problem, both on existing problem sets and on random problems designed to be similar to the data encountered in large rank-aggregation scenarios. Our algorithm achieves much better solutions for large and hard problems than existing algorithms, most of which have been optimized for quickly finding solutions to smaller, more "well-behaved" instances; additionally, it solves most smaller problems to optimality.

The decision support algorithm was also implemented in the **Resume** job application management system and the **Continue 2** conference management system [12], and evaluated using real-world data derived from the Brown University Computer Science Department's 2008 faculty, and from four small conferences that used **Continue 2**.

2 Local Search Algorithm

We can use reviewers' preference data to construct a weighted directed graph $G = (V, E)$, in which each vertex $v \in V$ represents an option being reviewed, and the edge from v_i to v_j has weight c_{ij} equal to the number of reviewers who prefer i to j. Given this graph, finding the Kemeny-optimal ordering

becomes an instance of the Linear Ordering Problem (LOP), in which the goal is to find a total order $>_o$ that minimizes

$$L(o) = \sum_{i,j \,:\, i >_o j} c_{ij}$$

Another formulation of this problem is to find the order that maximizes

$$U(o) = \sum_{i,j \,:\, i >_o j} c_{ji}$$

Since $L(o) + U(o)$ represents the sum of the weights of all edges in the graph, it has the same value for all $>_o$, and these formulations are equivalent. However, which formulation is chosen does affect how the results of heuristic algorithms are judged. If $\frac{L(o)}{U(o)}$ is 0.2 as it is in some of the instances found in the popular LOLIB problem set [17], then an algorithm that solves problems to within 1% of optimality when judged using the $U(o)$ formulation will solve problems to 5% optimality when judged by the $L(o)$ formulation. However, problems arising from rank aggregation tend to have much smaller values for $\frac{L(o)}{U(o)}$, as reviewers tend to generally agree with each other, resulting in a small number of "violations" relative to "non-violations". Indeed, real-world preference data from the Resume faculty application system shows $\frac{L(o)}{U(o)}$ values as low as 0.015; this means that this same algorithm would be up to 66% from the optimal $L(o)$ value for this class of problems.

All results below will be $L(o)$ values. This is opposed to most local-search algorithms' reports, which list $U(o)$ values, but the "minimal-violations" interpretation of this problem is the one that applies most to the particular case of Kemeny-optimal ordering.

2.1 High-Level Algorithm

Our algorithm has the same structure as the variable neighborhood search of Garcia et al. [6]:

$O \leftarrow$ findLocalMax(randomOrder())
bestOrder $\leftarrow O$
for $i = 1 \ldots$ numIters **do**
  for $k = 1 \ldots k_{\max}$ **do**
    $O \leftarrow \text{diversify}_k(O)$
    $O \leftarrow$ findLocalMax(O)

```
        if C(O) < C(bestOrder) then
            bestOrder ← O
            break
        else if C(O) > C(bestOrder) then
            O ← bestOrder
        end if
    end for
end for
```

Our algorithm begins with a random ordering, and proceeds to a local maximum. At every iteration, we then try to improve on that maximum by performing a series of successively larger "diversification" moves, followed by the finding of another local maximum.As soon as we reach a local maximum better than the one where we started, the series starts all over again, with the smallest diversification moves.

We improve on Garcia et al. are in two places: we have a novel technique for finding local maxima, and we modify the diversification step to be more appropriate for sparse graphs. These techniques result in major improvements, especially on the particular problem instances encountered in rank aggregation.

2.2 Finding Local Maxima: "Cascade"

Most attempts to find good local maxima for the linear ordering problem have used a *BestFit* search, in which the algorithm examines each vertex, and inserts that vertex in the position that results in the most improvement of the objective function. When none of the vertices can be moved in a way that improves the objective function, the algorithm terminates. While this does not produce a very good heuristic for the linear ordering problem on its own [16], many algorithms have used it to good effect for finding improvements on solutions arrived at by other means. [8, 2].

Our algorithm improves on *BestFit* by performing the insertion moves in an order more likely to result in a high local optimum. To do this, we use *cascadeMoveUp*, shown below, and a similar *cascadeMoveDown*.

```
procedure CASCADEMOVEUP(p)
    p_o ← p
    c_o ← 0
    for i = p − 1 . . . 0 do
```

```
        c_i ←The cost of moving the vertex at p to i
        if c_i ≤ c_o then
            c_o ← c_i
            p_o ← i
        end if
    end for
    if p_o > 0 then
        cascadeMoveUp(p_o − 1)
        cascadeMoveUp(p_o)
    end if
end procedure
```

The intuition behind these moves is that the cost of switching the vertices in p_o and $p_o - 1$ (or $p_o + 1$ in the MoveDown case) must be greater than 0, or else $p_o - 1$ rather than p_o would be the optimal position for the vertex at p. However, it is possible that moving the vertex in position p_o farther *would* decrease the cost if the vertex at $p_o - 1$ were not "in the way". Thus, we first try and move the vertex in $p_o - 1$ as far as we can, and then try once again to move the vertex in position p_o.

We can combine these two moves into a heuristic for finding local optima, by performing each of them on all vertices in order of their position:

```
procedure CASCADEALL
    repeat
        c_old ← C(O)
        for i = N − 2 . . . 0 do
            cascadeMoveDown(i)
        end for
        for i = 1 . . . N − 1 do
            cascadeMoveUp(i)
        end for
    until C(O) = c_old
end procedure
```

In practice, we have found that this move works much better than using the INSERT move alone, particularly on sparse matrices such as those produced in rank aggregation. For real-world data, we have found that this move produces results 75% closer to optimal on average than *BestFit*. On the other hand, it also performs significantly more slowly than BestFit, as the recursion results in each move examining the same vertices repeatedly. But if

we remember which vertices are "stuck"—already unable to move further—and use this information to terminate the recursion early, we can achieve a substantial speedup: with this optimization, *Cascade* takes only 50-75% longer than *BestFit*, as opposed to over 200% longer without it.

2.3 Diversification: Adapting to Sparse Problem Instances

In a typical conference, job search, or social Web site, most pairs of options have not been compared directly with each other. This means that the graph of reviewers' relative preferences will be very sparse. This poses a problem, because it results in there existing many moves that, despite changing the position of a vertex, do not substantially change the solution: if the only result of a move is that vertices with no connection to each other are rearranged, then it is unlikely this move will result in the discovery of a new local optimum.

Garcia et al. use a diversification step diversify_k which performs k random moves in which a randomly-chosen vertex is placed in a randomly chosen position other than its own. We modify this move by identifying, for a vertex k, the vertices $pred(k)$ and $succ(k)$—the closest vertices ordered lower than k and higher than k, respectively, to which k is directly connected. Our diversification step uses only random moves that switch relative order of k and either $pred(k)$ or $succ(k)$.

2.4 Computational Experiments

We evaluated our algorithm against the real-world data derived from the Brown Computer Science Department faculty search, against the instances in LOLIB [17], against the "Random Type 2" instances generated by Campos et al [1], and against a set of instances randomly generated to be similar to real-world rank aggregation instances. these instances were generated by simulating k reviewers each ranking p of n papers. Their rankings are each set initially to be identical, but are then each perturbed by m random INSERT moves. The rankings are then combined into an instance of the linear ordering problem. For each of $n = 300$ and $n = 400$ we generated both "sparse" instances ($k = n/10, p = 30, m = 15$) and "dense" instances ($k = n/5, p = 50, m = 20$). We generated 10 each of these 4 types, for 40 instances in total.

On each of these instances, we ran four variants of our algorithm:

- **VNS**: our algorithm without the *Cascade* heuristic or the modifications to the diversification step; essentially identical to the algorithm in [6].
- **VNS-D**: our algorithm with the modifications to the diversification step.
- **VNS-C**: our algorithm with the *Cascade* heuristic.
- **VNS-CD**: our algorithm with both the *Cascade* heuristic and the modifications to the diversification step.

For each of these algorithms, we set $k_{\max} = 20$, while Garcia et al. set $k_{\max} = 10$; we found that using a high $k_{\max}$ improved solutions for rank-aggregation problems in particular. We ran each algorithm for five seconds, rather than setting maxIters to a particular value; this allowed us to determine whether the additional time taken by the *Cascade* heuristic is put to better use than it would be by simply adding more iterations.

On LOLIB instances, each algorithm reached the optimal solution for every problem, with the exception of **VNS-CD**, which returned a solution 2 below optimal on be75np.

On Random Type 2 instances, our modifications to the diversification step made no substantial differences, but the *Cascade* heuristic improved the algorithm. The numbers below show the average scores of each type of algorithm on each set of instances, the average distance from the optimal solution, and the number of best solutions.

	Size 100	Size 150	Size 200
VNS	**111569.36**	385614.64	928062.52
	D: 0.00	D: 1.92	D: 8.00
	Best: 25	Best: 12	Best: 8
VNS-D	111569.92	385616.64	928064.28
	D: 0.56	D: 3.92	D: 9.76
	Best: 23	Best: 8	Best: 9
VNS-C	**111569.36**	385614.56	**928060.52**
	D: 0.00	D: 1.84	**D :6.00**
	Best: 25	Best: 13	**Best: 7**
VNS-CD	111569.52	**385613.52**	928060.92
	D: 0.16	**D: 0.90**	D: 6.40
	Best: 24	**Best: 18**	Best: 6

For the random rank-aggregation instances, the *Cascade* heuristic consistently improved on *BestFit*. The modifications to the diversification step resulted in slight improvements; however, surprisingly, these improvements were greater for dense instances than for sparse instances. Additionally, the **VNS-D** algorithm was the worst of the four on three of the four sets of problems; it seems that the *Cascade* heuristic and the diversification changes interact positively, but the diversification changes alone do not improve the solutions.

	400 Dense	300 Dense	400 Sparse	300 Sparse
VNS	2605.2	1589.6	539.5	446.5
	D: 14.2	D: 3.8	D: 3.7	D: 1.2
	Best: 1	Best: 0	Best: 1	Best: 6
VNS-D	2601.6	1593.2	541.1	448.2
	D: 10.6	D: 7.4	D: 5.3	D: 2.9
	Best: 1	Best: 1	Best: 0	Best: 2
VNS-C	2596.5	1587.0	537.0	**445.9**
	D: 5.5	D: 1.2	D: 1.2	**D: 0.6**
	Best: 4	Best: 5	Best: 5	**Best: 7**
VNS-CD	**2595.8**	**1586.9**	**536.4**	446.7
	D: 4.8	**D: 1.1**	**D: 0.6**	D: 1.4
	Best: 4	**Best: 5**	**Best: 6**	Best: 5

For the problems encountered by our decision-support system, our algorithm provides a real improvement over "standard" VNS. For the "Fit" data

set, which was sparser, VNS-CD performed best (and notably reached the best solution which much more regularity than the other algorithms), while for"Potential", both VNS-C and VNS-CD were roughly equal.

	Potential	Fit
VNS	328.89	308.33
	D: 3.6	D: 4.8
	Best: 1	Best: 0
VNS-D	328.11	308.78
	D: 3.3	D: 4.4
	Best: 1	Best: 1
VNS-C	**325.67**	306.89
	D: 0.7	D: 3.1
	Best: 5	Best: 1
VNS-CD	326.11	**305.56**
	D: 1.0	**D: 2.0**
	Best: 4	**Best: 4**

As can be seen above, the **VNS-CD** algorithm, while it appears to show gains over **VNS** for hard problems in general, succeeds especially in the domain of rank aggregation. In fact, this is the only domain in which the changes made to the diversification step are of any use; in other domains, our diversification step sometimes results in a worse heuristic. But in the particular area of Kemeny rank aggregation, both these modifications and the **Cascade** heuristic are effective.

3 Categorization Algorithm

While the **VNS-CD** algorithm yields a good ordering of the vertices, just finding a consensus ordering is not enough. This ordering may have a number of problems if used as direct a guide to which options ought to be picked. First, there may be some options that have very few reviews, and thus are not particularly constrained in their position within the order. Such options may be ranked very high, even though we are not necessarily justified in labelling them "good". Moreover, there may be multiple, equally good consensus rankings: the exact order of the options may be determined by chance. Lastly, this ranking is not linked to any judgements of the options' independent "goodness". There may be one case where the pool of options

contains a large number of good choices, and another in which the options to choose from are mostly quite poor. We would like to be able to indicate that more of the choices are worthy of consideration in the first case than in the second case.

An ordering is not the ideal tool for decision support for other reasons. Since the top options will ultimately be further reviewed in any event, the exact order—which is in any event the product, in part, of random chance—does not matter as much as the general position of the vertices. Additionally, when integrating reviewers' preferences in multiple different areas—a faculty candidate may be judged in both teaching and research ability, for example—we will not have a single consensus ordering, but a different one for each area. If we could find a way to produce a reliable and consistent picture of which vertices are good and which are bad, taking into account relative preferences and beliefs about objective quality with respect to multiple criteria, this would be much more useful than the original orderings.

To produce a categorization using our consensus ordering, we first attempt to move unconstrained vertices to better positions. This is accomplished by using the $succ(p)$ and $pred(p)$ functions defined in section 2.3. Since moving a vertex to any position in the order between its predecessor's position and its successor's would not affect the cost, all that our consensus ordering really tells us about a vertex's position is that it should be below its predecessor and above its successor. Thus, we create O_h, an order sorted by $succ(p)$ which we use to look for good vertices, and O_l, an order sorted by $pred(p)$, to use when looking for bad vertices.

The next step is anchoring each consensus ordering. To accomplish this, we return to the "raw" scores. Our goal is to figure out approximately which numerical score is equivalent to which position in the sorted order. Thus, we compute a moving average of scores—the average of the average scores within a given region in the ordering. Looking at vertices' average scores in isolation would result in the same biases we have set out to correct; on the other hand, we expect average scores to be predictive of quality most of the time. Thus, by taking a moving average, we can use the consensus ordering while still getting a rough idea of the score to which a given position in the ordering corresponds.

Using user-supplied cutoffs c_{good} and c_{bad}, we then find the "cutoff vertices" in O_h and O_l—the first vertex in O_h with a moving average score below c_{good}, and the first vertex in O_l with a moving average score above c_{bad}

Every vertex that is orderd before these vertices in O_h and O_l, respectively, are labelled "good" and "bad".

The next step is identifying vertices that are "in conflict". To accomplish this, we look at all "good" vertices $v \in G$, and calculate the percentage of preferences that compare v and some non-"good" vertex u in which u is ranked above v:

$$\text{conflict}(v) = \frac{\sum_{u \notin G} c_{uv}}{\sum_{u \notin G} c_{uv} + c_{vu}}$$

If this value exceeds the cutoff value c_{conflict}, v is labelled as "good but conflicted". A similar technique is used to identify "bad but conflicted" vertices.

Lastly, we combine the categories obtained from the rankings for each of the different criteria. For a vertex to be considered good overall, it must be categorized as good for each criterion; a similar rule applies for bad vertices. To be conisdered conflicted, however, a good (or bad) vertex need only be identified as conflicted on a single criterion.

3.1 Algorithm Effectiveness

Using the data acquired from **Resume** for the Brown Computer Science Department's faculty search, we conducted qualitative blind comparison studies in which the results of this categorization algorithm were compared with a categorization that used only the unadjusted average scores. The techniques were each rougly as effective at identifying good applicants. Evaluating our algorithm's ability to identify bad applicants was difficult since such applicants tend to be discussed less and thus be less well-remembered; however, none of the applicants that our algorithm identified as bad were invited for interviews. Our algorithm was very successful at identifying which applicants were conflicted, within both the "good" and "bad" groups.

Many of the difficulties that arose in **Resume** with our decision support technique related to applicants with only a single review; not only did such applicants have a tendency to be handled poorly by our algorithm, but we also hypothesized that their unreliable position also affected applicants with more scoring information. When we excluded from our algorithm all applicants with fewer than two scores in any category, the accuracy and stability of our algorithm vastly increased. On the other hand, it led to no information being provided for many applicants, particularly low-scored ones.

Excluding the least-reviewed applicants also makes the data in **Resume** more like a conference situation, where virtually all papers have three or more reviews. We tested our categorization algorithm using data from a number of conferences that used *Continue 2* with "Identify the Champion" [15] scoring. However, it is important to note that the number of papers ranked as "good" or "bad" was very dependent on the score cutoffs used; which cutoffs were effective varied by conference. Some cutoffs would produce zero "good" or zero "bad" papers for one conference, even though they produced reasonable breakdowns for others. This may be a result of the small range of scores offered by Identify the Champion, or it may be the case that deciding how many applicants to select as "good" and as "bad" using score-based cutoffs is not particularly effective.

To incorporate review information as fully as possible, we incorporated reviewers' expertise ratings into our algorithm. If reviewer r prefers paper a to b, and has experience e_a for a and e_b for b, we weight the reviewers' preference by $min(e_a, e_b)$. We assigned numerical values of 3, 4, and 5 to the three different experience levels specified in [15]. The intention is not to discount reviews by non-experts, but to ensure that if there is a direct disagreement between an expert and a non-expert, the expert will win.

We found that our algorithm was slightly worse at identifying which papers would be accepted than the "Identify the Champion" scoring system. While we never identified an accepted paper as "bad", we occasionally identified as "good" papers that would go on to be rejected. However, our algorithm labeled more papers as "good" or "bad" than were given scores of A and D; for papers to which no reviewer gave either of these ratings, our algorithm correctly identified which papers would go on to be accepted 80% of the time, which provided an improvement over the minimal information otherwise available.

It is worthwhile noting, furthermore, that these conferences do not represent the target environment for our decision support technique. All four conferences used for evaluation were small (between 10 and 36 submissions), and all four used the scoring system designed for "Identify the Champion"; our technique is designed for conferences with many more submissions, in which accpeting all A-rated papers may not be an option; it is also intended to be used with a scoring system with more granularity.

4 Real-World Scoring Behavior

In this section, we list some observations about the ways in which reviewers used the **Resume** scoring system, and discuss how these properties are likely to have affected the decision support algorithm.

Open Reviews In the **Resume** reviewing system, unlike those of most conferences, reviewers are not assigned to particular applicants, but are free to choose whom they review. This is a consequence of different domains: whereas reviewers in conferences are expected to arrive at their opinions only by reading the materials given, outside information is encouraged in the setting of a faculty search. Thus, as any reviewer may have information to contribute about a given applicant, reviews are left "open" to all. Additionally, reviewers may refer applicants to other reviewers if they feel a particular perspective or type of knowledge is needed; for instance, they may tell a reviewer about an applicant in their field. This results in a variable level of review coverage: some applicants receive many reviews, others receive few. This is one of the factors that results in the particular "sparseness" of the preference graph in this instance. Even if there are a large number of reviews, they will probably not be distributed evenly among all candidates.

Unscored Applicants Some applicants who are judged to be clearly "below threshold" or "out of area" for a given department are not given scores at all. This means that they are free to move anywhere in the ordering without affecting its cost. We address this particular issue in our categorization scheme: becuse we order vertices by their predecessors and successors rather than just using their supplied position, these applicants will never be judged as either good or bad. When we excluded applicants with too few scores, these are removed even before any ordering takes place, thus alleviating the problem completely.

Best-of-the-Worst Applicants Reviewers in the Brown Computer Science Department faculty search rated applicants using both a "Faculty Potential" scale and a "Goodness of Fit" scale. Reviewers were also permitted to not assign a value to one or the other of the scores if they chose not to. This led to some reviewers using the "Goodness of Fit" scale when they believed an applicant was such a bad fit that this alone should disqualify them.

Accordingly, the applicants to whom these reviewers gave their highest fit scores were not good at all, but were rather the "best of the worst". When combined with the poor review coverage for certain applicants, particularly bad ones, this led to some applicants—those who only got "best of the worst" fit scores—appearing very high in the consensus ranking for goodness of fit. This was partially addressed by the predecessors-and-successors scheme in the categorization algorithm, but some of these applicants were in fact constrained enough by the even worse fit scores given to other applicants that some of them were categorized as "good" based on their fit scores. Excluding applicants with too few scores fixed this problem entirely, as "best of the worst" applicants had only one "fit" score.

Counterpoint Reviewers Some reviewers mainly provided "counterpoint" reviews for highly-ranked applicants, in which they balanced overwhelmingly positive sentiment with questions and concerns. Though such reviews were generally accompanied by low scores, these scores did not necessarily indicate that the reviewer disliked the applicant, merely that negative information was being provided. This demonstrates a social aspect of review scoring that we found to be very important in **Resume**: scores do not just indicate relative preferences, but rather function as a richer means of communication to other reviewers. On the other hand, the relative scores reviewers give are generally *consistent* with their preferences, even if they are not given solely as a way of expressing these preferences. In this case, since reviewers who provided mostly "counterpoint" reviews tended to give very few (if any) high scores, our algorithm dealt with such reviews very nicely: relatively high "counterpoint" reviews did not affect the position or the category of the reviewed applicant, but lower reviews by the same reviewers did, as these lower reviews did likely indicate a preference against the applicants in question.

In general, despite the poor coverage resulting from the lack of assigned reviews, our algorithm performed well on data from **Resume**. Particularly when applicants without enough scores were excluded from the ranking, it managed to overcome many potential problems arising from how reviewers used the scoring system throughout the decision-making process.

5 Related Work

Cook et al. have previously examined the problem of creating a consensus ranking from those of individual reviewers [3]; a branch-and-bound algorithm is proposed that results in very fast optimal solutions for low-noise instances with up to 60 elements. By using a heuristic local-search algorithm, we sacrifice the ability to obtain an optimal solution for increased scalability; our algorithm is capable of handling much larger problems in a reasonable amount of time. We also address the problem of unreliable consensus rankings, allowing our system to be more robust and to provide more dependable information. Additionally, by returning a categorization rather than an ordering of the input elements, our system supports its users in their decision-making process, rather than simply attempting to return an "optimal" answer. Lastly, the procedure designed by Cook et al. is meant to be used in concert with their allocation scheme for peer reviews, which is not always practicable, especially in similar domains such as faculty and graduate-school application screening where reviewers must be given the opportunity to rate the applicants about whom they have personal knowledge.

Another approach to this problem was demonstrated by Hasan et al. [7] in the domain of trust recommendation. This approach replaces each score with its percentile—instead of revealing a score of 8/10, say, one would learn that a given reviewer ranked an option above 75% of the others she reviewed. While this approach is much computationally simpler, ours solves a number of problems that would plague this approach if applied to conferences, searches or similar situations. For instance, Hasan et al. assume that a person's "medium" score is reflected by the median of their observed scores. If a reviewer rates mostly good or mostly bad options, this may not be the case; as mentioned in section 4, we have encountered such behavior in real situations, with "counterpoint" reviewers. Additionally, this approach may obscure useful information that can be uncovered by rank aggregation; for instance, if we know that A is better than B and B is better than C, our algorithm will always (in the absence of conflicting information) place A above C, while this is not necessarily the case when using percentiles.

Identify the Champion [15] is a pattern language for co-ordinating the peer review process; among its components is a scoring system designed to reveal which reviewers are willing to "champion" papers at the program committee meeting. While our system uses scores in a different way, there is no reason why other components of the Identify the Champion system could

not be combined with our algorithm. Identify the Champion can also be used to design the prompts associated with the scores offered to reviewers.

Alternate Local Search Algorithms Other local search algorithms for the Linear Ordering Problem have been proposed, using metaheuristics such as tabu search [13], scatter search [1], and simulated annealing [16]; Huang and Lim have also developed a genetic algorithm for the Linear Ordering Problem [8]. However, most of these algorithms were designed for instances such as those encountered in the Triangulation Problem for Input-Output Matrices in economics; indeed, the popular LOLIB library of sample instances for the linear ordering problem are derived entirely from this source, and tend to be both smaller (60 elements or fewer) and denser than the problems encountered in our domain. Our algorithm is designed specifically for large, sparse problem instances, resulting in much improved behavior for rank aggregation in particular; in addition, the introduction of the "Cascade" method of finding local optima results in improvements over existing algorithms on all problems.

6 Conclusion and Future Work

This paper has presented an improved local search algorithm for the linear ordering problem based on variable neighborhood search. We have discussed the *Cascade* procedure, which generates significantly better local optima than the *BestFit* heuristic used in the past. This procedure, along with a strategy for more effectively diversifying solutions in "sparser" problem instances, yields major improvements, when integrated into VNS, particularly for problem instances arising from rank-aggregation.

While the Linear Ordering Problem has been studied extensively in the context of combinatorial optimization, these improvements demonstrate that there are still many possibilities to explore. Since the majority of research has focused on problem instances from domains other than rank aggregation, the work of producing algorithms tuned to this particular area has only just begun. Moreover, while many different local-search heuristics have been proposed, there has been very little research on finding neighborhoods for local search that go beyond the simple INSERT move.

We also discussed a decision-support algorithm that uses the consensus ordering produced by this local search procedure to help find which of the op-

tions under review are good and which are bad, as well as to identify conflict between reviewers over particular options. In the difficult case presented by data from the **Resume** job application system, this algorithm particularly excels at identifying conflict. For small conferences, while this algorithm does not improve the ability of "Identify the Champion"' [15] to reliably identify papers that will be accepted, it does give more useful information about papers in the middle of the pack. Combining our ranking approach with "Identify the Champion" may result in a very effective system, and would be an interesting direction for future research on this topic.

Much more empirical data is required to give a good assessment of the efficacy of this decision support algorithm in comparison to less-sophistocated methods; this data will be created as the algorithm is used in the **Continue 2** system. It may be possible to design effective decision-support algorithms for this task that do not rely on solving the NP-hard Linear Ordering Problem to find a consensus ordering. Additionally, finding better ways of reducing the variability of consensus rankings—perhaps identifying and extricating disconnected components of the preference graph, for instance—could result in algorithms that are based on Kemeny rank aggregation being less susceptible to strange patterns of reviews such as those discussed in section 4 above.

References

[1] V. Campos, M. Laguna, and R. Martí. Scatter search for the linear ordering problem. pages 331–340, 1999.

[2] I. Charon and O. Hudry. A branch-and-bound algorithm to solve the linear ordering problem for weighted tournaments. *Discrete Appl. Math.*, 154(15):2097–2116, 2006.

[3] W. D. Cook, B. Golany, M. Penn, and T. Raviv. Creating a consensus ranking of proposals from reviewers' partial ordinal rankings. *Computers & Operations Research*, 34(4):954–965, 2007.

[4] J.-C. de Borda. Mémoire sur les élections au scrutin. *Histoire de l'Académie Royale des Sciences*, 1781.

[5] I. Dinur and S. Safra. The importance of being biased. In *STOC '02: Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 33–42, New York, NY, USA, 2002. ACM.

[6] C. G. Garcia, D. Pérez-Brito, V. Campos, and R. Martí. Variable neighborhood search for the linear ordering problem. *Comput. Oper. Res.*, 33(12):3549–3565, 2006.

[7] O. Hasan, L. Brunie, J.-M. Pierson, and E. Betino. Elimination of subjectivity from trust recommendation, 2008.

[8] G. Huang and A. Lim. Designing a hybrid genetic algorithm for the linear ordering problem. In *GECCO*, pages 1053–1064, 2003.

[9] R. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, NY, 1972. Plenum Press.

[10] J. Kemeny. Mathematics without numbers. *Daedalus*, 88:571–591, 1959.

[11] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 95–103, New York, NY, USA, 2007. ACM.

[12] S. Krishnamurthi, P. Hopkins, J. McCarthy, and J. Baskin. Continue 2.0 conference management system, 2008.

[13] M. Laguna, R. Marti, and V. Campos. Intensification and diversification with elite tabu search solutions for the linear ordering problem. *Comput. Oper. Res.*, 26(12):1217–1230, 1999.

[14] Marie Jean Antoine Nicolas de Caritat, marquis de Condorcet. Essai sur l'application de l'analyse á la probabilité des décisions rendue á la pluralité des voix. *Paris: Imprimerie royale*, 1785.

[15] O. Nierstrasz. Identify the champion. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design*, volume 4, pages 539–556. Addison Wesley, 2000.

[16] J. Petit. Experiments on the minimum linear arrangement problem. *J. Exp. Algorithmics*, 8, 2003.

[17] G. Reinelt. Lolib, 1997.

[18] D. Saari and F. Valognes. Geometry, voting, and paradoxes. *Mathematics Magazine*, 71(4):243–259, 1998.

[19] H. Young and A. Levenglick. A consistent extension of condorcet's election principle. *SIAM Journal on Applied Mathematics*, 35(2):285–300, 1978.