On Discovering Bucket Orders from Preference Data

Sreyash Kenkre* Arindam Khan† Vinayaka Pandit*

Abstract

The problem of ordering a set of entities which contain inherent ties among them arises in many applications. Notion of "bucket order" has emerged as a popular mechanism of ranking in such settings. A bucket order is an ordered partition of the set of entities into "buckets". There is a total order on the buckets, but the entities within a bucket are treated as tied.

In this paper, we focus on discovering bucket order from data captured in the form of user preferences. We consider two settings: one in which the discrepancies in the input preferences are "local" (when collected from experts) and the other in which discrepancies could be arbitrary (when collected from a large population). We present a formal model to capture the setting of local discrepancies and consider the following question: "how many experts need to be queried to discover the underlying bucket order on n entities?". We prove an upperbound of $O(\sqrt{\log n})$. In the case of arbitrary discrepancies, we model it as the bucket order problem of discovering a bucket order that best fits the data (captured as pairwise preference statistics). We present a new approach which exploits a connection between the discovery of buckets and the correlation clustering problem. We present empirical evaluation of our algorithms on real and artificially generated datasets.

Keywords: rank aggregation, bucket order, correlation clustering.

1 Introduction

In rank aggregation, we are given multiple ranked lists of entities, and the goal is to compute a robust ranking that captures the essence of the input rankings. It was first studied in social choice theory [17, 18, 24, 23] and has diverse applications in meta-search [9, 4], combining experts [7], and similarity searching [11]. Much of the classical literature is on computing a total order on the entities. However, the nature of the required ordering varies depending on the application.

Consider as an example, the problem of aggregating movie ratings by experts. Let U denote the universe of movies of interest. Let us assume that the experts are asked to rate each movie in U on a scale of 1 to 10. Invariably, there will be contradictions in the assessments of different experts. Moreover, they may even apply different levels of granularity in their scores. In such a setting, it is not very meaningful to rank a movie with average rating of 7.61 higher than a movie with average rating of 7.56. Arguably, it is more meaningful to partition the movies into a set of ordered groups such that, (i) movies with similar rating patterns are in the same group and (ii) movies with significantly different ratings are in different groups.

The notion of *bucket order* was formalized by Fagin et al. [10] as a way to order entities in settings such as the movie ratings example. A bucket order is an ordered partition of the set of entities into "buckets", $B_1, B_2, \ldots B_k$. All the entities within a bucket are assumed to be tied or incomparable. The order between two entities of different buckets is given by the relative ordering of the buckets they belong to (entities in B_i are ranked higher than the entities in B_j if $i < j$). The *bucket order problem* [11, 10, 14, 12] is, given a set of input rankings, compute a bucket order that best captures the data. The input rankings could be defined on *subsets* of the universe of entities; they could even be a large collection of pairwise preferences. The bucket order problem has been used to discover ordering information among entities in many applications. It is used in the context of "seriation problems" in scientific disciplines such as Paleontology [22, 13], Archaeology [15], and Ecology [20]. Feng et al. [12] used it to aggregate browsing patterns of the visitors of a web portal. Agrawal et al. [1] implicitly use bucket orders to discover ordered labels from clickstream data. Some of these applications are explained in Section 6.

In this paper, we focus on discovering bucket orders from ranking preferences of a population. We consider two settings for disagreements between the ranking preferences of different people. These disagreements will also be referred to as discrepancies. In the first setting, the discrepancies are "local" (typical when data is collected from experts). We formalize this setting

*IBM Research - India. email:{srkenkre,pvinayak}@in.ibm.com

†Georgia University of Technology. This work was done when the author was working in IBM Research - India. email: arindamkhan.cs.iitkgp@gmail.com

and study the "query complexity": "how many experts need to be queried to reliably discover the underlying bucket order on n entities". We prove an upperbound of $O(\sqrt{\log n})$ in this setting. When the discrepancies are arbitrary, we model the problem as the traditional bucket order problem studied in [11, 10, 14, 12]. We develop a novel approach which exploits a natural relationship between correlation clustering [5] and the properties satisfied by the buckets in a bucket order. We conduct experiments on two well-studied real-life datasets and a large number of artificially generated datasets. In summary, we make important theoretical contribution in the form of query complexity, develop new algorithms, and present empirical evaluation of our and past approaches.

2 Past Work and Our Results

Some Definitions: We define a few concepts needed to survey the literature. Let U denote the universe of entities. A *full ranking* is a total order on the entities in U. A bucket order specifies a partial order on the set of entities in U and is referred to as a *partial ranking*.

The notion of "distance" between two rankings is critical in defining aggregation problems. The *Kendall tau* distance [18] between two full rankings π_1 and π_2 is the number of pairs which appear in opposite orders in π_1 and π_2. The *Spearman's footrule* distance between them is $\sum_i(|\pi_1(i) - \pi_2(i)|)$ where $\pi_k(i)$ is the position of i in π_k for $k = 1, 2$ [18]. These notions were generalized for the case of partial rankings in [10].

A *linear extension* of a bucket order is a full ranking such that, for all i, entities of bucket i are ranked higher than the entities of bucket $i + 1$. Entities within a bucket may appear in any order. Let $B_1 = \{2, 6, 3\}, B_2 = \{1, 5, 9\}, B_3 = \{4, 7, 8\}$ be a bucket order on $\{1, 2, \ldots, 9\}$ with $B_1 \prec B_2 \prec B_3$; $< 6, 3, 2, 9, 5, 1, 8, 7, 4 >$ and $< 2, 3, 6, 5, 1, 9, 8, 7, 4 >$ are examples of its linear extensions.

2.1 Past Work

Fagin et al. [10] defined four distance metrics based on generalized Kendall tau and Spearman's footrule distances for partial rankings. These metrics are within a constant-factor of each other. The goal is to discover a bucket order whose average distance from the input rankings is minimum. They present a dynamic programming based constant-factor approximation algorithm for one of the metrics. This in turn yields constant-factor approximation algorithms for the other metrics as well.

Ailon et al. [3] showed that a simple and powerful method called PIVOT yields constant-factor approximation algorithms for a host of related problems including rank aggregation of full rankings. Ailon [2] considered the problem of aggregating a set of partial rankings into a full ranking. He generalized the algorithm of [3] for this case and obtained an improved approximation ratio of $\frac{3}{2}$. Kenyon-Mathieu and Schudy [19] gave a Polynomial Time Approximation Scheme for the related problem of ranking players in a tournament (in a tournament, every player plays every other player).

In standard rank aggregation, preferences between entities are captured in the form of input rankings. Collecting the statistics on all pairwise preferences is an alternative way of capturing preferences, especially when data is collected from a large population. Cohen et al. [7] first considered normalized pairwise preference statistics, expressed in the form of a matrix, to compute an aggregate full ranking. Gionis et al. [14] formulated the bucket order discovery problem when the input is given as a normalized pairwise preference matrix. They showed how to convert the discovered bucket order into an equivalent normalized pairwise preference matrix. They consider the problem of computing a bucket order whose L_1 *distance* from the input matrix is minimum and showed it to be NP-Complete. They adopted the PIVOT algorithm [3] and established an approximation ratio of 9. Feng et al. [12] considered the same formulation as Gionis et al., but restricted the input to a set of total orders. They proposed the bucket gap algorithm (called GAP). At a high level, they follow the approach of Fagin et al. [11] of grouping the entities based on their median ranks. However, they strengthen this heuristic by taking different quantile ranks of the entities which depends on a parameter called "num_of_frequencies". In the context of generating ordered labels from click-stream data, Agrawal et al. [1] considered *Max Ordered Partition Problem* which implicitly uses bucket orders.

2.2 Our Results

As mentioned in Section 1, we consider two different models of discrepancies in the input: local and arbitrary discrepancies.

We describe the local discrepancy model using an example of rating movies by experts. Suppose there is an underlying bucket order that categorizes the movies into "classics", "excellent", "good", "average" and so on. Suppose further that the experts are required to provide total order on the movies. It is possible that an expert ranks an excellent movie at the level of good movies due to subjective considerations. But, it is unlikely that the expert ranks it at the level of average movies. In other words, the subjective inputs given by the experts differ from the underlying bucket order only in terms of misclassifications across neighboring buckets. Therefore, the total orders given by the experts are "close" to some linear extension of the underlying bucket order. We call this as "local noise". We show that with local noise under an appropriate probabilistic model, it is sufficient to query only $O(\sqrt{\log n})$ expert inputs, to reliably compute the underlying bucket order on n entities. We call this as the *query complexity* of the bucket order discovery.

Our model of query complexity is applicable in many scenarios. Consider the seriation problem in Paleontology [14]; the problem is to order a set of fossil sites in terms of their temporal order (See Section 6.1). Indeed, the broad agreement among domain experts is best captured by a bucket order. Assessments by individual experts differ at the boundary of different eras. In fact, Puolämaki et al. [22] generate a set of full rankings on a set of fossil sites in Europe based on this premise and it is used in the experiments of [14, 12].

The general setting of arbitrary discrepancies arises when the preferences are collected from large populations. In this case, we assume that the ordering preference of the population is captured by pairwise preference statistics, and model bucket order discovery as the bucket order problem considered in [14, 12]. We present a new approach which exploits the connections between the process of identifying the buckets and the well studied correlation clustering problem [5].

We need to present a brief overview of the PIVOT algorithm [14] in order to motivate our approach. The PIVOT algorithm randomly picks a pivot entity, and then classifies the remaining entities into three categories, B, R, and L. B is the set of entities which belong to the same bucket as the pivot. L is the set of entities that belong to buckets ranked higher than B and R is the set of entities belonging to the buckets which are ranked lower than B. It recurses on R, L to find the bucket order of the entities in R and L. The final bucket order is: bucket order for L followed by B followed by the bucket order for R. Thus, this algorithm combines the two important steps, of finding the appropriate buckets, and of finding an ordering among them, into a single step. This method has the drawback of being sensitive to noise in pairwise comparisons involving the pivot (See Section 5). Our approach is to first get appropriate buckets via clustering and then obtain a total order on the clusters. Specifically, we use correlation clustering to obtain the buckets.

Correlation Clustering [5] is a clustering problem on a graph in which the edges are labeled positive or negative. The goal is to find a clustering that minimizes the number of disagreements. Disagreements can arise in two ways: (i) a positive edge cuts across two different clusters, and (ii) a negative edge lies completely inside a cluster. We first demonstrate how to convert the problem of obtaining the buckets to a correlation clustering instance. We use a simple algorithm for correlation clustering by Ailon et al. [3]. We develop two different heuristics for computing the total order on the buckets obtained in the first step.

We present detailed empirical evaluation of our and previous algorithms. Specifically,

- We present empirical evaluation on real-life data: the *g10s10* dataset in Paleontology [22] and the data on the browsing patterns of the visitors of MSNBC [1]. We identify and correct a flaw in the experiment on the MSNBC dataset in [12].

- We experiment with large number of artificially generated datasets that are designed to test the robustness of the algorithms.

- We demonstrate the strengths and the weaknesses of different algorithms. We present experimental results to validate the query complexity result.

3 Problem Formulations

In this section, we formally define the bucket order problem and the query complexity of bucket order discovery. In the rest of the paper, we use the symbol "$<$" to denote an order between entities; $a < b$ means a is ranked higher than b. For the rest of the paper, we use the terms *entities* and *elements* interchangeably.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be the universe of entities (or elements) Let $\mathbf{B} = \{B_1, B_2, \ldots, B_\ell\}$ be a partition of V (i.e. $B_i \subseteq V$, $\cup_{i=1}^{i=\ell} B_i = V$ and $B_i \cap B_j = \phi$ if $i \neq j$). Let $\prec$ be a total order on $\mathbf{B}$, i.e. $B_1 \prec B_2 \prec \ldots \prec B_\ell$. We say that $(\mathbf{B}, \prec)$ forms a bucket order on V. We refer to each block B_i of the partition as *bucket*. The semantics of the bucket order is as follows. If $v \in B_i$ and $u \in B_j$ and $B_i \prec B_j$, then, $v < u$. In addition, entities within a bucket are incomparable or tied.

We say that $<_T$ is a *linear extension* of $(\mathbf{B}, \prec)$, if it is a total order on V that obeys the bucket order, i.e, for $v \in B_k$ and $u \in B_l$ such that $B_k \prec B_l$, $<_T$ is guaranteed to have $v <_T u$.

A *pair ordered matrix* (p.o. matrix in short) is a generic way of capturing pairwise preferences in the input data. It is a $|V| \times |V|$ matrix M. For each $v_i, v_j \in V$, $M(i,j) \geq 0$ is the fraction of the comparisons between v_i and v_j, which ranked $v_i < v_j$. Further, it satisfies that, $M(i,j) + M(j,i) = 1$ and by convention, $M(i,i) = 0.5$. It is easy to convert an input given in the form of full or partial ranks into an

[1]http://kdd.ics.uci.edu/databases/msnbc/msnbc.html

equivalent p.o. matrix. Given a bucket order $(\mathbf{B}, \prec)$, we define an equivalent p.o. matrix as follows: let $C_{\mathbf{B}}(i,j) = 1$ if $v_i < v_j$, $C_{\mathbf{B}}(i,j) = \frac{1}{2}$ if v_i and v_j belong to the same bucket, and $C_{\mathbf{B}}(i,j) = 0$ if $v_j < v_i$.

3.1 Bucket Order Problem [14, 12] The input to the problem is, a set of rankings (full or partial) or a collection of pairwise preferences, which is represented as an equivalent p.o. matrix M on V. For a bucket order $(\mathbf{B}, \prec)$ on V, let $C_{\mathbf{B}}$ denote the corresponding p.o. matrix. The goal is to find a bucket order such that the L_1 distance between M and $C_{\mathbf{B}}$ is minimum. In other words, find a bucket order $(\mathbf{B}, \prec)$ which minimizes $|C_{\mathbf{B}} - M| = \sum_{i,j} |C_{\mathbf{B}}(i,j) - M(i,j)|$.

Remarks: In some applications (Paleontology application in Section 6), there is an unknown underlying bucket order from which the input data is sampled, and the goal is to find a bucket order that closely approximates it. While in some applications (MSNBC application in Section 6), there is no such underlying bucket order. However, a bucket order is used only as a means of effectively capturing the aggregate preference order in the input. Feng et al. [12] first highlighted these two aspects of the bucket order problem.

3.2 Query Complexity We now formalize the problem of the number of expert inputs required to discover an unknown underlying bucket order when the inputs are obtained in the form of total orders. As argued in the introduction, it is reasonable to assume that the expert inputs are close to some linear extension and the discrepancies are "local". Querying the experts (human or access to competing ranking functions) is a costly process and it is a worthy goal to minimize the number of queries required to learn the underlying bucket order.

Suppose there is an underlying bucket order $(\mathbf{B}, \prec)$ on the elements of V. Let the bucket order be $B_1 \prec B_2 \prec \ldots \prec B_\ell$. Let b_i denote the size of B_i and let $S_0 = 0$ and $S_i = S_{i-1} + b_i \forall i \geq 1$. The *range* of an element $v \in B_i$ is said to be $range(v) = [S_{i-1}+1, S_i]$. Thus, in a linear extension, every element occurs within its range. Given a total order on V, the *displacement error* of an element v is said to be d, if v occurs within a distance d to the right, or to the left of $range(v)$.

We model the discrepancies in the expert inputs taking the movie rating application as an example (see Section 2.2). It is likely that an expert, due to subjective considerations ranks an excellent movie at the level of good movies, i.e, an entity in bucket B_i may be placed in buckets B_{i-1} or B_{i+1}. It is highly unlikely that the expert ranks an excellent movie at the level of average movies. So, we may assume that no entity of B_i is placed beyond B_{i-1} or B_{i+1}. Further, it is highly unlikely that the expert ranks a large number of good movies ahead of an excellent movie, i.e, the displacement error of an entity v is no more than half the size of the adjacent bucket to which it is getting displaced. We thus have the following model for *local errors* that captures biases or erroneous recordings.

1. If a displacement error occurs for $v \in B_i$, then v is placed either in B_{i-1} or B_{i+1}.

2. If an entity $v \in B_i$ is erroneously placed in B_j, $j \in \{i-1, i+1\}$, then, its displacement error is atmost $|B_j|/2$.

For the rest of this paper, the term *local error* will refer to the above two conditions. Let Q be the set of all total orders which satisfy the local error property. We assume that the expert inputs are drawn uniformly at random from Q. The *query complexity* problem is defined as follows: given n, the number of entities, how many expert inputs need to be sampled for discovering the bucket order? Our formulation is similar in spirit to the problem of *sorting under noise* which is studied in prior literature (see [6, 16]).

4 Discovering the underlying Bucket Order

We consider an unknown underlying bucket order $(\mathbf{B}, \prec)$ with l buckets. Let $\mathcal{T}$ be the set of orders that have a displacement error of at most half the size of the adjacent buckets. Each query returns a total order chosen uniformly at random from $\mathcal{T}$. The goal is to bound the number of queries required to reconstruct $(\mathbf{B}, \prec)$ with high probability. Let $\mathbf{B} = \{B_1, B_2, \ldots, B_\ell\}$. Let b_i denote the size of B_i and d_i denote $b_i/2$.

Claim 4.1. *The probability that a query to $(\mathbf{B}, \prec)$ returns a particular total order with local error is equal to*

$$\frac{1}{b_1!b_2!\ldots b_\ell!\binom{d1+d2}{d1}\binom{d2+d3}{d2}\ldots\binom{d_{\ell-1}+d_\ell}{d_{\ell-1}}} \tag{4.1}$$

Proof. Number of linear extensions of $(\mathbf{B}, \prec)$ is $b_1!b_2!\ldots b_\ell!$. Fix one of these linear extensions and let O_i denote the order in which elements of B_i occur. There are $l-1$ regions, each of length $d_i + d_{i+1}$, at the boundaries of O_i and O_{i+1} for $i = 1, \ldots, l-1$ where local error can arise. Moreover, there are $\binom{d_i+d_{i+1}}{d_i}$ ways of placing the last d_i elements of O_i in this region while preserving the ordering of elements of O_{i+1}. Therefore, there are $\Pi_{i=1}^{i=\ell-1}\binom{d_i+d_{i+1}}{d_i}$ ways of obtaining unique total orders from each linear extension obeying the local error conditions. This implies that

$|\mathcal{T}| = b_1!b_2!\dots b_\ell!\binom{d_1+d_2}{d_1}\binom{d_2+d_3}{d_2}\dots\binom{d_{\ell-1}+d_\ell}{d_{\ell-1}}$ Since the samples are chosen uniformly at random from $\mathcal{T}$, the proof of the claim follows.

To prove an upper bound of $O(\sqrt{\log n})$ on the query complexity, we give an algorithm which queries only $\sqrt{200 \log n}$ expert inputs for total orders, and produces the correct bucket order with a very high probability. The algorithm is described in **Algorithm 1**. In the algorithm, we first get the elements of the left most bucket, remove them from the input total orders and iterate the process to get the remaining buckets. Let $T_1^j, T_2^j, \dots, T_k^j$ be the k input bucket orders at the j^{th} iteration. Let a_i denote the leftmost element of T_i^j, and let $L^j = \{a_1, a_2, \dots a_k\}$. For an element v, we count the number of times it occurs to the left of some element in L^j. This gives us a *score*, which we use to decide which bucket v belongs to. So if $\mathbf{1}_{T_i^j}(u < v)$ is a function that is 1 if $u < v$ in order T_i^j, and zero other wise (i.e. the indicator function of $\{u < v\}$), we have the following definition of $\mathbf{score}(j, v)$.

$$\begin{aligned} \mathbf{score}(T_i^j, v) &= \sum_{a \in L^j} \mathbf{1}_{T_i^j}(v < a) \\ \mathbf{score}(j, v) &:= \sum_{i=1}^{i=k} \mathbf{score}(T_i^j, v) \end{aligned} \tag{4.2}$$

At the j^{th} iteration we assume that the buckets $B_1, B_2, \dots, B_{j-1}$ have been output correctly, and their elements deleted from $T_1^j, \dots, T_k^j$. Then the set L^j of all the left most elements of T_i^j are in B_j. Hence it is likely that the $\mathbf{score}()$ for entities of B_j are higher than for the entities not in B_j. We output as A_j, the entities with a high score.

```
1 Set $i = 0$;
2 Query $(\mathbf{B}, \prec)$ to get $k$ input orders
  $T_1^i, T_2^i, \dots, T_k^i$;
3 Let $L^i$ be the left most elements of
  $T_1^i, T_2^i, \dots, T_k^i$;
4 For every entity $v$, calculate $\mathbf{score}(i, v)$;
5 Let $A_i$ be the set of elements that have a score
  of at least $(3/8)\binom{k}{2}$;
6 Output $A_i$ as the bucket $B_i$. Delete the
  elements of $A_i$ from each of the orders
  $T_1^i, \dots, T_k^i$, to get $T_1^{i+1}, \dots, T_k^{i+1}$;
7 If there are elements left in the orders, set $i$ to
  $i + 1$ and repeat the steps starting from step 3;
```

Algorithm 1: Bucket Reconstruction Algorithm

Let A_j be the bucket returned by the algorithm as B_j. We first prove that every element v in B_j, will be in A_j with a high probability.

Theorem 4.1. *let $v \in B_j$. Suppose that the algorithm returns the buckets $B_1, B_2, \dots, B_{j-1}$ correctly. Then the probability that v is not in A_j is at most $\frac{1}{n^2}$, for $k = \sqrt{200 \ln n}$.*

Proof. Let $L^j = \{a_1, a_2, \dots, a_k\}$ be the left most elements of the input orders $T_1, T_2, \dots, T_k$, at the j-th iteration. Since we assume that the buckets $B_1, B_2, \dots, B_{j-1}$ have been returned correctly, the left most elements of $T_1^j, \dots, T_k^j$ at the j^{th} iteration belong to B_j, i.e. $L^j \subseteq B_j$. The $\mathbf{score}()$ function is now defined with respect to L^j. Suppose $v \notin A_j$. Then, from the algorithm we see that $\mathbf{score}(j, v) < (3/8)\binom{k}{2}$. For $a_i \in L^j$, let X_i^m be the indicator random variable that v occurs to the left of a_i in T_m^j. We then have

$$\begin{aligned} \mathbf{score}(j, v) = \; & X_1^1 + X_1^2 + \dots + X_1^k \\ & + X_2^1 + X_2^2 + \dots + X_2^k \\ & \dots \\ & + X_k^1 + X_k^2 + \dots + X_k^k \end{aligned} \tag{4.3}$$

Now, since $v \in B_j$, and $a_i \in B_j$, the probability that v occurs before a_i in a random order is $1/2$. Thus the expected value if X_i^m is

$$\begin{aligned} E[X_i^m] &= \frac{1}{2} \; \forall m = 1, \dots, k, m \neq i \\ &= 0 \;\; \text{if } m = i \end{aligned}$$

Thus, the expected value of $\mathbf{score}(j, v)$ is

$$\begin{aligned} E[\mathbf{score}(j, v)] &= \frac{k(k-1)}{2} \\ &= \binom{k}{2} \end{aligned}$$

Since the X_i^m are independent $0-1$ random variables, we have using *Chernoff Bounds* (theorem 4.5 in [21])

$$\begin{aligned} \mathbf{Pr}\left(\mathbf{score}(j, v) < \frac{3}{8}\binom{k}{2}\right) &\leq \exp\left(-\frac{1}{2}\binom{k}{2}\frac{5}{8}\right) \\ &\approx \exp\left(-\frac{25}{256}k^2\right) \end{aligned}$$

For $k = \sqrt{200 \ln n}$, we get the above probability to be at most n^{-2}, which proves the above theorem.

Next, we bound the probability that an element w not in B_j is in A_j.

Theorem 4.2. *Let $w \notin B_j$. Suppose that the algorithm returns the buckets $B_1, B_2, \dots, B_{j-1}$ correctly. The probability that $w \in A_j$ is at most n^{-2}, for $k = \sqrt{200 \ln n}$.*

Proof. Since the buckets $B_1, \dots B_{j-1}$ have been returned correctly, and since the displacement error is at most half the size of the buckets, $w \in B_{j+1}$. Let Y_i^m denote the indicator random variable that in the order T_m^j, w occurs before a_i. Suppose w occurs before a_i in an order. Then a_i lies to the right of $range(a_i)$ by at most d_{j+1} places, or w lies to the left of $range(w)$ by at most d_j places. Let this $d_j + d_{j+1}$ region on the boundary of B_j and B_{j+1} be denoted by D. Suppose w occurs at index x and a_i occurs index y in D. The number of orders in which this can occur is

$$b_1! b_2! \dots (b_j - 1)! \binom{d_j + d_{j+1} - 2}{d_j - 1} (b_{j+1} - 1)! b_{j+2}! \\ \dots b_\ell! \binom{d_1 + d_2}{d_1} \dots \binom{d_{j-1} + d_j}{d_j} \binom{d_{j+1} + d_{j+2}}{d_{j+1}} \\ \dots \binom{d_{\ell-1} + d_\ell}{d_{\ell-1}}$$

Since there are $\binom{d_j + d_{j+1}}{2}$ choices for the indices x and y where w occurs before a_i, and since each order is equally likely, the probability that w occurs before a_i in a given order is (after multiplying by $\binom{d_j + d_{j+1}}{2}$ and the probability of occurrence of the order and simplifying)

$$\begin{aligned} \mathbf{Pr}(Y_i^m = 1) &= \frac{1}{2} \frac{d_j}{b_j} \frac{d_{j+1}}{b_{j+1}} \text{ if } i \neq m \\ &= 0 \text{ if } i = m \end{aligned}$$

We now get the expectation of the score of w to be

$$\begin{aligned} E[\mathbf{score}(j, w)] &= k(k-1) \frac{1}{2} \frac{d_j}{b_j} \frac{d_{j+1}}{b_{j+1}} \\ &= \binom{k}{2} \frac{1}{2} \frac{1}{2} \text{ as } d_j = \frac{1}{2} b_j \\ &= \frac{1}{4} \binom{k}{2} \end{aligned}$$

Since the Y_i^m are independent $0-1$ random variables, we have using *Chernoff Bounds* (theorem 4.4 in [21])

$$\begin{aligned} \mathbf{Pr}\left(\mathbf{score}(j, w) > \frac{3}{8} \binom{k}{2}\right) &\leq \exp\left(-\frac{1}{4} \binom{k}{2} \frac{1}{3} \frac{1}{4}\right) \\ &\approx \exp\left(-\frac{k^2}{96}\right) \end{aligned}$$

Since $k = \sqrt{200 \ln n}$, we find that the above probability is bounded by n^{-2}.

We now use the above lemmas to prove that our algorithm succeeds with high probability.

Theorem 4.3. *By sampling from $k = \sqrt{200 \ln n}$ total orders, the above algorithm gets the correct bucket order with a probability of at least $1 - \frac{1}{n}$.*

Proof. The algorithm first outputs A_1, then deletes the entities in A_1 from each of the input orders $T_1, \dots, T_k$ and repeats to find A_2 and so on. Let S_i be the event that the algorithm outputs the i^{th} bucket correctly. The algorithm fails if there is some step i, during which it outputs the a faulty bucket. Suppose $S_1, S_2, S_3, \dots, S_{i-1}$ occur. Then $A_i \neq B_i$ only if an element of B_i is missing from A_i, or an element not in B_i is in A_i. From the above two lemmas, we conclude that the probability of this occurring is at most $2n^{-2}$. Thus the probability that the algorithm fails in some step is

$$\begin{aligned} &\leq \mathbf{Pr}(\overline{S_1} \cup (\overline{S_2} | S_1) \cup (\overline{S_3} | (S_1 \cup S_2) \dots)) \\ &\leq \sum_i \mathbf{Pr}(\overline{S_i} | (S_1 \cup S_2 \cup \dots \cup S_{i-1})) \\ &\leq \ell \frac{2}{n^2} \\ &\leq \frac{1}{n} \end{aligned}$$

Thus, with probability at least $(1 - \frac{1}{n})$ our algorithm outputs the correct bucket order.

5 Algorithms for the Bucket Order Problem

In this section, we consider the general setting of arbitrary discrepancies and present a new approach for the bucket order problem (see Section 3.1).

Recall the main "pivoting" step of the PIVOT algorithm of Gionis et al. [14]: based on a randomly chosen pivot p, three sets B, R, and L are created. B is the set of entities belong to the pivot's bucket, L is the set of entities that should occur before the pivot, and R is the set of entities that should occur after the pivot. The pivoting step determines the relative ordering of non-pivot entities with respect to the pivot and the algorithm recurses on L and R respectively. Suppose M is the input p.o. matrix. The splitting of the entities into B, R, L is based on a parameter called β. An entity q is put in B if $|M[p,q] - 0.5| \leq \beta$. It is put in L is $M[p,q] < (0.5 - \beta)$ and in R if $M[p,q] > 0.5 + \beta$. The PIVOT algorithm combines the two steps of (i) obtaining appropriate buckets and (ii) finding relative ordering among the buckets.

The pivot step is sensitive to discrepancies in the pairwise comparisons involving the pivot p. Let us consider an extreme form of discrepancy in which the pairwise preferences involving p are reversed. In this case, the pivot step introduces a major error where the relative ordering between the entities classified as R and L is reversed. While such a discrepancy is highly unlikely, it emphasizes the danger of ordering entities based solely on their pairwise preferences with respect to the pivot. So, even if a small (but not negligible) fraction of the entities have discrepancies in their pairwise comparisons, the pivot step has a corresponding probability of introducing errors in its ordering. Our approach is to decouple the steps of obtaining the buckets and ordering them.

We adopt a two-phase approach in which we first discover the appropriate buckets. An ordering on the buckets is obtained by looking at preference statistics at the level of buckets, rather than individual entities as in the PIVOT algorithm. Our first observation is that, ideally, we would like the buckets to be "clustered" in the following sense. Let S be a bucket discovered in the first phase. We would like to ensure the following conditions: (i) $\forall a, b \in S,\ M(a, b) \approx 0.5$ and (ii) $\forall a \in S, \forall b \notin S,\ |0.5 - M(a, b)| \gg 0$. We can parameterize this by a number $0 \leq \beta \leq 0.5$ as follows: (i) $\forall a, b \in S,\ |0.5 - M(a, b)| \leq \beta$ and (ii) $\forall a \in S, \forall b \notin S,\ |0.5 - M(a, b)| > \beta$. We now show how to reduce the problem of discovering buckets which satisfy the above properties to the problem of *correlation clustering* [5].

The input to correlation clustering is a graph whose edges are labeled +ve or -ve. Our goal is to find disjoint clusters that cover all the nodes and minimize overall *disagreements*. A disagreement is said to occur if a +ve edge cuts across two clusters or if a -ve edge is inside a cluster.

```
  input : p.o. matrix $M$ over entities $V$
  output: Bucket Order $(\mathbf{B}, \prec)$
1 $G$ = ObtainCCInstance($M$);
2 $\mathbf{B}$ = CorrelationCluster($G$) ;
3 $(\mathbf{B}, \prec)$ = SortHeuristic($\mathbf{B}$) (OR PIVOT($\mathbf{B}$); See
  text for details);
```

Algorithm 2: Aggregation via Correlation Clustering

```
  input : p.o. matrix $M$ on $V$ and parameter
          $\beta \leq 0.5$
  output: Edge Labeled Graph $G = (V, E)$
1 for ($\forall i \in V,\ \forall j \in V$) do
2     if $(0.5 - \beta) < M(i, j) < (0.5 + \beta)$ then
3         label the edge $(i, j)$ positive
4     else
5         label the edge $(i, j)$ negative
6     end
7 end
8 return $G$
```

Procedure ObtainCCInstance

The labeling of the edges is done based on the pairwise preference between its end-points. The edge connecting a pair (i, j) is labeled +ve if the pairwise preference between i and j suggests that they should be in the same bucket. Ideally, we should have $M[i, j] = M[j, i] = 0.5$ if the pair (i, j) are in the same bucket. Instead, we use the region $[0.5 - \beta, 0.5 + \beta]$ to characterize pairs for whom the corresponding edges must be labeled positive. Procedure " ObtainCCInstance" presents the pseudocode which constructs a correlation clustering instance based on the pairwise preference matrix M. We get a solution to the correlation clustering problem using the powerful technique of Ailon et al. [3] (See Procedure " CorrelationCluster" for details). Note that this approach circumvents the danger that we highlighted with respect to discrepancies in pairwise preferences involving the pivot entity. It simply puts the pivot in a separate bucket and recurses on the remaining entities.

```
  input : Labeled Graph $G = (V, E)$
  output: The set of buckets $\mathbf{B}$
1 $p \leftarrow$ Random pivot entity. $O \leftarrow \emptyset$ and
  $B \leftarrow \{p\}$;
2 for for all entities $v \in V$ do
3     if label$(p, v)$ is positive then
4         add $v$ to $B$
5     else
6         add $v$ to $O$
7     end
8 end
9 Output
  $B \cup CorrelationCluster(G = (O, E(O))$
```

Procedure CorrelationCluster

The next problem is to obtain a total ordering on the set of buckets obtained via correlation clustering. We first collapse the entities inside each bucket into metanodes. We then define an appropriate p.o. matrix on the metanodes as shown in steps 1 and 2 of Procedure " SortHeuristic". We present two different heuristics to obtain a total ordering. The first heuristic is to define appropriate indegree for each metanode which is indicative of how preferred it is over others (higher the value higher is its preference over others) and then sort the metanodes in the decreasing order of their indegrees. This heuristic of sorting based on in-degrees is similar to the one proposed by Coppersmith et al. [8] for ranking players in weighted tournaments. The second heuristic is to call the PIVOT algorithm of [14] on the p.o. matrix of metanodes. Since the correlation clustering is based on minimizing disagreements, the expectation here is that the PIVOT will now return a total order. The pseudocodes in

"Algorithm 2" and Procedures " ObtainCCInstance", "CorrelationCluster", "SortHeuristic" show all the details. We call the correlation clustering algorithm with the sorting heuristic as "SortCC" and correlation clustering followed by PIVOT as "PivotCC".

```
  input  : Set of Buckets $\mathbf{B}$, p.o. matrix $M$
  output: Bucket Order $(\mathbf{B}, \prec)$
1 Collapse each bucket $B_i$ into a meta node $N_i$;
2 Compute $M(N_i, N_j) = \sum_{v \notin N_i} \sum_{u \in N_i} M(u, v)$;
3 Compute $indegree(N_i) = \frac{\sum_{N_j \neq N_i} M(N_i, N_j)}{|N_i| \cdot (|V| - |N_i|)}$;
4 Define $\prec$ on $B$ based on decreasing order of
  indegrees and return $(\mathbf{B}, \prec)$;
```

Procedure SortHeuristic

6 Experimental Evaluation

In this section, we present empirical evaluation of the following algorithms: SortCC, PivotCC, PIVOT, and GAP algorithms. The SortCC, PivotCC, and PIVOT are parameterized by β and the GAP algorithms is parameterized by "num_of_frequencies" [12].

Let $C_{\mathbf{I}}$ denote the p.o. matrix corresponding to the input rankings. If the input is sampled from an unknown underlying bucket order (called *ground truth*), then we denote the p.o. matrix of the ground truth by $C_{\mathbf{G}}$. Let $C_{\mathbf{B}}$ be the output p.o. matrix computed by an algorithm for the bucket order problem. As in [12], we call $|C_{\mathbf{I}} - C_{\mathbf{B}}|$ as the *I-distance* and $|C_{\mathbf{G}} - C_{\mathbf{B}}|$ as the *G-distance* of the output p.o. matrix $C_{\mathbf{B}}$.

6.1 Seriation Problem in Paleontology

A typical fossil discovery database contains details of the fossil remains of different species found across different sites. Let F denote the set of fossil sites and let S denote the set of species whose fossil remains are documented. For every discovered fossil, the database records the site at which it was discovered and the species the fossil belongs to. This in turn can be translated to a 0-1 present/absent matrix of order $|F| \times |S|$ where the (i, j)th entry indicates whether fossil remains of species j were found in site i. The seriation problem is one of biochronology: in the absence of geological evidences and lack of geochronologically datable materials, obtain a temporal order on the sites based purely on the $|F| \times |S|$ present/absent matrix. In this experiment, we consider a dataset based on the data collected on 124 fossil sites in Europe and 139 species. Domain experts have assigned each of the 124 sites to a mammal neogene (MN) class ranging from 3 (oldest) to 17 (youngest). This is indeed a bucket order on the sites. So, one evaluation criteria for the algorithms is how well does the discovered bucket order compare with this ground truth.

Puolämaki et al. [22] developed a Monte Carlo Markov Chain heuristic for generating total orders based on the $|F| \times |S|$ present/absent matrix. The probability of a total order being picked depends on its nearest distance to a valid linear extension of the ground truth bucket order. Closer a total order is to a linear extension, higher is its probability of being sampled. Note the resemblance between this heuristic and our model for sampling in the query complexity problem. They generated about 2000 total orders based on present/absent information for the 124 sites and 139 species. This dataset is popularly referred to as *g10s10* dataset. We conducted empirical evaluation of the different algorithms on the *g10s10* dataset.

Figures 1 and 2 show the the *G-distance* and *I-distance* graphs for the Paleontological dataset respectively. For the PIVOT, SortCC, and PivotCC algorithms, the x-axis corresponds to the labeled β values from 0.05 to 0.25. For the GAP algorithm, x-axis corresponds to its main parameter num_of_frequencies which can range from 1 to 99. We report the best value for the parameter in the following five ranges: $[1 : 19], [20 : 39], [40 : 59], [60 : 79], [80 : 99]$. We follow this convention for the rest of the experiments.

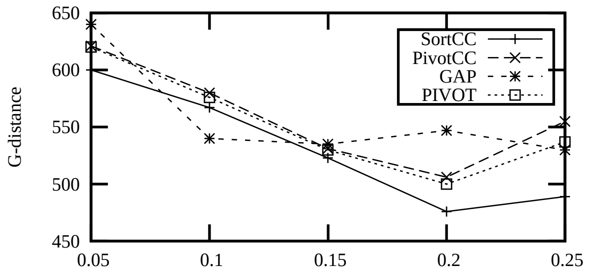

Figure 1: *G-distance* on Paleontological dataset.

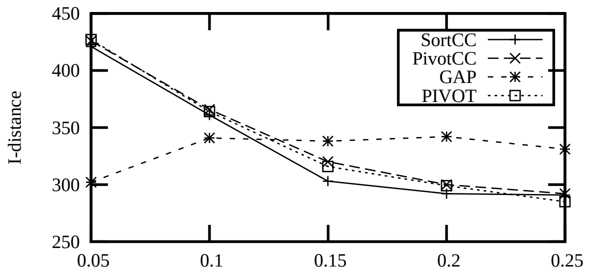

Figure 2: *I-distance* on Paleontological dataset

Consider the *G-distance* which is the real objective of optimization in this application. The SortCC algorithm computed the best solution. The performance of the GAP algorithm was the most inferior. In case of the *I-distance*, there is not much to separate the algorithms in terms of their best output. But, the GAP algorithm has the advantage that it is relatively stable across different settings of num_of_frequencies.

Different settings of the parameters of the algorithms give rise to different results. So, we must have a heuristic to pick a setting which, hopefully, minimizes the distance of the solution to the ground truth. One obvious way is to pick the solution with minimum *I-distance* over different parameter settings. In the case of *g10s10*, it is interesting to note that, for the pivot based algorithms, the minimum *I-distance* also coincides with the minimum *G-distance*. However, for the GAP algorithm, minimum *I-distance* corresponds to a solution with maximum *G-distance*.

6.2 Aggregating the Browsing Patterns of visitors of MSNBC We use the MSNBC dataset used by Feng et al. [12]. It contains the browsing sequences of all the 981818 visitors to the MSNBC website on September 28th 1999. The webpages were classified into 17 categories. The dataset contains, for each visitor, the sequence of the categories of the webpages she visited. They can be viewed as preference orders of the visitors. The goal is to capture the likely orders that visitors are likely to follow across the different categories, such as "front-page"->"news"->"business"->"sports"->"exit". As it is unlikely that there will be one strict ordering, bucket order turns out to be a convenient way of capturing the likelihood of different orderings.

Feng et al. [12] first reported experiments with the MSNBC dataset. But, their experiment is severely limited by a particular feature of their algorithm. We need to describe the main idea of their algorithm to present the limitation in their experiment. The input to their algorithm is a set of total orders. The "median rank" of an entity is the median of its rank across the different total orders in the input. The heuristic used in [11] is to rank the entities based on their median ranks. Feng et al. [12] generalize this heuristic, first by not only considering median rank of an entity, but also its rank at different quantiles. To determine if two entities should belong to the same bucket, they use similarities in their ranks at different quantiles.

Observe that the main idea of capturing the rank of an entity at different quantiles implicitly assumes that the input is given in the form of total orders. In the MSNBC application, this means that each visitor visits pages from each of the 17 categories. However, this is not true. In fact, the number of visitors who visit pages from four or more categories is less than 10%. So, to apply their algorithm in this context, they have the following preprocessing step. Let S be the ordered set of categories visited by a visitor and let S' be the missing categories in the increasing order of category number. They obtain a total order by appending S' at the end of S. It is very important to note that the appended S' introduces erroneous preference statistics for the categories within S'. Moreover, simply relabeling the categories changes the input as considered by their preprocessing step! To limit the effect of errors introduced by the preprocessing step, they consider only those visitors whose sequence has atleast 14 categories. This brings the number of visitors considered in the experiment to just 160 (out of 981818)! This small sample of visitors is unlikely to be representative of the aggregate preferences of the visitors. We correct this anomaly in our experiment.

We need the labels of different categories to help present the experimental results in a meaningful manner. The labels for the categories are: front-page(1), news(2), tech(3), local(4), opinion(5), on-air(6), misc(7), weather(8), msn-news(9), health(10), living(11), business(12), msn-sports(13), sports(14), summary(15), bbs(16), travel(17). Figure 3 presents the experimental results. The first column specifies two parameters: minimum sequence length for a visitor to quality for the experiment, and the number of visitors who qualified. For the GAP algorithm, we reproduce the two bucket orders reported in [12]: median bucket order and bucket order with minimum cost.

Interpretation: Since the GAP algorithm considers just 160 users, its bucket order cannot highlight rare categories like opinions(5) and bbs(16). Other algorithms consistently do this (quite significantly, even for 160 users). Categories like on-air(6), msn-news(9), and msn-sports(13) which are quite popular with many visitors are not reflected high enough in GAP because of confining to just 160 users. Rest of the algorithms do bring out this distinction. The three algorithms, PIVOT, SortCC, and PivotCC, consistently put the categories opinions(5), bbs(16) at the end and the categories front-page(1), msn-news(9), business(12), msn-sports(13), and news(2) at the top. We have manually checked the corresponding p.o. matrices to ensure that these orderings indeed reflect pairwise preferences. Conceptually, this experiment highlights the limitation of the GAP algorithm of requiring total order inputs.

Details	Bucket Order
GAPmin (14,160)	{1,2,12} < {3,4,5,6,7,8,10,11,14,15} < {9,13,16,17}
GAPmed (14,160)	{1,2} < {3,4,6,7,10,11,12,14,15} < {5,8} < {9,13,16,17}
PIVOT (14,160)	1<{2,9,12,13}<15< {3,6,8,9,10,11,14,17}<7<4<5<16
SortCC (14,160)	1 < {2,12,9,15} < {4,10} < {3,6,7,8,11,13,17} < 5 < 16
PivotCC (14,160)	1<{2,12}<15<{4,6,7,9,11,13,14,17}< {3,8,10}<5<16
PIVOT (4,79331)	{9,13} < 1 < {2,3,6,7,8,10,11,12,15,17} < {4,14} < 5 < 16
SortCC (4,79331)	1 < 9 < {2,3,4,6,7,8,10,12,13,14,17} < {11,15,16} < 5
PivotCC (4,79331)	{6,9,13} < 1 < {2,3,7,8,10,11,12,14,15,17}<4<5<16
PIVOT (all users)	{6,9,12,13}<1<{2,7}< {3,4,8,10,11,14,15,17}<5<16
SortCC (all users)	1<9<{2,3,4,6,7,8,10,11,12,14} <{13,15,16}<17< 5
PivotCC (all users)	9<1<{2,4,6,8,10,12,13,14,17}<{3,7,11} <{15,16}<5

Figure 3: Table of bucket orders for different subsets of visitors of the MSNBC portal.

6.3 Experiments on Artificially Generated Data We now present our experiments with artificially generated data.

• *Input Generator:* This module generates the input for testing the algorithms. The input to this module is a tuple $(N, T, b, \delta, B, f1, f2, f3)$ where N specifies the number of entities, T specifies the number of total orders to be generated, b specifies the minimum size of a bucket, δ specifies the bound on the displacement error of the entities, B specifies the number of buckets. Given a specification like this, the module first generates G, the ground truth bucket order consisting of B buckets over N entities in which each bucket is of size at least b. It then generates $f1 \cdot T$ number of linear extensions of G, $f2 \cdot T$ number of total orders whose local error w.r.t. G is at most δ, and $f3 \cdot T$ completely random total orders. The fractions $f1, f2, f3 \geq 0$ are such that $f1 + f2 + f3 = 1.0$. The linear extensions generator is such that it picks each possible linear extension of G with equal probability. This is also true for the total orders with local errors. We use the generator with various combination of values that the tuple $(N, T, b, \delta, B, f1, f2, f3)$ can take.

• *Experimentation:* For each of the dataset, we run all the four algorithms. The randomized algorithms SortCC, PivotCC, and PIVOT are run multiple times. We collected statistics like best solution, median solution etc. The GAP algorithm is run for every possible value of num_of_frequencies.

• *Results:* We report results as follows. For each β value, we report the *G-distance* value of the solution with minimum *I-distance*. This is necessary as a predetermined setting of the parameters may not work well for every instance. For the GAP algorithm, we report the best results for the five ranges as described in Section 6.1.

• *Selection of Cases:* We have generated 180 datasets with the number of entities ranging from 50 to 4000 and for different combinations of the tuple $(N, T, b, \delta, B, f1, f2, f3)$. Due to space considerations we present the experimental results on a select few cases. The chosen cases highlight some of the interesting aspects of the experiments.

6.3.1 Low Noise: $(250, 25, 25, 2, 10, 0.1, 0.80, 0.10)$. This dataset was designed to test the upperbound on query complexity. It contains 25 total orders on 250 entities (upperbound in Section 4 suggests 40) of which 22 have local error. It is similar to an input that would be drawn under local noise setting. All the algorithms were able to consistently discover the ground truth, thus validating the upperbound.

6.3.2 High Noise, Small Displacement: $(250, 250, 25, 2, 10, 0.10, 0.40, 0.50)$ This dataset contains 250 total orders on 250 entities with 50% of them being random. The remaining 50% are total orders with displacement error of 2. This is sufficient for all the algorithms to recover the ground truth. But, they show different behaviour for different parameter settings. With the usual semantics of the x-axis as explained in Section 6.1, the results are presented in Figure 4. The y-axis is on logscale. It can be seen that the GAP algorithm is relatively stable across its different parameter settings. The other three algorithms show sudden deterioration when the β value crosses a threshold. We identify general guidelines for β values for different conditions in the following experiments.

6.3.3 Low Noise, Large Displacement: $(1000, 1000, 1, 20, 100, 0.10, 0.80, 0.10)$ This dataset contains 1000 total orders on 1000 entities. The noise was low with just 10% of them being random. More importantly, the displacement error is large at 20 and the minimum bucket size is low at 1. Figure 5 shows

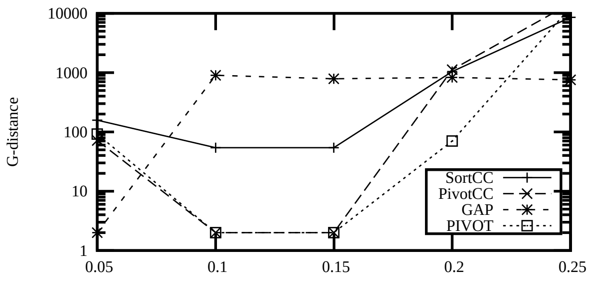

Figure 4: *G-distance* graph for the data used in Section 6.3.2

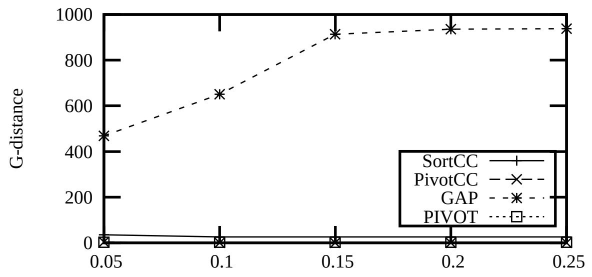

Figure 5: *G-distance* graph for the data used in Section 6.3.3

the graph of the four algorithms for this dataset. The *G-distance* for all the algorithms is quite low considering there are 1000 entities. The pivot based algorithms perform exceedingly well and are even able to recover single entity buckets. But, the GAP algorithm's performance is inferior compared to the other algorithms. The reason that GAP suffers is because it is heavily dependent on median ranks of the entities which get affected because of large displacements.

6.3.4 High Noise, Large Displacement: $(500, 500, 1, 20, 50, 0.10, 0.40, 0.50)$ This dataset contains 50% random total orders and 40% total orders with displacement error of 20 on 500 entities. Figure 6 shows the graph of the four algorithms for this dataset. In this case, we observe the phenomena observed in Figure 4. While the algorithms PIVOT, PivotCC, and SortCC are able to compute solution with very low *G-distance* for low values of β, they show rapid deterioration for higher values. The GAP algorithm on the other hand is stable across different settings, but the best solution it obtains is inferior to the best solutions of other methods.

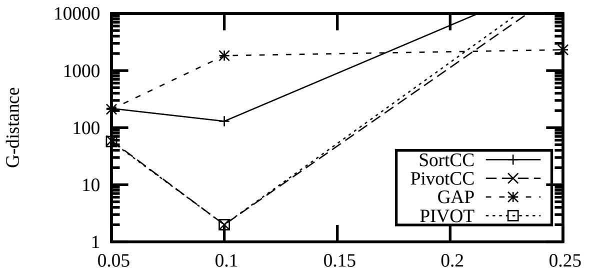

Figure 6: *G-distance* graph for the data used in Section 6.3.4

6.3.5 Summarization of Experimental Results

Based on the above cases presented and an analysis of the results for the other 100+ remaining datasets, we can infer certain patterns in the performance of the different heuristics. They are:

- For the real datasets, the correlation clustering methods SortCC and pivotCC performed better than all others, thus highlighting the potential value of the correlation clustering idea. Across all experiments, PivotCC i.e, correlation clustering for discovering the buckets followed by pivot algorithm to order them performed most robustly. Even the PIVOT algorithm performed particularly well, especially when the number of input total orders was high.

- The pivot based algorithms are capable of approximating *G-distance* in presence of both high noise and large displacements. But, they are not stable across all β values. For noisy inputs, we suggest that the β value should be low: 0.05 to 0.10.

- The GAP algorithm performance is comparable to the pivot based algorithms only when the noise is low. The median (and percentile) rank based heuristic of GAP suffers when the input is noisy or displacements are large. However, one advantage of the GAP algorithm is its relative stability across different settings of num_of_frequencies.

- When the noise was very high, i.e, the percentage of random total orders was more than 70% and the number of input ordering were less (10% of number of entities), even pivot based algorithms performed poorly. This was because of increased number of misclassifications at the time of pivot itself introduced by the global noise.

7 Conclusions

We studied the bucket order aggregation problem from both theoretical and empirical viewpoints. We formalized the notion of query complexity of discovering bucket orders and showed that a small sample is sufficient to discover the bucket order with high probability when the local error is bounded. We presented novel algorithms based on an insight of relating the process of discovering the buckets to the notion of correlation clustering. We presented extensive experimental results to establish the efficacy of our approach. It would be interesting to analyze our correlation clustering heuristics, especially SortCC, from the point of view of approximation ratio.

References

[1] Rakesh Agrawal, Alan Halverson, Krishnaram Kenthapadi, Nina Mishra, and Panayiotis Tsaparas. Generating labels from clicks. In *Proceedings of the International Conference on Web Searching and Web Data Mining (WSDM)*, pages 172–181, 2009.

[2] Nir Ailon. Aggregation of partial rankings, -ratings and top- lists. *Algorithmica*, 57(2), 2010.

[3] Nir Ailon, Moses Charikar, and Alantha Newman. Aggregating inconsistent information: Ranking and clustering. *Journal of the ACM (JACM)*, 55(5), 2008.

[4] Javed A. Aslam and Mark H. Montague. Models for metasearch. In *Proc. of the ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 275–284, 2001.

[5] Nikhil Bansal, Avrim Blum, and Shuchi Chawla. Correlation clustering. *Machine Learning*, 56(1-3):89–113, 2004.

[6] Mark Braverman and Elchanan Mossel. Sorting without resampling. In *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 268–276, 2008.

[7] William W. Cohen, Robert E. Schapire, and Yoram Singer. Learning to order things. *Journal of Artificial Intelligence Research (JAIR)*, 10:243–270, 1999.

[8] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. *ACM Transactions on Algorithms*, 6(3), 2010.

[9] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proc. of the International Conference on World Wide Web (WWW)*, pages 613–622, 2001.

[10] Ronald Fagin, Ravi Kumar, Mohammad Mahdian, D. Sivakumar, and Erik Vee. Comparing and aggregating rankings with ties. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 47–58, 2004.

[11] Ronald Fagin, Ravi Kumar, and D. Sivakumar. Efficient similarity search and classification via rank aggregation. In *SIGMOD Conference*, pages 301–312, 2003.

[12] Jianlin Feng, Qiong Fang, and Wilfred Ng. Discovering bucket orders from full rankings. In *SIGMOD Conference*, pages 55–66, 2008.

[13] M. Fortelius, A. Gionis, J. Jernvall, and H. Mannila. Spectral ordering and biochronology of european fossil mammals. *Paleobiology*, pages 206–214, 2006.

[14] Aristides Gionis, Heikki Mannila, Kai Puolamäki, and Antti Ukkonen. Algorithms for discovering bucket orders from data. In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data mining (KDD)*, pages 561–566, 2006.

[15] U. Halekoh and W. Vach. A bayesian approach to seriation problems in archeology. *Computational Statistics and Data Analysis*, 45(33):651–673, 2004.

[16] Richard Karp and Robert Klienberg. Noisy binary search and its applications. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 881–890, 2007.

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

[18] M. Kendall. *Rank Correlation Methods*. Griffin, London, 4 edition, 1970.

[19] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *ACM Symposium on Theory of Computing (STOC)*, pages 95–103, 2007.

[20] I. Miklos, I. Somodi, and J. Podani. Rearrangement of ecological matrices via markov chain monte carlo simulation. *Ecology*, 86:3398–3410, 2005.

[21] Michael Mitzenmacher and Eli Upfal. *Probability and Computing*. Cambridge University Press, UK, 2005.

[22] M. Puolamäki, M. Fortelius, and H. Mannila. Seriation in paleontological data matrices via markov chain monte carlo methods. *PLoS Computational Biology*, 2(2), 2006.

[23] H. Young. Condorcet's theory of voting. *American Political Science Review*, 82:1231–1244, 1988.

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