TKK Dissertations in Information and Computer Science
Espoo 2008 TKK-ICS-D4

ALGORITHMS FOR FINDING ORDERS AND ANALYZING SETS OF CHAINS

Antti Ukkonen

Dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the Faculty of Information and Natural Sciences for public examination and debate in Auditorium TU1 at Helsinki University of Technology (Espoo, Finland) on the 4th of June, 2008, at 12 noon.

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FIN-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358-9-451 1
Fax +358-9-451 3369
E-mail: series@ics.tkk.fi

ISBN 978-951-22-9373-5 (printed version)
ISBN 978-951-22-9374-2 (electronic version)
ISSN 1797-5050 (Print)
ISSN 1797-5069 (Online)
URL: http://lib.tkk.fi/Diss/2008/isbn-9789512293742

Multiprint Oy
Espoo 2008

Ukkonen, A. (2008): **Algorithms for finding orders and analyzing sets of chains**. Doctoral thesis, Helsinki University of Technology, Dissertations in Information and Computer Science, TKK-ICS-D4, Espoo, Finland.

Keywords: data analysis, ranking, approximation algorithm, clustering, visualization, dimensionality reduction, MCMC, randomization testing

Abstract

Rankings of items are a useful concept in a variety of applications, such as clickstream analysis, some voting methods, bioinformatics, and other fields of science such as paleontology. This thesis addresses two problems related to such data. The first problem is about finding orders, while the second one is about analyzing sets of orders.

We address two different tasks in the problem of finding orders. We can find orders either by computing an aggregate of a set of known orders, or by constructing an order for a previously unordered data set. For the first task we show that bucket orders, a subclass of partial orders, are a useful structure for summarizing sets of orders. We formulate an optimization problem for finding such partial orders, show that it is NP-hard, and give an efficient randomized algorithm for finding approximate solutions to it. Moreover, we show that the expected cost of a solution found by the randomized algorithm differs from the optimal solution only by a constant factor. For the second approach we propose a simple method for sampling orders for 0–1 vectors that is based on the consecutive ones property.

For analyzing orders, we discuss three different methods. First, we give an algorithm for clustering sets of orders. The algorithm is a variant of Lloyd's iteration for solving the k-means problem. We also give two different approaches for mapping orders to vectors in a high-dimensional Euclidean space. These mappings are used on one hand for clustering, and on the other hand for creating two dimensional visualizations (scatterplots) for sets of orders. Finally, we discuss randomization testing in case of orders. To this end we propose an MCMC algorithm for creating random sets of orders that preserve certain well defined properties of a given set of orders. The random data sets can be used to assess the statistical significance of the results obtained e.g. by clustering.

Tiivistelmä

Alkioiden järjestykset ovat hyödyllinen käsite useissa sovelluksissa, kuten verkkopalvelun lokien analysoinnissa, tietyissä äänestysjärjestelmissä, bioinformatiikassa ja muissa luonnontieteissä, kuten esimerkiksi paleontologiassa. Tässä työssä tarkastellaan kahta järjestyksiin liittyvää ongelmaa. Ensimmäinen ongelma koskee järjestysten etsintää ja toinen järjestysjoukkojen analysointia.

Tarkastelemme kahta eri järjestysten etsintää koskevaa tapausta. Järjestyksiä voidaan etsiä joko laskemalla yhteenveto potentiaalisesti suuresta joukosta tunnettuja järjestyksiä, tai konstruoimalla järjestys aiemmin järjestämättömälle aineistolle. Ensimmäiseen tapaukseen liittyen esitämme, että sankojärjestykset, eräs osittainjärjestysten luokka, ovat hyödyllinen tapa esittää järjestysjoukkoja. Muotoilemme optimointitehtävän kyseisten osittainjärjestysten löytämiseksi, osoitamme että se on NP- kova ja esitämme tehokkaan satunnaisalgoritmin jolla voidaan laskea likimääräisiä ratkaisuja. Lisäksi näytämme, että satunnaisalgoritmin palauttaman ratkaisun kustannuksen odotusarvo on vakiokertoimen päässä optimiratkaisun kustannuksesta. Toiseen tapaukseen liittyen esitämme yksinkertaisen peräkkäisten ykkösten ominaisuuteen perustuvan menetelmän, jolla voidaan poimia järjestyksiä 0–1 vektoreille.

Järjestysjoukkojen analysointia varten käsittelemme kolmea eri menetelmää. Ensin esitämme algoritmin järjestysjoukkojen klusterointia varten. Algoritmi on variantti Lloyd:in iteraatiosta k-means onglman ratkaisemiseen. Annamme myös kaksi eri tapaa kuvata järjestyksiä korkeaulotteiseen euklidiseen avaruuteen. Näitä kuvauksia voidaan toisaalta soveltaa klusterointiin, ja toisaalta niiden avulla voidaan laatia kaksiulotteisia visualisointeja (sirontakuvioita) järjestysjoukoille. Lisäksi käsittelemme satunnaistustestausta järjestysten tapauksessa. Tätä tarkoitusta varten esitämme MCMC algoritmin jolla voidaan luoda satunnaisia järjestysjoukkoja, jotka säilyttävät annetun järjestysjoukon tietyt, hyvin määritellyt ominaisuudet. Satunnaisia aineistoja voidaan käyttää arvioimaan esimerkiksi klusterointimenetelmillä saatujen tulosten tilastollista merkitsevyyttä.

Contents

Preface

This dissertation was written while I was working for the Pattern Discovery group at the Laboratory of Computer and Information Science, now part of the Department of Information and Computer Science, at Helsinki University of Technology. I have also been affiliated with Helsinki Institute for Information Technology and the HeCSE graduate school during my studies. Many thanks to these organizations and their personnel for providing me with the necessary resources to complete my work.

I have not been alone. My advisor Professor Heikki Mannila has been an invaluable guide during the past few years, and without his support this thesis would not exist. Much of the same can be said about Kai Puolamäki. Both work and non-work related discussions with him have made the daily routines a lot more interesting. And thanks to my co-author Aristides Gionis I had the opportunity to visit Yahoo! Research Barcelona for one summer, which was a very entertaining and educational experience. Gentlemen, thank you very much.

In addition, I must thank Professor Dimitrios Gunopulos from the University of California at Riverside and Professor Luc De Raedt from Katholieke Universiteit Leuven for serving as the pre-examiners of this dissertation.

I also want to thank my colleagues Hannes Heikinheimo, Sami Hanhijärvi and Gemma Garriga for being there to listen to me complain about different things of varying (usually limited) importance. Particularly I am grateful to Hannes because of his continuous support ever since our undergraduate years. Also other members of the Pattern group, Jaakko Hollmén, Jouni Seppänen, Nikolaj Tatti, Mikko Korpela, Janne Toivola, Niko Vuokko and Markus Ojala deserve my sincere thanks.

I am grateful to all of my friends, especially Aleksi, Johan, Tuomas and Vili, who gave me also other stuff to think about besides computer science. Finally, I have to thank my brother, father, and mother, for everything.

Otaniemi, May 2008

Antti Ukkonen

Chapter 1

Introduction

This thesis is about orders. It is not about the kind of orders one places at, say, a restaurant or online bookstore. Neither is it about orders given by superiors to subordinates. This thesis is about orders that reflect a certain mutual relationship between some objects in a finite set. Given two objects from the set, an order tells us which one of the two objects precedes the other one, or that this information is not available. For example, if the objects in the set are movies released in 2008, an order might place movie A before movie B because a larger number of people have seen movie A than movie B. There can be several orders reflecting different properties of the same set of objects. Another order might place movie B before movie A because movie B has a higher rating at the Internet Movie Database[1] than movie A. An order is thus always related to some property of the objects that we are ordering.

More precisely, this thesis is mostly about chains. A *chain* is a special kind of order. Simply put, a chain is a list of objects that belong to the set we are interested in. It does usually not contain all objects in the set, but only concerns a *subset* of it. For each pair of objects that both belong to this subset, the chain can tell us which of the two objects precedes the other one. But for any pair where one or both of the objects do not belong to the subset, the chain can not tell us which of the objects comes first. A chain is thus a *partial order.* In the special case where the subset happens to be equal to the entire set of objects, we say the chain is a *total order.*

Finally, this thesis is about algorithms both for *finding* and *analyzing* orders. By finding we refer to the problem of using a given data set for constructing a previously unknown order for the set of objects. For example, in a certain paleontological application the objects in the set are geographical locations, and to each location is associated a list of fossils that were found there. We can use

[1] `http://www.imdb.com`

this information to construct a temporal order for the locations. In this case, given two locations the order tells us which one of them is older than the other. Note that the problem of finding an order is not to be confused with sorting, where the task is simply to arrange the set of objects according to some known order.

By analyzing we refer to the problem of gaining previously unknown information from a data set that contains many, possibly up to thousands of different orders on the set of objects. Continuing with the movie example, maybe we have asked thousand people to order some movies they have seen recently from best to worst, and want to divide the people to groups so that respondents with similar preferences are put into the same group, while respondents with different preferences are put into different groups. Of course the problem of finding an order can be seen as an analysis problem as well. Given the thousand responses we might want to combine them into a single previously unknown order that reflects the preferences of the entire set of respondents.

Motivation Orders are an interesting research topic from a computational point of view. For instance, the relatively mundane task of computing the mean, which is trivial if we are analyzing numbers (or vectors), becomes a very challenging problem with orders. What is the mean of a set of orders? Or, what is a good clustering algorithm for sets of orders? Clustering is a well known problem and can be solved efficiently for inputs that consist of numerical data only. Also, many techniques exist for visualizing sets of high-dimensional vectors. How should we visualize sets of orders? We address all these questions in this thesis.

Moreover, the problem of finding a previously unknown order is probably even more difficult. There are $n!$ possible solutions when the set we are studying contains n objects. Even for seemingly small sets, say $n = 20$, there are already about $2.4 \cdot 10^{18}$ different ways of ordering the objects. Thus efficient algorithms are needed. If we would blindly go through every possible order and check if it indeed is the “best” solution, the time required for computing the answer increases to unreachable magnitudes very quickly. For example, if our computer is able to check one million orders for the 20 objects per second, the computation would take more than seventy-five thousand years to complete. As a consequence, finding an exact solution to a problem that involves orders is probably not possible in many interesting cases. This calls for algorithms that can quickly find a solution that may not be perfect, but is “good enough”.

We can also motivate the research from a practical point of view. Orders are in some applications a natural approach to representing the data. In surveys people are usually asked to evaluate some objects on a fixed scale. The five-star system for rating movies is a typical example of this. Suppose that Alice and Bob have both seen movies A and B, and that Alice has given movie A one star and movie B three stars, while Bob gave movie A three stars and movie B five stars. If we compare the preferences of Alice and Bob based on the ratings alone, they seem fairly different: one star versus three stars in case of movie A, and three stars versus five stars in case of movie B. However, Alice and Bob would still agree that movie B is better than movie A.

Analyzing the ratings as such is problematic because Alice, Bob, You, and me tend to have different scales for evaluating the movies. In the previous example Alice may be a very critical viewer while Bob is less demanding. When we compare the orders to which Alice and Bob place the movies, we only make use of information that is independent of the absolute ratings, and hence the individual scales of the respondents.

In addition to preference surveys, orders occur in other application domains as well:

1. *Clickstreams*: Web server log files can be used to construct sequences in which a user (identified by a HTTP session or IP address) browsed through different pages on a web site. These sequences can be viewed as orders, provided they are modified so that each sequence does not contain multiple instances of the same page.

2. *Voting systems*: In some voting systems a vote is an order of set of the candidates or a subset thereof.

3. *Bioinformatics*: Gene expression data has become abundant with the popularity of microarray technology. This data is usually produced in an experiment where the expression level of a number of genes is measured under different environmental conditions. For each gene we have a list of expression values, one for each condition. Instead of using the expression values directly, we can sort the conditions in increasing (or decreasing) level of expression for each gene and use this order for further analysis.

Contributions of this thesis. We summarize the results of this thesis in the following points:

- In Chapter 3 we propose Bucket orders as an alternative model for rank aggregation. We give an algorithm motivated by [ACN05] for finding bucket orders. We show that this algorithm has a expected constant factor approximation guarantee.

- In Chapter 4 we give a simple variant of Lloyd's algorithm for clustering chains. We also present two techniques for representing chains as high dimensional vectors. We report results on experiments that demonstrate that the algorithm can find interesting clusters from real data sets. We also compare our algorithm with those of [KA06].

- In Chapter 5 we demonstrate empirically how the high dimensional representations discussed in Chapter 4 can be used to create informative two-dimensional scatterplots of sets of chains.
- In Chapter 6 we give a MCMC based algorithm for sampling random sets of chains that share certain properties of an initial set of chains. We demonstrate experimentally that the random sets of chains can be used for randomization testing.
- In Chapter 7 we present a method for finding a partial order for a set of 0–1 vectors. We also discuss a simple test for assessing the "orderability" of a set of 0–1 vectors. We demonstrate empirically that the method can find meaningful orders from real data sets.

Some of the results have appeared in previous publications of the author. This thesis is an attempt to discuss the previous work in more detail using a common terminology and notation. The algorithm for finding Bucket Orders and the theorem on the approximation guarantee appeared without proofs in [GMPU06]. In Chapter 3 we present the proof for the approximation ratio and discuss alternative algorithms in more detail. Chapter 5 is mostly based on [Ukk07]. Also, some of the concepts discussed in Section 4.3 are very briefly presented in [Ukk07]. The claim of Theorem 4.3.2 appeared in [Ukk07] but again the proof was omitted due to space constraints. The MCMC sampling algorithm of Chapter 6 was originally discussed in [UM07]. The problem of finding a partial order for 0–1 vectors appeared first in [UFM05]. This discussion is extended in Chapter 7.

Chapter 2

Preliminaries

Raw data as such is rarely very informative. In order to make useful conclusions based on data one typically needs to summarize the information it contains somehow. For example, given a cloud of points in an n-dimensional space we could compute the center of the cloud and the expected distance of a point from this center. If the data happens to consist of real valued vectors, computing these estimators can usually be done in a computationally efficient manner.

The same approach can be taken also when the data contains *rankings* of some set of items. However, it is no longer obvious what a “mean” of a set of rankings would look like, or how to compute it efficiently. For example, suppose we have conducted a survey about people’s preferences towards recently released movies. Each respondent has named a number of movies and ranked them best to worst according to his/her preferences. Such a ranking is a *chain* on the set of movies, and the data contains one chain for each respondent. Given this data the goal is to compute a global ranking of all movies mentioned by the respondents, so that this ranking reflects the preferences of the entire population. Intuitively this means that if a movie was ranked high by many of the respondents, it’s position in the global ranking should be higher than the position of a movie that was ranked low in many of the responses.

In this chapter we discuss the background of the so called rank aggregation problem outlined above. Many definitions that are needed throughout this thesis are given here. We also discuss data sets that we use in later chapters when performing experiments. In Section 2.2 we will give a formal definition of rank aggregation, discuss its complexity and describe some possible solutions. We also consider vertex ordering problems in graphs and their connections to rank aggregation.

2.1 Basic definitions

In this section we define some of the basic concepts this thesis deals with. Here we also describe a model for generating random data sets and four real data sets that are used in the experiments in later chapters. Let M be a finite set of items that we are ordering. Elements of M are typically denoted by letters $u, v, w, \ldots$, while orders are denoted by small Greek letters π and τ.

Order relations and chains

A *partial order* π on M is a subset of $M \times M$, i.e., it is a binary relation on M. We require that π is asymmetric and transitive. If the pair (u, v) belongs to π, we say that u precedes v according to π. If the pair (u, v) does not belong to π, we say the items u and v are unordered by π.

If for all $u, v \in M$ we have either $(u, v) \in \pi$ or $(v, u) \in \pi$, we say π is a *total order*. We say a total order τ is compatible with a partial order π if all (u, v) that belong to π also belong to τ. A total order that is compatible with a partial order π is called a *linear extension* of π. The set of all linear extensions of π is denoted by $\mathcal{E}(\pi)$. In the examples we will denote total orders by comma separated lists of items in parenthesis, i.e., $\pi = (2, 3, 1)$ denotes the total order where item 2 precedes items 1 and 3, while item 3 precedes item 1.

Next we define chains and bucket orders. These are two special cases of partial orders that play an important part in this thesis.

Chains: The partial order π on M is a *chain*, if for some $M' \subseteq M$ π defines a total order, and we have $\{M' \times (M \setminus M') \cup (M \setminus M') \times M'\} \cap \pi = \emptyset$.

Bucket orders: A bucket order $\mathcal{B}$ consists of an ordered partition of M, i.e., a sequence $M_1, M_2, \ldots, M_b$ of b disjoint subsets of M such that $\bigcup_i M_i = M$. We write $\mathcal{B} = \{M_1 \prec M_2 \prec \cdots \prec M_b\}$. A bucket order is thus a partial order on M. The item u precedes the item v, or equivalently $(u, v) \in \mathcal{B}$, if and only if $u \in M_i$ and $v \in M_j$ and $i < j$.

A generating model for sets of chains

In this section we discuss a generating model for sets of chains. This model is used in some of the empirical sections to evaluate the proposed algorithms. Using the model we can create artificial data sets of which certain properties are known. This allows us to create controlled experiments for investigating the effect of different parameters to the performance of the algorithms. Note that the algorithms themselves do not make any use of this model.

We explain the model with an example related to movie preferences, but the same model can also be used in other application domains. Recall, that M is the set of items that are ordered by the chains. In this case, we let M be the set of

	SUSHI	MLENS	DUBLIN	MSNBC
n	5000	2191	5000	5000
m	100	207	12	17
min. l	10	6	4	6
avg. l	10	13.3	4.8	6.5
max. l	10	15	6	8

Table 2.1: Key statistics for different real data sets. Number of chains: n, number of items m, length of a chain l.

movies that were released in 2008. Each chain will be a list of movie titles. The model generates a set of such chains.

We assume the chains are generated by a population of individuals, moviegoers in this case. This population can be segmented to groups $j = 1, \ldots, k$ so that members of a certain group j have similar preferences regarding the movies. These common preferences are modeled by a group specific partial order π_j on M.

Suppose an individual i from group j has access to all of M, i.e., has seen all movies that came out in 2008. Then i can in theory specify a total order τ_i on the items according to his preferences. As i belongs to group j, we assume τ_i is a linear extension of π_j. However, since in general an individual can only evaluate a subset of the items (those movies he has seen), i can specify only the chain τ_i' that covers the subset known to him, but ranks those as τ_i would.

The generating model works as follows: initialize the model by picking k partial orders π_j, $j = 1, \ldots, k$, on M. Then for each individual i, first pick one π_j, then pick one linear extension τ_i of π_j, and finally pick a subset of l items and create the chain by projecting τ_i on this subset. In each case we use a uniform distribution on the respective sample space.

To simplify matters computationally, we select the π_js from a restricted class of partial orders that we call bucket orders (see Section 2.1 above). One parameter of the model is thus the number of buckets in a bucket order, denoted b. The other parameters are the number of groups, denoted by k, the number of items in each chain, denoted by l, and the number of chains generated by each component.

Data sets

In addition to artificial data generated using the model given above, we will use the following four real data sets in the empirical sections of this thesis. All data sets are based on publicly available sources. Their key statistics are summarized in Table 2.1.

Voting data In a simple voting system each voter gets to place one vote for one candidate. The candidate with the highest number of votes is the winner of the election. However, in some instances more complex methods are used. From the perspective of this paper, voting systems where the voters rank the candidates in order of preference are of interest. This data can be used to segment the population of voters based on their views of the candidates, for example.

We made experiments with voting data of this type from the 2002 general election held in Ireland.[1] The data (DUBLIN) contains votes in form of chains of 12 candidates from the electoral district of northern Dublin. We selected a subset of the votes that ranked at least 4 and at most 6 candidates. This left us with 17737 votes in total. Of this we picked a random subset of 5000 votes.

Clickstream data By clickstream data we mean the sequence in which a user has visited different sections of a web site. The data set (MSNBC) we use was collected at `msnbc.com` on September 28th 1999.[2] For each user the data gives the categories the user visited and the order in which this took place. We pruned the original data to create chains by using only the first occurrence of a category in each sequence. In total there are 17 categories. We selected a subset of the users whose chain had at least 6 and at most 8 different categories. The total number of chains left was 14598. Also in this case we used a random subset of 5000 chains.

Movie preference data The MovieLens data[3] was originally collected by the GroupLens research group at University of Minnesota. It contains 10^6 ratings for about 3900 movies from over 6000 users. The ratings are given on a scale of 1-5.

Before turning the ratings into chains we preprocess the data as follows. First we discard movies that have been ranked by less than 1000 users. This is done to reduce the number of different movies to 207. As many movies have been seen by only very few users the data does not contain enough information about their relation to the other movies. Next we prune users who have not used the entire scale of five stars in their ratings. This way the resulting partial rankings are more useful as they all reflect the entire preference spectrum from "very bad" to "excellent". This leaves us with 2191 users. For each user we create a chain by picking a sample of movies at random, so that at most three movies with the same number of stars are included in the sample, and order them according to the number of stars, with better movies appearing before the worse ones. The mutual order between two movies with the same number of stars is arbitrary. We call the resulting data set MLENS.

[1] `http://www.dublincountyreturningofficer.com/` (21.1.2008)
[2] `http://kdd.ics.uci.edu/databases/msnbc/` (21.1.2008)
[3] `http://www.movielens.org` (21.1.2008)

Sushi preference data The SUSHI data set is from a Japanese survey that studied people's preferences towards different type of sushi[4]. The number of different sushi flavors was 100, of which each respondent was asked to rank a randomly chosen subset of size ten in order of preference. The number of respondents in this data set is 5000.

2.2 Rank aggregation

We now turn our attention to the problem of computing a "mean" for a set of permutations. This is an interesting problem itself, but it also is relevant background information for the algorithm discussed in the next chapter.

Given n permutations of a fixed set M (e.g., n permutations of $|M|$ movie titles), the *rank aggregation* problem is defined as follows:

Problem (RANK-AGGREGATION) Let $D = \{\pi_1, \pi_2, \ldots, \pi_n\}$ be a set of permutations of the set M. Find the permutation π^*, such that

$$\pi^* = \arg\min_{\pi} \sum_{i=1}^{n} d(\pi, \pi_i), \tag{2.1}$$

where $d(\pi, \pi_i)$ is a distance between π and π_i.

Clearly this definition is similar to the case of computing the center of a set of points in a vector space. In practice we want to summarize the collection of permutations with one permutation that disagrees as little with the permutations in the collection as possible. Here disagreement is measured by a distance function. Two commonly used distance measures for permutations are Kendall's tau and Spearman's footrule. Let π and π' be two permutations of the set M.

Kendall's tau (or the Kendall distance) d_K between permutations π and π' is defined as

$$d_K(\pi, \pi') = \sum_{i \in M} \sum_{j \in M} I\{i \prec_{\pi} j \wedge j \prec_{\pi'} i\},$$

where $i \prec_{\pi} j$ denotes that item $i \in M$ precedes item $j \in M$ in the permutation π, and $I\{\cdot\}$ is the indicator function, i.e., $I\{X\} = 1$ when X is *true* and $I\{X\} = 0$ when X is *false*.

Intuitively the Kendall distance between two permutations is the number of pairs (i, j) such that the permutations disagree on the order of i and j. It is also the number of swaps made by the bubble-sort [CLRS01] sorting algorithm if either π or π' is the natural ordering of the items.

[4]`http://www.kamishima.net/sushi` (22.1.2008)

Spearman's footrule (or the footrule distance) d_S between permutations π and π' is defined as

$$d_S(\pi, \pi') = \sum_{i \in M} |\pi(i) - \pi'(i)|,$$

where $\pi(i)$ denotes the position of the item i in π. (E.g., if i appears as the first item in π, we have $\pi(i) = 1$.)

Unlike Kendall distance, the footrule distance does not concern the pairwise relations between items, but their actual positions in the permutations. Even though this is a fundamental difference between the distance functions, the measures are closely related, as shown by the following theorem from [DG77].

Theorem 2.2.1 *[DG77] For any two permutations π and π', we have*

$$d_K(\pi, \pi') \leq d_S(\pi, \pi') \leq 2d_K(\pi, \pi').$$

Note that it is also fairly easy to construct cases where the two distance measures give inconsistent results, that is $d_S(\pi, \pi_1) < d_S(\pi, \pi_2)$ and $d_K(\pi, \pi_1) > d_K(\pi, \pi_2)$. For example, let

$$\begin{aligned} \pi &= [1, 2, 3, 4, 5, 6], \\ \pi_1 &= [3, 2, 1, 6, 5, 4], \\ \pi_2 &= [6, 1, 2, 3, 4, 5]. \end{aligned}$$

Now we have $d_S(\pi, \pi_1) = 8$ and $d_S(\pi, \pi_2) = 10$, meaning that π_1 is closer to π in terms of the footrule distance. However, $d_K(\pi, \pi_1) = 6$ and $d_K(\pi, \pi_2) = 5$, which puts π_2 closer to π. This example suggests that the footrule distance is in some sense more sensitive to "outliers" (item 6 in π_2) than the Kendall distance. Hence, Theorem 2.2.1 does not imply that d_K and d_S would give the same solution for Equation 2.1.

It is easy to see that the footrule distance can be computed in time linear in the size of M. Computing the Kendall distance by iterating over all possible (i, j) pairs is obviously quadratic in the size of M, but it is possible to improve the complexity to $O(|M| \log |M|)$ for instance by using an algorithm similar to mergesort.

Rank aggregation has turned out to be a computationally interesting problem. It has been shown that finding the π^* defined by Equation 2.1 is NP-hard when $d = d_K$ for $n \geq 4$ [DKNS01]. However, when $d = d_S$, it is easy to construct a weighted bipartite graph $G = (V_1, V_2, E, W)$, so that the optimal aggregation is given by the min-cost perfect matching in G [DKNS]. The construction is simple: let the vertices in V_1 correspond to the items in M and the vertices in V_2 correspond to all possible positions, i.e., $V_2 = \{1, 2, \ldots, |M|\}$. The weight

$W(u, j)$ for all $u \in M$ and $j \in V_2$ is defined

$$W(u, j) = \sum_{i=1}^{n} |j - \pi_i(u)|,$$

which is simply the cost of putting item u to position j in the aggregate permutation. In the perfect matching each vertex of V_1 (an item) is matched with a vertex of V_2 (a position). Hence we can represent the matching as a permutation $\pi^* : V_1 \to V_2$. Since π^* is a minimum cost perfect matching, it is defined by

$$\pi^* = \arg\min_{\pi} \sum_{u \in M} W(u, \pi(u)),$$

which is exactly the same as Equation 2.1 when $d = d_S$. [DKNS]

Constructing the graph G takes time $O(n|M|^2)$ and computing the matching with a simple algorithm is of order $O(|M|^3 \log |M|)$ [KT06]. Using a more sophisticated approach the problem can be solved in $O(|M|^2)$ expected time [SSW05]. Hence, the footrule distance leads to a problem that belongs to **P**, whereas the Kendall distance seems to be hard. This may be somewhat surprising given the result of Theorem 2.2.1 and that the complexity of computing the distances d_S and d_K differs only by a factor of order $O(\log |M|)$. Note that this construction no longer works if the input consists of chains instead of permutations.

Connections to voting theory

Combining individual preferences is a key problem in group decision making, of which elections are probably the most common example. Voting systems have been studied extensively already for over two centuries. The oldest and probably still the most relevant approaches are those of *Condorcet* and *Borda*. In both cases a *vote* is a permutation of the set M of all candidates (or possibly a subset thereof) and the objective is to determine which of the candidates is the winner of the election given the set D of votes. What differs is the definition of a winning candidate.

In the Condorcet method (see Definition 9.2 in [Mou91]) the candidate c^* is the winner if for all $c' \neq c^*$

$$|\{\pi \in D : c^* \prec_\pi c'\}| > |\{\pi \in D : c' \prec_\pi c^*\}|.$$

Less formally, c^* is the candidate that has the most "wins" in pairwise comparisons of the candidates. It is easy to construct an example where there is no unique Condorcet winner. Let the votes be $\pi_1 = [c_1, c_2, c_3]$, $\pi_2 = [c_3, c_1, c_2]$, and $c_3 = [c_2, c_3, c_1]$. This example shows what is known as *Condorcet's paradox*: a majority prefers c_1 to c_2 (votes π_1 and π_2), another majority prefers c_2 to c_3

(votes π_1 and π_3) while a third majority prefers c_3 to c_1 (votes π_2 and π_3), all in the same set of votes.

The method proposed by Borda, also known as the Borda Count (see Definition 9.1 in [Mou91]), is a positional voting method. Each candidate $c \in M$ is given a score that depends on the positions in which c appears in the votes. The winner c^* of the election is defined using the Borda count as follows

$$c^* = \arg\max_c \sum_{\pi \in D} s(\pi(c)),$$

where s is a function from positions to scores. Thus, c^* is the candidate with the highest sum of scores. A common choice is to have $s(x) = |M| - x$, i.e., the score of a candidate c in vote π is the number of other candidates that are placed after c in vote π.

Note that the methods do not necessarily select the candidate ranked first in a majority of the votes. However, the Borda method reduces to simple majority voting if we have $s(x) = 1$ when $x = 1$ and $s(x) = 0$ when $x \neq 1$.

2.3 Vertex ordering problems

We can formulate an instance of the rank aggregation problem as an instance of a vertex ordering problem in directed graphs. Let $G_u = (V, E, C)$ be a *weighted, directed graph.* To each edge $(i, j) \in E$ is associated a cost $C(i, j)$. An ordering of the vertices V is a mapping $\pi : V \to \mathbb{N}$. The problem we consider here is about finding the *minimum weight feedback-arc set* (MIN-FAS), which can be formulated as an ordering problem.

Problem (MIN-FAS) Given $G = (V, E, C)$, find a subset E' of E of minimum weight such that $E \setminus E'$ is acyclic. Edges in E' are the feedback arcs. The problem can be formulated as the task of finding a permutation π^* of V, defined as

$$\pi^* = \arg\min_\pi \sum_{(u,v) \in E} I\{v \prec_\pi u\} C(u, v).$$

In less formal terms, we want to arrange the vertices left-to-right on a line so that the sum of the weights of the edges that point to the left (to the "wrong" direction) is minimized.

The MIN-FAS problem is known to be NP-complete for general graphs (see Problem GT8 in [GJ79]), and also for tournaments, as shown recently in [Alo06]. We also note that MIN-FAS is very similar to the MAXIMUM-ACYCLIC-SUBGRAPH problem, where the input is also a directed graph $G = (V, E)$, and the task is to find a subset of E that is acyclic and of maximum size (of all acyclic subsets of

E). Indeed, an optimal solution for the other yields an optimal solution for the other one.

A special case of the MIN-FAS problem is where G is a complete directed graph with a weighted edge from u to v and v to u for all vertices u and v. This special case is interesting from the point of view of rank aggregation, as it turns out that the RANK-AGGREGATION problem when using the Kendall distance is the same as the MIN-FAS problem in G. An instance of the rank aggregation problem can be cast into a MIN-FAS instance in G by letting each item correspond to a vertex, and having $C(u,v) = |\{\pi_i \in D : u \prec_{\pi_i} v\}|$. To see this, consider Equation 2.1 with $d = d_K$. We have:

$$\begin{aligned}
\pi^* &= \arg\min_{\pi} \sum_{i=1}^{n} d_K(\pi, \pi_i) \\
&= \arg\min_{\pi} \sum_{i=1}^{n} \sum_{u \in M} \sum_{v \in M} I\{u \prec_{\pi} v \wedge v \prec_{\pi_i} u\} \\
&= \arg\min_{\pi} \sum_{u \in M} \sum_{v \in M} I\{u \prec_{\pi} v\} \sum_{i=1}^{n} I\{v \prec_{\pi_i} u\} \\
&= \arg\min_{\pi} \sum_{u \in M} \sum_{v \in M} I\{u \prec_{\pi} v\} |\{\pi_i \in D : v \prec_{\pi_i} u\}| \\
&= \arg\min_{\pi} \sum_{(u,v) \in E} I\{u \prec_{\pi} v\} C(v,u).
\end{aligned}$$

Above the third equality is obtained by seeing that the second indicator function merely counts the number of permutations in D where v precedes u. The last equality follows from the fact that E contains both (u,v) and (v,u) for all $u, v \in M$.

Both RANK-AGGREGATION and MIN-FAS are computationally hard problems. The MIN-FAS problem has been studied extensively and a lot is known about its approximability. Moreover, using MIN-FAS as a starting point makes it possible to study the problem of aggregating a possibly very large number of rankings using only the weights $C(u,v)$, the number of which is only $|M|^2$, possibly a lot less than the number of rankings in our input. To this end we still need one important definition before we can move on to the next chapter.

The pair order matrix: Let D be a set of rankings. The pair order matrix C_D associated with D is an $|M| \times |M|$ matrix, where

$$C_D(u,v) = |\{\pi \in D : u \prec_{\pi} v\}|.$$

The normalized pair order matrix $\hat{C}_D$ is an $|M| \times |M|$ matrix, where

$$\hat{C}_D(u,v) = \frac{C_D(u,v)}{C_D(u,v) + C_D(v,u)}.$$

If D contains chains, it might be that some u and v never occur together in a chain $\pi \in D$. In this case we let $\hat{C}_D(u,v) = \hat{C}_D(v,u) = 0.5$.

In the regular pair order matrix C_D we only store the counts of how many times u precedes v in the input rankings for all u and v. The normalized matrix can be interpreted as a set of probabilities. Therefore we say it satisfies the *probability constraint* given by

$$\hat{C}_D(u,v) + \hat{C}_D(v,u) = 1. \tag{2.2}$$

The pair order matrix will be a key concept in Chapters 3, 4, and 6.

Chapter 3

Algorithms for Finding Bucket Orders

The main result of this chapter is a randomized algorithm with an expected approximation guarantee for the problem of computing a global ranking of a set of items given a pair order matrix on the same items. This global ranking can be a total order, but we also argue why it might be better to use a partial order as the aggregate representation. The algorithm we propose can be used to find certain types of partial orders called *Bucket orders* by tuning a single parameter. We also discuss some alternative algorithms for finding good bucket orders. These are computationally considerably more inefficient than the randomized algorithm but may lead to better solutions under some circumstances.

3.1 The bucket order model

In the previous chapter we discussed the problem of finding a total order on the items in M given a set of rankings. This is a natural approach, but has some drawbacks. First of all, our assumption that there actually exists a total order for the items may be incorrect. This is a question of selecting the correct model family, if one uses learning theoretic concepts.

The second issue concerns actual usefulness of a total order. In some cases we might argue that a total order is too complex for this task, especially if the number of items in M is large. Remember that one of our original intentions is to find a concise representation for the entire set of rankings, not necessarily rank them for the sake of finding the best or worst alternative.

A natural approach would be to use arbitrary partial orders instead of total orders. This is has been done previously [MM00, UFM05, RY06], but it is even easier to say that an arbitrary partial order does not meet the requirements of a

	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8
u_1	0.5	0.5	1	1	1	1	1	1
u_2	0.5	0.5	1	1	1	1	1	1
u_3	0	0	0.5	0.5	0.5	1	1	1
u_4	0	0	0.5	0.5	0.5	1	1	1
u_5	0	0	0.5	0.5	0.5	1	1	1
u_6	0	0	0	0	0	0.5	0.5	1
u_7	0	0	0	0	0	0.5	0.5	1
u_8	0	0	0	0	0	0	0	0.5

Figure 3.1: Matrix representation B of the bucket order $\mathcal{B} = \{\{u_1, u_2\} \prec \{u_3, u_4, u_5\} \prec \{u_6, u_7\} \prec \{u_8\}\}$.

concise representation. For instance, in practical situations with a large number of ranked items, users are likely to find that comparing two partial orders is difficult. Moreover, it is not obvious to give a precise definition for a partial order that "fits the data in the best possible way". On one hand all important aspects of the available information should be covered, on the other hand we may not let the partial order become too simple or too complex. Specifying a clear definition that does not make use of additional parameters or regularization components based on the structure of the partial order is nontrivial.

We argue that bucket orders, as defined in Section 2.1, are a better model. The idea is to combine good properties of total and partial orders by having a simply defined structure that is still flexible enough to account for cases where the input lacks information to make exact conclusions about the order. Moreover, total orders are a special case of bucket orders. Recall, that given the set D of orders, we can compute the normalized pair order matrix $\hat{C}_D$. We define an analogous structure for bucket orders.

Matrix representation of bucket orders: Let $\mathcal{B} = \{M_1 \prec \ldots \prec M_k\}$ be a bucket order on the set M. The pair order matrix B associated with $\mathcal{B}$ is defined as follows:

$$B(u,v) = \begin{cases} 1 & \text{if } (u,v) \in \mathcal{B}, \\ 0.5 & \text{if } (u,v) \notin \mathcal{B} \wedge (v,u) \notin \mathcal{B}, \\ 0 & \text{if } (v,u) \in \mathcal{B}. \end{cases}$$

Note that $B(u,v) + B(v,u) = 1$ for all u and v. An example of a bucket order in matrix representation is given in Figure 3.1.

The matrix representation of a bucket order $\mathcal{B}$ is convenient when defining the optimal bucket order given D. The bucket order $\mathcal{B}$ is optimal, when its matrix representation B is close to the normalized pair order matrix $\hat{C}_D$ in terms of the L_1 norm. We define the *cost of a bucket order* as follows:

Cost of a bucket order: The cost of a bucket order $\mathcal{B}$ given the normalized pair order matrix $\hat{C}_D$ is

$$c(\mathcal{B}, \hat{C}_D) = \sum_{u \in M} \sum_{v \in M} |\hat{C}_D(u,v) - B(u,v)|,$$

where B is the bucket matrix associated with $\mathcal{B}$.

Note that this resembles the MIN-FAS (and hence RANK-AGGREGATION) problem if we let $\mathcal{B}$ be a total order. The difference is that we do not only count the sum of the costs of backward edges, but also assign a cost to the forward edges that depends on their weight. Ideally the edge (u, v) should have zero weight if it ends up being a backward edge and unit weight if it ends up being a forward edge. In this case its cost would be zero, because for backward edges $B(u, v) = 0$ and for forward edges $B(u, v) = 1$.

Problem (OPTIMAL-BUCKET-ORDER) Let $\hat{C}_D$ be a normalized pair order matrix. The bucket order $\mathcal{B}^*$ is an optimal bucket order for D when

$$\mathcal{B}^* = \arg\min_{\mathcal{B}} c(\mathcal{B}, \hat{C}_D).$$

Note that we have defined $\mathcal{B}^*$ without using any parameters or additional regularization terms that penalize for too complex models. Simplicity of this definition is one of the reasons to use bucket orders instead of arbitrary partial orders. However, as stated in the next theorem, finding the optimal bucket order is computationally hard. Therefore we propose an approximation algorithm in Section 3.2 that finds solutions with an expected cost within a constant factor of $c(\mathcal{B}^*, \hat{C}_D)$.

Theorem 3.1.1 *[GMPU06]* OPTIMAL-BUCKET-ORDER *is NP-hard.*

Proof The proof is a reduction from MIN-FAS-TOURNAMENT. In that problem, given a complete directed graph $G = (V, E)$ (without cycles of two vertices) the task is to find a total order T^* for the vertices V on a line so that the number of edges that point to the left is minimized. This problem was recently shown to be NP-hard [Alo06]. For the adjacency matrix A of G we have $A(u, v) = 1$ if $(u, v) \in E$ and $A(u, v) = 0$ if $(u, v) \notin E$. Note that G is complete, and thus we have $(v, u) \in E$ whenever $(u, v) \notin E$ for $u \neq v$. Thus, A has the same structure as a pair order matrix.

Next we show that given G, the cost $c_G(\mathcal{B}^*)$ of the optimal bucket order $\mathcal{B}^*$ is the same as the cost $c_G(T^*)$ of the optimal total order T^*. As the class of bucket orders contain total orders as a special case, it is obvious that $c_G(\mathcal{B}^*) \leq c_G(T^*)$, because we can always set $\mathcal{B}^* = T^*$. It remains to be shown that $c_G(\mathcal{B}^*) \geq c_G(T^*)$.

Suppose that $\mathcal{B}^*$ is not a total order. We show how to convert $\mathcal{B}^*$ to a total order T^* so that the cost does not increase. First, note that the vertices belonging to a single bucket M_i can be considered independently of vertices belonging to the other buckets. That is, we can order the vertices in M_i arbitrarily without affecting the cost that is incurred by vertices in other buckets.

Let $|M_i| = s$. For all $u, v \in M_i$, the matrix B^* associated with $\mathcal{B}^*$ has $B^*(u, v) = 0.5$. As there are $s(s-1)$ edges between the s vertices, and for all of them the pair order matrix A associated with G has either $A(u, v) = 0$ or $A(u, v) = 1$, it is easy to see that the total cost of M_i given $\mathcal{B}^*$ is $0.5s(s-1)$. Now, let us assign some arbitrary total order π to the vertices in M_i. Denote by π^R the reverse order of π. If the cost of M_i given π, denoted c_π, is less than $0.5s(s-1)$ we are done. If the cost is higher, we must have $c_{\pi^R} \leq 0.5s(s-1)$. To see this, remember that all edges between vertices in M_i incur a cost of 1 given either π or π^R. As there are $s(s-1)$ edges in total, we have $c_\pi + c_{\pi^R} = s(s-1)$. If now $c_\pi \geq 0.5s(s-1)$, then it trivially follows that $c_{\pi^R} \leq 0.5s(s-1)$. Hence we can find a total order for vertices in M_i that does not increase the cost of M_i. We can repeat this for all buckets, and obtain a total order T^* with $c_G(T^*) \leq c_G(\mathcal{B}^*)$.

Thus, for matrices $\hat{C}$ that correspond to tournament graphs, the problem of finding $\mathcal{B}^*$ is equivalent to the problem of finding the optimal total order T^*. As finding T^* is NP-hard, finding $\mathcal{B}^*$ is also NP-hard. [GMPU06] □

The basic idea of the proof warrants a further comment. Even though for unweighted tournaments finding the optimal bucket order is equivalent to finding the optimal total order, and the costs of these are the same, this is not the case for arbitrary pair order matrices. In fact, for a pair order matrix with a clear *bucket structure*, such as the one shown in Figure 3.1, a bucket order will have a cost of zero while a total order has a significantly higher cost. To see this, consider the L_1 norm between the matrix in Figure 3.1 and a square matrix T of the same size with 0.5s on the diagonal, 1s in the upper triangle and 0s in the lower triangle. Such a matrix corresponds to a total order. This matrix will induce a cost of 1 for the bucket $\{u_1, u_2\}$, since

$$\begin{aligned} & |\hat{C}_D(u_1, u_2) - T(u_1, u_2)| + |\hat{C}_D(u_2, u_1) - T(u_2, u_1)| \\ = \; & |0.5 - 1| + |0.5 - 0| = 1. \end{aligned}$$

Likewise, we get a cost of 3 for bucket $\{u_3, u_4, u_5\}$ and a cost of 1 for bucket $\{u_6, u_7\}$. The total cost of T is thus 5, while the cost of a bucket order is 0 in this simple case.

Of course, typically the normalized pair order matrices corresponding to a set of rankings do not have a bucket structure as clear as the matrix of Figure 3.1.

3.2 The Bucket-Pivot algorithm

Given the result of Theorem 3.1.1, it seems very unlikely that we could find the optimal bucket order $\mathcal{B}^*$ in polynomial time given an arbitrary pair order matrix. However, turns out that the problem admits a constant factor approximation algorithm that we describe in this section. The algorithm is a modification of the method of Ailon et. al. [ACN05], who gave an approximation algorithm for the MIN-FAS problem. We show that the same approach can be used for finding bucket orders and that a similar proof technique yields similar bounds for the approximation.

We start by formally defining constant factor approximation algorithms.

Approximation algorithm: Let $\mathcal{I}$ be an *instance* of the problem $\mathcal{P}$ and $\mathcal{S}_\mathcal{I}$ a set of *feasible solutions* for $\mathcal{I}$. An algorithm $\mathcal{A}$ is a mapping from $\mathcal{I}$ to $\mathcal{S}_\mathcal{I}$. To every $s \in \mathcal{S}_\mathcal{I}$ we associate a cost with the *objective function* $f : \mathcal{S}_\mathcal{I} \to \mathbb{R}$. Let $s^* = \arg\min_{s \in \mathcal{S}_\mathcal{I}} f(s)$ be the optimal solution for instance $\mathcal{I}$ and denote by $f(s^*)$ it's cost. We say the algorithm $\mathcal{A}$ is a *constant factor approximation algorithm* for problem $\mathcal{P}$ when there is an $\alpha > 1$ such that

$$f(\mathcal{A}(\mathcal{I})) \leq \alpha f(s^*)$$

for all instances $\mathcal{I}$. In case of a randomized algorithm $\mathcal{A}$ the definition is

$$E[f(\mathcal{A}(\mathcal{I}))] \leq \alpha f(s^*),$$

where $E[X]$ denotes the expectation of the random variable X. Thus, an algorithm is a constant factor approximation algorithm if the cost of the solution found by the algorithm deviates from the optimal solution at most by a constant factor.

The Bucket Pivot -algorithm is a randomized algorithm with an expected constant factor approximation guarantee. The algorithm can be characterized as a "probabilistic quicksort". Recall, that quicksort is a (randomized) algorithm that sorts a set of items M in ascending order given an order relation on M. The idea of quicksort is to pick one of the items as a *pivot*, and place all other items either on the left or right side of the pivot based on the given order relation, and then recursively sort the left and right subsets. This will lead to a correct ordering of the items when the order relation is known.

In our case, however, the order relation is not known. What we do have, is the pair order matrix $\hat{C}_D$. As the values in $\hat{C}_D$ are normalized so that $\hat{C}_D(u, v) + \hat{C}_D(v, u) = 1$, we can interpret them as probabilities of the items to precede one another. Given two items u and v and a known and correct order relation on M, we can say with certainty which of the two items comes first in the ordering. Given $\hat{C}_D$ we have only a (possibly inaccurate) estimate of the probability for one of the items to precede the other. But if we set a threshold on this probability we can still run quicksort as usual. The intuitive choice is to place u before v if $\hat{C}_D(u, v) > 0.5$, v before u if $\hat{C}_D(v, u) > 0.5$ and choose the order arbitrarily when $\hat{C}_D(u, v) = 0.5$. This is precisely what is done in [ACN05], where it is shown that this quicksort like algorithm has an expected approximation guarantee. If the probabilities are estimated from a set of rankings, this approach can be used for solving RANK-AGGREGATION with the Kendall distance.

One of the motivations we listed for using bucket orders as a good model for summarizing a set of rankings was that a total order, as given by quicksort, can in some cases be too strict. Especially if the probabilities in $\hat{C}_D$ are only rough estimates, a total order τ constructed based on them may lead to incorrect conclusions. Recall the example where an input as shown in Figure 3.1 is approximated with a total order. Not only does the total order have a higher cost, but also the order of some of the items is completely random and not backed by the data. For instance, a total order produced quicksort might have placed item u_4 before items u_3 and u_5 and item u_3 before item u_5. However, saying that item u_4 precedes item u_5 input is not meaningful in this case. A better model should reflect the uncertainty present in the data and avoid making any arbitrary selections.

To overcome these problems we make a simple modification to quicksort, and call the resulting algorithm Bucket Pivot. Bucket Pivot works exactly like quicksort, with the exception that when the probability of an item u to precede the pivot item is close to 0.5, we do not place u to the left or right side of the pivot, but instead use a third set, denoted S, that means u is on the "same" level as the pivot. Again we recursively find a bucket order for the left and right subsets while the set S forms a single bucket. These are then combined to produce the final bucket order.

Pseudocode for Bucket Pivot is given in Algorithm 1. The parameter $\beta > 0$ is used to control the size of the buckets. Smaller values of β lead to many small buckets, whereas large values lead to a few large buckets. Choosing the optimal β depends on how much evidence we expect there to be for an item to precede another. If lots of evidence is required, then we should assign β a larger value than in the case when already a small difference in the probabilities $\hat{C}_D(u, v)$ and $\hat{C}_D(v, u)$ is enough. Note that by setting $\beta = 0$ the Bucket Pivot algorithm reduces to the algorithm of [ACN05]. The default value of β is $\frac{1}{4}$, this is also used in the later proof for showing the approximation guarantee of Bucket Pivot.

Randomized quicksort has worst-case running time $O(m^2)$ but the average case complexity is $O(m \log m)$ [CLRS01]. Bucket Pivot differs from randomized quicksort only by constructing the S in addition to the sets L and R. This can only cause Bucket Pivot to stop the recursion *earlier* than quicksort, meaning the worst case and expected running times of Bucket Pivot are the same as with randomized quicksort. Of course the running time is also affected by β. If we set

Algorithm 1: The Bucket Pivot Algorithm. By default, we use $\beta = \frac{1}{4}$.

```
1: BP($M$,$\mathcal{C}_D$,$\beta$) {Input: $M$, set of items; $\mathcal{C}_D$, pair order matrix; $\beta \geq 0$, parameter. Output: Bucket order.}
2: **if** $M = \emptyset$ **then**
3:    **return** $\emptyset$
4: **end if**
5: Pick a pivot $p \in M$ uniformly at random.
6: $L \leftarrow \emptyset$
7: $S \leftarrow \{p\}$
8: $R \leftarrow \emptyset$
9: **for all** items $u \in M \setminus \{p\}$ **do**
10:    **if** $\mathcal{C}_D(p,u) < \frac{1}{2} - \beta$ **then**
11:       Add $u$ to $L$.
12:    **else if** $\frac{1}{2} - \beta \leq \mathcal{C}_D(p,u) < \frac{1}{2} + \beta$ **then**
13:       Add $u$ to $S$.
14:    **else if** $\frac{1}{2} + \beta \leq \mathcal{C}_D(p,u)$ **then**
15:       Add $u$ to $R$.
16:    **end if**
17: **end for**
18: **return** order $\langle \mathrm{BP}(L,\mathcal{C}_D,\beta), S, \mathrm{BP}(R,\mathcal{C}_D,\beta) \rangle$
```

$\beta = \frac{1}{2}$ Bucket Pivot will have a worst case running time of $O(m)$, but the result produced is also not very useful as all items are placed in one single bucket.

An important consequence of the expected $O(m \log m)$ running time is that we only need to inspect $O(m \log m)$ elements of the matrix $\hat{C}_D$. Under some circumstances this can be used to speed up the total running time of finding the model, as we do not need to estimate all pairwise probabilities.

Approximation guarantee

Next we show that Bucket Pivot is an expected constant factor approximation algorithm. The proof follows the technique used in [ACN05] with some modifications to accommodate the case when the output is not restricted to be a total order. Note that for now we fix the parameter $\beta = \frac{1}{4}$. We also rewrite the expected cost $E[c(\mathcal{B}_{BP}, \hat{C}_D)]$ in terms of a complete, weighted directed graph $G = (M, M \times M, \hat{C}_D)$, where M is the set of vertices, and the weight of the edge $e = (u,v)$ is given by $\hat{C}_D(u,v)$. We let

$$E[c(\mathcal{B}_{BP}, \hat{C}_D)] = E[c(\mathcal{B}_{BP}, G)] = \sum_{(u,v) \in M \times M} E[|B_{BP}(u,v) - \hat{C}_D(u,v)|].$$

Note that as (u, v) and (v, u) both belong to $M \times M$, the above equation counts the cost of each pair twice; once for (u, v) and once for (v, u). Their costs are the same since $\hat{C}_D(u, v) = 1 - \hat{C}_D(v, u)$ and $B_{BP}(u, v) = 1 - B_{BP}(v, u)$ for all u and v. To simplify this, we let A denote the set of all unordered pairs of items in M and write

$$E[c(\mathcal{B}_{BP}, G)] = \sum_{\{u,v\} \in A} 2E[|B_{BP}(u, v) - \hat{C}_D(u, v)|] = \sum_{e \in A} E[c(B_{BP}, e)]. \quad (3.1)$$

The key of the approximation result is to find a proper expression for the expected cost of a pair, $E[c(B_{BP}, e)]$. To do this we consider the possible cases that can happen to e during the execution of the algorithm. Crucial here is to see how e is related to the pivot vertex p.

First of all, we claim that when p is one of the endpoints of e, the cost $c(B_{BP}, e)$ will always be optimal. Indeed, let $e = \{p, u\}$ and consider the following cases:

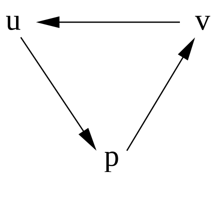
1. $\hat{C}_D(p, u) \in [0, \frac{1}{4})$: The algorithm will put u in the set L, and hence u will appear before p in the final output. Thus, we will have $B_{BP}(p, u) = 0$ and $B_{BP}(u, p) = 1$.

2. $\hat{C}_D(p, u) \in [\frac{1}{4}, \frac{3}{4}]$: The algorithm will put u in the set S, and u will appear in the same bucket as p in the final output. Thus, we will have $B_{BP}(p, u) = 0.5$ and $B_{BP}(u, p) = 0.5$ as well.

3. $\hat{C}_D(p, u) \in (\frac{3}{4}, 1]$: The algorithm will put u in the set R, and u will appear after p in the final output. Thus, we will have $B_{BP}(p, u) = 1$ and $B_{BP}(u, p) = 0$.

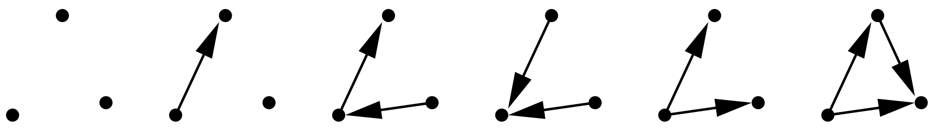
In every case the absolute value $|B_{BP}(u, v) - \hat{C}_D(u, v)|$ will be minimized and the cost $c(B_{BP}, e)$ is optimal.

Thus, the only possibility of $c(B_{BP}, e)$ to be nonoptimal is when the pair e is assigned a cost without the pivot p being a part of e. This can only happen when e is opposite to the pivot in a triangle of three vertices. See the figure below for an example:

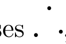


The arrows indicate which direction has a large weight in $\hat{C}_D$, i.e., when the arrow points from u to p, we have $\hat{C}_D(u, p) > \frac{3}{4}$. In this case if p is chosen as the pivot and $e = \{u, v\}$, the cost $c(B_{BP}, e)$ will be nonoptimal. The vertex u will be put to the set L and v will be put to the set R. As a result u will precede v in the final output, and we will have $B_{BP}(u, v) = 1$, although it actually is the case that $\hat{C}_D(u, v) < \frac{1}{4}$, and the optimal placement of u and v would place v before u.

The weights in $\hat{C}_D$ can be used to define other kinds of triangles as well. In addition to the directed cycle shown above, we have also six other classes of triangles. These are:



We define them as follows: two vertices u and v are connected by a line whenever $\hat{C}_D(u, v)$ is either less than $\frac{1}{4}$ or larger than $\frac{3}{4}$. The direction of the arrow indicates what vertex is likelier to precede the other if the values of $\hat{C}_D$ are interpreted as probabilities. If $\hat{C}_D(u, v)$ is in the range $[\frac{1}{4}, \frac{3}{4}]$ we do not draw a line between the points.

We already argued that the cost $c(B_{BP}, e)$ will be smallest when either endpoint of e is chosen as the pivot, and claimed that the cost of an edge e can only be nonoptimal when it is opposite to the pivot vertex. However, this is only a necessary condition, it is not sufficient. Consider the triangle classes , , and . It is easy to see that in every case, no matter what vertex is chosen as the pivot, the edge opposite to it will always either

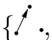
a) have the smallest possible cost, i.e., it's orientation in the final output will be concordant with $\hat{C}_D$, or

b) both of its endpoints will be put to the set L (or R), and hence it's cost will be defined in a following recursive call of the algorithm.

For example, in case of , if we choose the vertex with two outgoing (or incoming) edges as the pivot, the opposite edge will be put to the set R (or L) and it's cost is not yet defined. If we choose the vertex with one incoming and one outgoing edge as the pivot, the edge opposite to it will be oriented in the correct way according to $\hat{C}_D$.

This leaves us with only the classes , and . In the example above we considered . Since it is completely symmetric, it is obvious that no matter what vertex is chosen as the pivot, the edge opposite to it will always incur a nonoptimal cost. The remaining cases are and . In case of , if we choose the vertex that is adjacent to the undirected edges, both endpoints of the directed edge (u, v) will be put to the set S and hence we will have $B_{BP}(u, v) = 0.5$ even though $\hat{C}_D(u, v) \in (\frac{3}{4}, 1]$. If either of the vertices adjacent to the directed edge is chosen as the pivot, one of the vertices of the undirected edge (u, v) that is opposite to it will be put to the set S, while the other one is put to the set R (or L). This means $B_{BP}(u, v) = 1$ or $B_{BP}(u, v) = 0$, even though $\hat{C}_D(u, v) \in [\frac{1}{4}, \frac{3}{4}]$.

In case of the following occurs: If the vertex with one outgoing (or incoming) edge is chosen as the pivot, the other vertex of the opposite edge will be put to the set R (or L) while the other one is put to the set S. In both cases the arrow will end up pointing the "wrong" way. If the vertex with one incoming and one outgoing edge is chosen as the pivot, the edge on the opposite side will be assigned a direction even though both of its endpoints should belong to the same bucket.

Define the set $\mathcal{T} = \{$, , $\}$; these are the triangle classes that will always cause a nonoptimal cost $c(B_{BP}, e)$ when the vertex opposite to e is chosen as the pivot. Let $c_T(e)$ denote the cost that e incurs given that it appears in a triangle of class T opposite to the pivot in a recursive call of the algorithm. Denote the probability of this with $p_T(e)$. Furthermore, let $c_{opt}(e)$ denote the cost incurred to e when it either is adjacent to the pivot or appears opposite to the pivot in one of the triangle classes not belonging to $\mathcal{T}$. Note that we have

$$c_{opt}(e) = \min_{x \in \{0, 0.5, 1\}} 2|x - \hat{C}_D(u, v)|,$$

where $e = \{u, v\}$. Finally, let $c^*(e)$ denote the cost of the edge e in the *globally optimal* solution $\mathcal{B}^*$. Obviously we have $c_{opt}(e) \leq e^*(e)$, since in the optimal solution the cost of an edge might sometimes be higher than the locally optimal cost $c_{opt}(e)$. For instance in case of the directed cycle ( ) one of the edges *always* has to pay a nonoptimal cost.

Using the above, we can write the expected cost of e as follows:

$$E[c(B_{BP}, e)] = \sum_{T \in \mathcal{T}} p_T(e) c_T(e) + \big(1 - \sum_{T \in \mathcal{T}} p_T(e)\big) c_{opt}(e). \tag{3.2}$$

Either e pays the nonoptimal cost as it ends up opposite to the pivot in some of the triangles belonging to $\mathcal{T}$, or it pays $c_{opt}(e)$ in the remaining cases. When Equation 3.2 is substituted into 3.1, we get

$$E[c(\mathcal{B}_{BP}, G)] = \underbrace{\sum_{e \in A} \sum_{T \in \mathcal{T}} p_T(e) c_T(e)}_{H_{BP}} + \underbrace{\sum_{e \in A} \big(1 - \sum_{T \in \mathcal{T}} p_T(e)\big) c_{opt}(e)}_{L_{BP}}, \tag{3.3}$$

where H_{BP} is the part with the nonoptimal ("high") costs, and L_{BP} the part with the locally optimal ("low") costs.

The next task is to derive $p_T(e)$. Define the events $X_T(t,e)$ and $B(e)$. The event $X_T(t,e)$ means that all vertices of the triangle $t \in T$, one side of which is e, appear in a recursive call of the algorithm, and one of t's vertices is chosen as the pivot. The event $B(e)$ happens when the pivot chosen is the vertex opposite to edge e. Given that $X_T(t,e)$ happens, the probability of $B(e)$ is just $\frac{1}{3}$ as each of the three vertices has equal probability of becoming the pivot. If p_t is the probability of $X_T(t,e)$, we can write

$$Pr(X_T(t,e) \wedge B(e)) = Pr(B(e)|X_T(t,e))Pr(X_T(t,e)) = \frac{1}{3}p_t.$$

This is the probability of one $t \in T$ causing e to pay a nonoptimal cost. For the entire class T we have

$$p_T(e) = \sum_{t:e \in t \in T} \frac{1}{3}p_t. \tag{3.4}$$

This must be a probability (≤ 1), since the events $X_T(t,e) \wedge B(e)$ and $X_T(t',e) \wedge B(e)$ are disjoint for all $t, t' \in T$. If e is charged to triangle t, it can not be charged to triangle t'. In fact, the same goes for all triangles in the input graph. Any edge e can only cause a nonoptimal cost with *one* triangle, no matter what class this triangle belongs to. This means that for all e

$$\sum_{T \in \mathcal{T}} \sum_{t:e \in t \in T} \frac{1}{3}p_t \leq 1.$$

When Equation 3.4 is substituted into H_{BP}, we obtain

$$\begin{aligned}
H_{BP} &= \sum_{e \in A} \sum_{T \in \mathcal{T}} \sum_{t:e \in t \in T} \frac{1}{3}p_t c_T(e) \\
&= \sum_{T \in \mathcal{T}} \sum_{e \in A} \sum_{t:e \in t \in T} \frac{1}{3}p_t c_T(e) \\
&= \sum_{T \in \mathcal{T}} \sum_{t \in T} \frac{1}{3}p_t \sum_{e \in t} c_T(e) \\
H_{BP} &= \sum_{T \in \mathcal{T}} \sum_{t \in T} \frac{1}{3}p_t c_T(t),
\end{aligned}$$

where $c_T(t) = \sum_{e \in t} c_T(e)$. Based on the discussion above we are now able to state the following result:

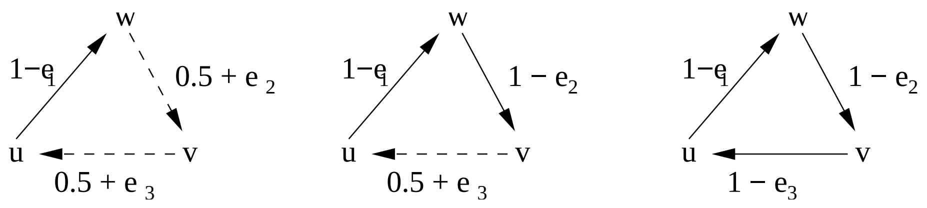


Figure 3.2: Triangle classes leading to suboptimal alignment of the edge opposite to the pivot. To each edge we have associated its possible value of $\hat{C}_D$. For the cases where the weight is $1 - e_i$, we have $e_i \in [0, \frac{1}{4}]$. If the weight is $0.5 + e_i$, we have $e_i \in [-\frac{1}{4}, \frac{1}{4}]$.

Theorem 3.2.1 *The Bucket-Pivot algorithm is randomized α-approximation algorithm, meaning*

$$E[c(\mathcal{B}_{BP}, \hat{C}_D)] \le \alpha c(\mathcal{B}^*, \hat{C}_D),$$

if for all triangle classes $T \in \mathcal{T}$ we have

$$c_T(t) \le \alpha c^*(t),$$

where $c^*(t) = \sum_{e \in t} c^*(e)$ is the cost of the edges belonging to t in the globally optimal solution $\mathcal{B}^*$.

Proof We decompose the optimal cost as follows: Let $c(\mathcal{B}^*, \hat{C}_D) = H^* + L^*$, where

$$H^* = \sum_{T \in \mathcal{T}} \sum_{t \in T} \frac{1}{3} p_t c^*(t) \text{ and} \tag{3.5}$$

$$L^* = \sum_{e \in A} \big(1 - \sum_{T \in \mathcal{T}} p_T(e)\big) c^*(e). \tag{3.6}$$

If we assume there exists an α so that $c_T(t) \le \alpha c^*(t)$ for all $T \in \mathcal{T}$, then we obtain that $H_{BP} \le \alpha H^*$, which implies the claim of the theorem, as it must be the case that $L_{BP} \le L^*$, since $c_{opt}(e) \le c^*(e)$ for all e. $\square$

The exact value of α is determined next. To do this we will consider all triangle classes in $\mathcal{T}$ separately. See Figure 3.2 for a graphical representation of all three classes with the possible weights (values of $\hat{C}_D$) for each edge. The weights are given as deviations from $\frac{1}{2}$ and 1. The weight of a directed edge deviates from 1 by the amount e, where $e \in [0, \frac{1}{4}]$, and the weight of an undirected edge deviates from $\frac{1}{2}$ by an amount in the range $[-\frac{1}{4}, \frac{1}{4}]$.

We will start with $\nearrow\cdot$, which is the leftmost triangle in Figure 3.2. In the table below we have collected the cost of every edge for every possible choice of the pivot.

pivot	{v,w}	{u,w}	{u,v}
u	$\frac{1}{2}+e_2$	e_1	$\lvert e_3\rvert$
v	$\lvert e_2\rvert$	$\frac{1}{2}-e_1$	$\lvert e_3\rvert$
w	$\lvert e_2\rvert$	e_1	$\frac{1}{2}+e_3$

That is, if u is chosen as the pivot, both $\{u,v\}$ and $\{u,w\}$ are placed optimally and only pay the costs e_1 and $|e_3|$, respectively. The undirected edge $\{v,w\}$ is opposite to u and will become a directed edge in the solution, and is hence charged the locally nonoptimal cost of $|0-(\frac{1}{2}+e_2)| = \frac{1}{2}+e_2$. Now we must find an α so that $c_T(t) \leq \alpha c^*(t)$, where $T =$. By definition we have that $c_T(t)$ equals the sum of the nonoptimal costs of each edge, hence

$$c_T(t) = \frac{3}{2} - e_1 + e_2 + e_3.$$

In this case the best possible configuration is obtained when v is chosen as the pivot. We have

$$c^*(t) = \frac{1}{2} - e_1 + |e_2| + |e_3|,$$

where $e_1 \in [0, \frac{1}{4}]$ and $e_2, e_3 \in [-\frac{1}{4}, \frac{1}{4}]$. If we let $e_1 = \frac{1}{4}$ and $e_2 = e_3 = 0$, we get

$$\alpha = \frac{c_T(t)}{c^*(t)} = \frac{\frac{5}{4}}{\frac{1}{4}} = 5.$$

Note that having e_2 and e_3 less than $\frac{1}{4}$ would only make α smaller, as $c_T(t)$ would decrease but $c^*(t)$ increase.

We continue with $T =$. The costs for different choices of the pivot have been again tabulated below.

pivot	{v,w}	{u,w}	{u,v}
u	$1-e_2$	e_1	e_3
v	e_2	$1-e_1$	e_3
w	e_2	e_1	$\frac{1}{2}+e_3$

Now we have

$$c_T(t) = \frac{5}{2} - e_1 - e_2 + e_3,$$

and

$$c^*(t) = \frac{1}{2} + e_1 + e_2 + e_3,$$

where $e_1, e_2 \in [0, \frac{1}{4}]$ and $e_3 \in [-\frac{1}{4}, \frac{1}{4}]$. Clearly it makes sense to set $e_1 = e_2 = 0$, as any other values would decrease $c_T(t)$ and increase $c^*(t)$, which we do not want. If $e_3 \geq 0$, we have $c_T(t)c^*(t)^{-1} \leq 5$, and when $e_3 = -\frac{1}{4}$, we have $c_T(t) = \frac{9}{4}$ and $c^*(t) = \frac{1}{4}$ which gives $\alpha = 9$.

Finally, we still need to cover the class . Again, the edge costs are tabulated below.

pivot	{v,w}	{u,w}	{u,v}
u	$1 - e_2$	e_1	e_3
v	e_2	$1 - e_1$	e_3
w	e_2	e_1	$1 - e_3$

Now $e_1, e_2, e_3 \in [0, \frac{1}{4}]$. We have

$$c_T(t) = 3 - e_1 - e_2 - e_3,$$

and the optimal pivot depends on the precise values of the e_i. Without loss of generality we let

$$c^*(t) = 1 + e_1 + e_2 - e_3.$$

Again we let $e_1 = e_2 = 0$ to maximize the ratio. This gives

$$\alpha = \frac{c_T(t)}{c^*(t)} = \frac{\frac{11}{4}}{\frac{3}{4}} = \frac{11}{3}.$$

We conclude the discussion of the approximation ratio by summarizing the above results. For we have $\alpha \leq 5$, for $\alpha \leq 9$ and for $\alpha \leq \frac{11}{3}$. Thus, Theorem 3.2.1 holds when $\alpha = 9$ and we obtain the following corollary.

Corollary 3.2.2 *Bucket-Pivot is a randomized* 9*-approximation algorithm.*

Bucket Pivot with Pruning

Note that in general the Bucket Pivot algorithm does *not* compute the cost $c(\mathcal{B}, \hat{C}_D)$, otherwise it's running time would be quadratic in the number of items. However, with small data sets we can afford to compute the entire cost and use this as a heuristic to improve the solution. We now discuss a situation where Bucket Pivot, as described in Algorithm 1, does not always compute the intuitive solution.

Consider the case where the normalized pair order matrix satisfies the probability constraint of Equation 2.2 but does not contain any other structure. We can generate such a matrix for instance by assigning uniformly distributed random values from the range $[0, 1]$ to the upper triangle and set the lower triangle so that the probability constraint is satisfied. Intuitively the correct solution for this kind of input would be to put everything into the same bucket. But when

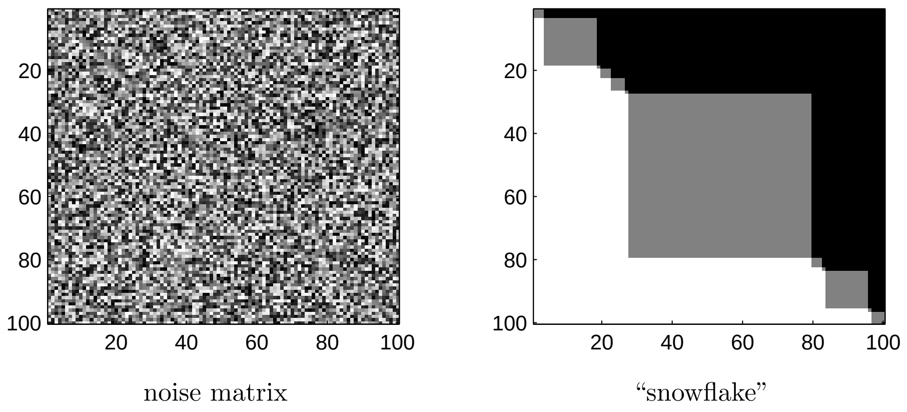


Figure 3.3: When Bucket Pivot is run with the noise matrix on the left as the input and $\beta = \frac{1}{4}$ we obtain the “snowflake” structured bucket order shown on the right.

Bucket Pivot is run with this matrix as the input and $\beta = \frac{1}{4}$, we obtain the “snowflake” shaped bucket order shown in Figure 3.3.

To see why this occurs, consider what happens at each level of the recursion. No matter which row we pick as the pivot, roughly half of the elements on the row lie between $\frac{1}{4}$ and $\frac{3}{4}$. These are put into a single big bucket. The remaining elements for which the value is either less than $\frac{1}{4}$ or larger than $\frac{3}{4}$ are put into the L and R sets, respectively. In these sets we pick new pivots, but the fraction of elements in the ranges $[0, \frac{1}{4}]$, $[\frac{1}{4}, \frac{3}{4}]$ and $[\frac{3}{4}, 1]$ remains the same. Hence, the resulting bucket order has a fractal-like structure.

Clearly this is not a desirable property. If there is no structure in the input, the algorithm should not artificially introduce it either. To overcome the problem we propose the following simple modification to Bucket Pivot. Let $\text{BP}_L = \text{BP}(L, \hat{C}_D, \beta)$ and $\text{BP}_R = \text{BP}(R, \hat{C}_D, \beta)$. On line 18 in Algorithm 1, instead of simply returning the bucket order $\langle \text{BP}_L, S, \text{BP}_R \rangle$, we first compute the cost of both BP_L and BP_R. These are simply the L_1 norms between the matrices corresponding to BP_L (and BP_R) and $\hat{C}_D$ projected to L (and R), respectively. We also compute the cost of using only one bucket for representing the L and R sets. The alternative with the smaller cost is used in the final solution returned by the algorithm.

With this modification Bucket Pivot no longer outputs the “snowflake” orders with inputs as shown on the left in Figure 3.3. Instead, the algorithm returns a solution with only one bucket that contains all of the items as expected. Also note that this modification can only make the solutions returned by the algorithm better; the approximation bounds obtained for the original algorithm still hold.

3.3 Alternative algorithms

The Bucket Pivot algorithm described in the previous section has several nice properties. The approximation guarantee is foremost of theoretical interest due to the fairly large constant factors. Even though the algorithm does use a parameter, one does not need to specify the exact number of buckets in advance, which makes model selection easier. And finally, the algorithm is very fast in practice.

One problem of Bucket Pivot is that its output may vary considerably between individual runs of the algorithm due to the randomization. One approach would be to run the algorithm a number of times and select the solution with the smallest cost. This strategy, however, leads to a quadratic time complexity, as we must access each element of an $m \times m$ matrix when computing the total cost of a the bucket order $\mathcal{B}_{BP}$. In this section we discuss an alternative approaches that are not as computationally efficient, but can lead to more robust results.

Instead of finding the bucket order directly, we can use a two-step method. First we compute a total order on the items M, and then find the bucket boundaries in a second step. This way we can obtain a fully deterministic algorithm for finding bucket orders. Alternatively, we can first try to group the items to form the buckets, and then find an order for the buckets. We describe these two approaches that take the pair order matrix $\hat{C}_D$ and a number k as input. The algorithms will output a bucket order with k buckets.

An approach based on segmentation

Next we will consider the method where one first orders the items and then finds k buckets by placing bucket boundaries at $k-1$ positions in the order. As discussed in Section 2.2 in the context of vertex ordering problems and rank aggregation, finding the optimal total order for M given $\hat{C}_D$ is hard. By optimal we mean a total order that minimizes the sum of the edge weights pointing to the "wrong" direction. An order that approximates this can always be found, however. In fact, we can use Bucket Pivot to do this simply by setting $\beta = 0$. This will yield a constant-factor approximation. Alternatively, we can rank the items in decreasing order of the sums of each row of $\hat{C}_D$. This was shown in [CFR06] to be a 5-approximation for the same problem.

Once we have obtained the total order τ, we must find $k-1$ locations for the bucket boundaries. If τ is a linear extension of the true bucket order $\mathcal{B}^*$, it is possible to find $\mathcal{B}^*$ in polynomial time by using dynamic programming. We will discuss this shortly. However, there is no guarantee that any polynomial time algorithm will in fact find such a τ, so the problem of finding $\mathcal{B}^*$ remains hard. Finding the bucket boundaries can be seen as a *segmentation problem.*

Problem (TIME-SERIES-SEGMENTATION) Let $X(1,n) = (x_1, x_2, x_n)$ be a time-series of length n, where each x_i is a h dimensional vector. A *segment* of X is defined by its begin and end points i_b and i_e, respectively. The segment $X(i_b, i_e)$ contains all such x_i where $i \in [i_b, i_e]$. A *segmentation* $\mathcal{S}_k$ of X is a set disjoint set of k segments so that every x_i belongs to one and only one segment. The optimal segmentation $\mathcal{S}_k^*$ is defined as

$$\mathcal{S}_k^* = \arg\min_{\mathcal{S}_k} \sum_{X(i_b,i_e)\in\mathcal{S}_k} c(X(i_b, i_e)),$$

where $c(X(i_b, i_e))$ is a cost associated with the segment starting from i_b and ending at i_e.

The precise definition of the cost $c(X(i_b, i_e))$ depends on the model we use to represent a segment. The optimal segmentation $\mathcal{S}_k^*$ can be found in polynomial time by using dynamic programming. Essentially we must solve the following equation, which states that the cost of the optimal segmentation for $X(1,n)$ using k segments, denoted $c(\mathcal{S}_k^*, X(1,n))$, satisfies

$$c(\mathcal{S}_k^*, X(1,n)) = \min_{i<n}\{c(\mathcal{S}_{k-1}^*, X(1,i)) + c(X(i+1,n))\}. \tag{3.7}$$

That is, the optimal k-segmentation of $X(1,n)$ is given by the optimal $k-1$-segmentation of $X(1,i)$ plus the cost of having $X(i+1,n)$ as a single segment, minimized over all $i < n$. We first compute the cost $c(X(i_b, i_e))$ for all i_b, i_e pairs where $1 \le i_b < i_e \le m$. Then the optimal segmentation is found levelwise for each k starting from $k = 2$ up to $k = K$. The running time of this is $O(n^2K)$. To run this in practice we must define the cost $c(X(i_b, i_e))$ of a single segment.

Let $\hat{C}_D(\tau)$ denote the pair order matrix $\hat{C}_D$ after rearranging both its rows and columns according to the total order τ. We can find the bucket boundaries by solving TIME-SERIES-SEGMENTATION with $\hat{C}_D(\tau)$ as the input. To do this we simply view $\hat{C}_D(\tau)$ as an m-dimensional time series of length m. That is, we let $X = (x_1, \ldots, x_m)$, where x_i is the ith column of $\hat{C}_D(\tau)$.

The cost $c(X(i_b, i_e))$ of a segment is defined as follows. Consider a submatrix of $\hat{C}_D(\tau)$ that only contains the columns from i_b to i_e. Denote this matrix by $\hat{C}_D(X(i_b, i_e))$. The cost $c(X(i_b, i_e))$ is simply the L_1 norm between $\hat{C}_D(X(i_b, i_e))$ and the matrix B of the same size, where for all v

$$B(u,v) = \begin{cases} 1 & \text{iff } u < i_b, \\ 0.5 & \text{iff } i_b \le u \le i_e, \\ 0 & \text{iff } u > i_e. \end{cases}$$

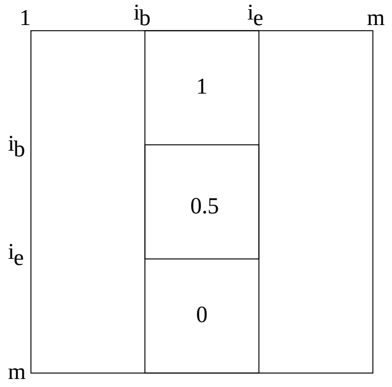


Figure 3.4: When segmenting $\hat{C}_D(\tau)$, the cost of a segment starting at i_b and ending at i_e is defined as the L_1 norm between $\hat{C}_D(X(i_b, i_e))$ and the matrix shown here between i_b and i_e.

This is because items $u < i_b$ appear *before* every item in the segment $X(i_b, i_e)$ and items $u > i_e$ appear *after* every item in $X(i_b, i_e)$. See Figure 3.4 for an illustration of this.

The drawback of dynamic programming is that it is very slow. We must compute the cost for every possible segment, which is inefficient, considering how the cost of a single segment is defined in this case. More efficient techniques have been proposed, such as Global Iterative Replacement [HKM$^+$01] and Divide & Segment [TT06]. In the experiments we use Global Iterative Replacement (GIR) instead of dynamic programming when computing the bucket boundaries.

An approach based on clustering

Instead of first ordering all m items and then finding the bucket boundaries, we can take the opposite approach and first find items that should be put into the same bucket and subsequently order the buckets obtained this way.

We use the pair order matrix $\hat{C}_D$ to find the buckets simply by clustering the columns of $\hat{C}_D$ with some known clustering algorithm, such as k-means or a hierarchical clustering algorithm. To see how this works, remember that if two items u and v in fact belong to the same bucket M_i, their columns in $\hat{C}_D$ will be similar. For all items w that belong to buckets M_j such that $j < i$ we should have $\hat{C}_D(w, u) \approx \hat{C}_D(w, v) \approx 1$. And likewise for items w that belong to buckets M_j such that $j > i$ we expect to have $\hat{C}_D(w, u) \approx \hat{C}_D(w, v) \approx 0$. See Figure 3.1 on page 3.1 for an example. This is the pair order matrix of a bucket order, but $\hat{C}_D$ should have a similar structure if it can be approximated well by a bucket order. Clearly columns that belong to the same bucket are similar to each other but different from columns belonging to other buckets.

Once we have obtained the clusters $c_1, \ldots, c_k$ we have to order them. An easy way of doing this is based on the mean vectors of each cluster. Recall, that by sorting the rows in increasing order of the column sums of $\hat{C}_D$, we obtain a 5-approximation for the MIN-FAS problem in weighted tournaments [CFR06]. We use the same approach and sort the clusters in increasing order of the sums of their centroid vectors. If $\hat{C}_D$ has a bucket structure as the matrix in Figure 3.1, then this will result in the correct order for the buckets.

3.4 Experiments

In this section we compare the methods discussed above with real data sets from a paleontological application. We will use the following notation to refer to the individual algorithms:

BP	This is the regular Bucket Pivot algorithm as described in Algorithm 1.
BP-PR	This is the Bucket Pivot algorithm extended with pruning of unnecessary buckets as described in Section 3.2.
BP-GIR	This is an algorithm based on segmentation, see Section 3.3. We first compute a total order using the Bucket Pivot algorithm (by setting $\beta = 0$) and subsequently place the bucket boundaries using Global Iterative Replacement [HKM+01].
CS-GIR	This is another algorithm based on segmentation. We first compute a total order on the items with the column-sum method, and find the bucket boundaries again using Global Iterative Replacement.
KM-CS	This is an algorithm based on clustering, see Section 3.3. We first cluster the columns of the normalized pair order matrix with k-means, and then order the clusters based on the sums of the centroid vectors.

The application that first motivated bucket orders as potentially interesting models was that of biostratigraphy. This is the problem of determining the age of a sediment based on the fossils it contains. In our case we want to find the temporal order of a number of *sites* where fossils have been found.

We use two data sets called G10S10 and G5S5. They are both based on a public database[1], and are freely available as supplementary material to [PFM06]. The data contains a 0–1 matrix with sites on the rows and genera on the columns.

[1]`http://www.helsinki.fi/science/now/index.html` (25.1.2008)

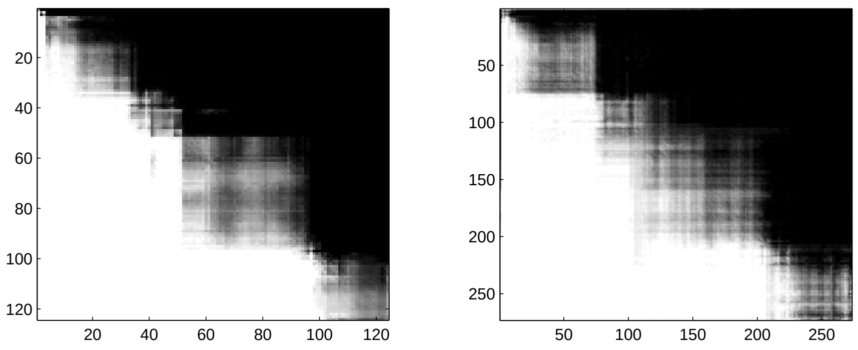


Figure 3.5: The normalized pair order matrices of the paleontological data sets. Left: G10S10, right: G5S5. Black corresponds to the value 1, white to 0, intermediate values are shown in shades of gray.

A row has ones on columns that correspond to genera whose fossils have been found at the site.

In G10S10 there are 124 distinct sites, while G5S5 has 273 sites. Each of the sites is either simply a geographical location, or a certain sediment at a certain location. The difference in the age of the youngest and oldest site is in the order of millions of years. Some of the sites are known to belong to the same era, i.e., in a bucket order model they should be placed in the same bucket. Currently the domain experts use a so called *MN classification* to evaluate the age of a given site. When two sites belong to the same MN class they are approximately of the same age.

To use the algorithms discussed in this chapter we need the normalized pair order matrix of the sites. These are shown in Figure 3.5 for both G10S10 and G5S5. The matrices are obtained by using the method of [PFM06]. This is a MCMC algorithm that computes a set D of permutations on the sites, we compute the pair order matrix based on these as explained earlier. (Later, in Chapter 7, we discuss an alternative method for computing pair order matrices from data sets such as G10S10 and G5S5.)

In Figure 3.5 the sites are ordered according to their age estimated by domain experts. There is a surprisingly clear bucket order structure to be seen in both matrices, indicating that bucket orders indeed are a suitable model class for this data set. However, we acknowledge that this structure might also have been introduced as a side effect by the MCMC algorithm of [PFM06].

In our experiment we ran each of the aforementioned algorithms 100 times

	Bp	Bp-pr	Bp-gir	Km-cs	Cs-gir
avg. c_C	315.20	314.67	278.16	369.51	277.08
min. c_C	287.59	283.86	277.05	304.52	277.08
max. c_C	368.00	355.17	280.49	516.63	277.08
std. c_C	18.41	17.64	0.75	27.50	0.00
avg. c_{MN}	356.03	358.85	232.64	337.52	228.00
min. c_{MN}	229.50	240.50	220.50	244.50	228.00
max. c_{MN}	490.00	485.00	259.00	481.50	228.00
std. c_{MN}	65.86	59.01	7.56	34.68	0.00
avg. N_b	17.54	16.86	17.16	17.00	17.00
min. N_b	14.00	14.00	14.00	17.00	17.00
max. N_b	20.00	19.00	20.00	17.00	17.00
std. N_b	1.43	1.22	1.42	0.00	0.00

Table 3.1: Results on the G10S10 data set using different algorithms. The results are based on 100 independent trials. Here c_C denotes the cost of the solution with respect to the pair order matrix, c_{MN} is the cost with respect to the bucket order given by the MN-classes, and N_b is the number of buckets in the found solution.

with the matrices of Figure 3.5 as the input. For the Bucket Pivot based algorithms we used $\beta = \frac{1}{4}$, with the exception of BP-GIR, where we let $\beta = 0$ to obtain a total order. We evaluated the results using the cost function $c(\mathcal{B}, C)$, where C is either the pair order matrix given as the input (as usual) or a bucket order matrix that equals the MN-classification[2] of the sites. We denote the first cost with c_C and the second one with c_{MN}. We also show the number of buckets found (N_b) for those algorithms that do not have this as a parameter. For KM-CS and CS-GIR we set the number of buckets to 17 to get comparable numbers with the Bucket Pivot based algorithms.

Results for G10S10 are shown in Table 3.1. The segmentation based algorithms (BP-GIR and RS-GIR) are very stable, that is, they tend to return the same bucket order on every trial. The clustering based approach clearly gives the worst results, both in terms of c_C and c_{MN}. And as expected, there is also considerable variance in the results produced by BP and BP-PR. However, the best solutions (those with the smallest cost) obtained with the Bucket Pivot based approaches are almost as good as the best ones produced by the segmentation algorithms BP-GIR and CS-GIR. Running BP or BP-PR is several orders of magnitude faster than running e.g. CS-GIR, meaning we can run Bucket Pivot a number of times and pick the best solution. Of course this means one has to evaluate the cost

[2]A classification system used by domain experts for comparing the ages of rock sediments.

	Bp	Bp-pr	Bp-gir	Rs-km	Rs-gir
avg. c_C	1787.97	1764.40	1557.32	2035.22	1545.77
min. c_C	1590.97	1607.47	1549.45	1874.18	1545.41
max. c_C	2191.56	2069.72	1576.73	2391.84	1546.25
std. c_C	97.11	97.21	5.71	78.85	0.42
avg. c_{MN}	1929.17	1895.63	1592.67	1715.98	1581.60
min. c_{MN}	1419.50	1562.50	1523.00	1531.50	1573.00
max. c_{MN}	2501.00	2520.50	1700.50	2170.50	1593.00
std. c_{MN}	222.48	197.09	36.01	90.65	9.95
avg. N_b	17.98	16.40	16.17	17.00	17.00
min. N_b	15.00	13.00	13.00	17.00	17.00
max. N_b	22.00	21.00	20.00	17.00	17.00
std. N_b	1.50	1.64	1.53	0.00	0.00

Table 3.2: Results on the g5s5 data set using different algorithms. The results are based on 100 independent trials. Here c_C denotes the cost of the solution with respect to the pair order matrix, c_{MN} is the cost with respect to the bucket order given by the MN-classes, and N_b is the number of buckets in the found solution.

function which is not of order $O(m \log m)$, but with relatively small data sets such as the ones used here it is a feasible approach. The results for g5s5 are shown in Table 3.2. There are no obvious qualitative differences to Table 3.1.

3.5 Conclusion

We have discussed the problem of computing a bucket order on a set of items given the pair order matrix $\hat{C}_D$. This problem can be seen as a variant of the rank aggregation problem with a different model class; we compute a bucket order instead of a total order. We proposed a number of algorithms for the problem. The first one is a randomized quicksort-like algorithm, for which we give a constant factor approximation bound. This algorithm is conceptually easy and runs fast. The other algorithms are based on time-series segmentation and clustering.

Chapter 4

Clustering chains

4.1 Introduction and background

Clustering is a traditional method for data exploration and analysis. The task is to partition a given set of points to homogenous groups, so that points belonging to different groups are less similar than points belonging to the same group. Scientific literature about the topic is abundant and there exist a myriad of different clustering algorithms for different purposes.

In this chapter we consider the problem of clustering chains. We describe a technique similar to Lloyd's algorithm [DH73, Llo82, BH67], also known as k-means, and conduct a number of experiments to study the algorithm's performance with real and artificial data sets. We start the discussion by characterizing typical approaches to clustering, and give arguments whether or not they are suitable for use with chains. In Section 4.2 we describe our algorithm in more detail.

A brief overview of clustering algorithms

A *clustering* of a set D of points is a disjoint partition of D, denoted $\mathcal{C} = \{D_1, \ldots, D_k\}$, where every $x \in D$ belongs to one and only one D_i, where $i \in [1, k]$ is the cluster number of x, and $D_i \cap D_j = \emptyset$ for all $i \neq j$. We abuse notation slightly and denote by $\mathcal{C}(x)$ the cluster number of point $x \in D$ in clustering $\mathcal{C}$. The aim is to have similar points belonging to the same cluster D_i, while putting dissimilar points to the other clusters D_j with $j \neq i$.

Clustering algorithms can be organized to groups based on their operation. *Hierarchical* clustering algorithms construct a tree, also called a *dendrogram*, with the points as leaves. All leaves belonging to the same subtree are considered to belong to the same cluster. This tree can be constructed either in a *top-down* or *bottom-up* fashion. To obtain the clustering $\mathcal{C}$ from the dendrogram, one has to

Algorithm 2: Lloyd's algorithm

```
$k$-means($D$, $k$) {Input: $D$, set of points; $k$, number of clusters. Output: The
clustering $\mathcal{C} = \{D_1, \ldots, D_k\}$.}
$\{D_1, \ldots, D_k\} \leftarrow$ PickInitialClusters( $D$, $k$ );
$e \leftarrow \sum_{i=1}^{k} \sum_{x \in D_i} d(\pi, \text{Centroid}(D_i))$;
**repeat**
  $e_0 \leftarrow e$;
  $\mathcal{C}_0 \leftarrow \{D_1, \ldots, D_k\}$;
  **for** $i \leftarrow 1, \ldots, k$ **do**
    $D_i \leftarrow \{x \in D | D_i = \arg\min_j d(x, \text{Centroid}(D_j)\}$;
  **end for**
  $e \leftarrow \sum_{i=1}^{k} \sum_{x \in D_i} d(x, \text{Centroid}(D_i))$;
**until** $e \geq e_0$ ;
**return** $\mathcal{C}_0$;
```

define on what level to cut the subtrees. A higher level results in a clustering with fewer clusters, whereas a low level can result in a clustering with very many clusters.

In a top-down approach all points first belong to the same cluster, which is then split to two smaller clusters according to some criteria. These clusters will from two subtrees of the final tree. The algorithm proceeds by recursively clustering both of the smaller clusters.

The bottom-up approach operates in an opposite fashion. Given a clustering, a bottom-up algorithm finds two clusters that are similar enough to be merged into one cluster. In the beginning all n points form a clustering of n clusters, and in the first step the algorithm finds two points that are closest to each other and combines them to form a cluster of two points.

In the top-down approach the hard part is to find a good split of a cluster, whereas in the bottom-up strategy the final result is considerably affected by the choice of criteria for combining two clusters. In general the top-down approach is considered more difficult, as finding the best split is harder than finding the best two clusters to merge. To find the two clusters to merge one only needs to keep a structure with the distances between every pair of clusters and update it when a merge occurs. Turns out this is relatively simple to do when compared to finding the optimal split.

Lloyd's algorithm, or k-means [DH73, Llo82, BH67], is based on a different approach. It computes a clustering to k clusters without using distances between points. Instead, it finds a partitioning $\mathcal{C}$ of the input points to k groups D_i so that the cost

$$\sum_{i=1}^{k} \sum_{x \in D_i} d(x, \mathrm{centroid}(D_i)) \tag{4.1}$$

is minimized. Here d is a distance function and centroid(D_i) refers to a "center point" of the cluster D_i. Typically one uses the mean as the centroid and squared Euclidean distance as d. This is also called the *reconstruction error*. The algorithm is given in Algorithm 2. On every iteration k-means updates the clustering by assigning each point $x \in D$ to the cluster with the closest centroid. The algorithm terminates when the clustering error no longer decreases. Note that the resulting clustering may not be a global optima of (4.1), but the algorithm can end up at a local minimum.

Problems with chains

Both hierarchical approaches and Lloyd's algorithm require a distance function. In the case of hierarchical clustering we must be able to compute distances between two points in the input, while with k-means we have to compute a distance to a centroid, which is usually a point as well.

Defining a good distance function for chains is not straightforward. For example, if we have the chains $1 < 4 < 5$ and $2 < 3 < 6$, it is not easy to say anything about their similarity, as they share no common items. We will return to this problem later in Section 4.3, for now it is sufficient to remember that defining the distance between two chains is not trivial.

This makes the use of agglomerative techniques hard. Lloyd's algorithm is more interesting, as we do not need to compute distances between two chains, but only consider the distance to the cluster centroid. This centroid does not have to be a chain, which makes defining a good distance function easier.

Thus, instead of using a chain for the cluster centroid, we can use a total order. This is the approach taken in [KA06]. Recall that the Kendall distance between permutations π_1 and π_2 is defined as the number of disconcordant pairs (u, v) normalized by the total number of pairs. For permutations of M (with m items) this is

$$d_K(\pi_1, \pi_2) = \big(\frac{1}{2}m(m-1)\big)^{-1} \sum_{u \in M} \sum_{v \in M} I\{(u, v) \in \pi_1 \wedge (v, u) \in \pi_2\}.$$

If either of π_1 or π_2 is a chain, we can still use almost the same definition. Let $M(\pi)$ denote the subset of M that is covered by the chain π. Now we can only compare the items in $\tilde{M} = M(\pi_1) \cap M(\pi_2)$, and have

$$d_K(\pi_1, \pi_2) = \big(\frac{1}{2}h(h-1)\big)^{-1} \sum_{u \in \tilde{M}} \sum_{v \in \tilde{M}} I\{(u, v) \in \pi_1 \wedge (v, u) \in \pi_2\}, \tag{4.2}$$

where $h = |\tilde{M}|$. This definition is of course meaningful only if $h \geq 2$. When $h < 2$ we can either say that $d_K(\pi_1, \pi_2)$ is not defined, or let $d_K(\pi_1, \pi_2) = d'$, where $d' \in [0, 1]$ is some predefined constant. If either of π_1 or π_2 is a total order we always have l common items as the total order is guaranteed to contain everything in M.

The problem of using a total order for representing the centroid is that one basically has to solve the rank aggregation problem discussed in Chapter 2. Given all chains belonging to the cluster C_i, we have to compute a total order that is the "average" of C_i. This is not trivial, but can be solved by several different approaches, some of which have theoretical performance guarantees, and some of which are just heuristics that happen to give reasonable results in practice. One option is to use the Bucket Pivot algorithm presented in the previous chapter. By setting $\beta = 0$ the algorithm outputs a total order instead of a bucket order. In [KA06] two different approaches are proposed for computing the centroid. They perform in practice almost equally well in the experiments given in [KA06]. It is not very well understood how the quality of the resulting clustering is affected by the method used to compute the centroid.

Our main objective, however, is to consider the case where the centroid is not a total order. The main motivation for this is that we want to avoid having to solve the rank aggregation problem for each cluster on each iteration of Lloyd's algorithm. The details of our approach are discussed next.

4.2 Distances and centroids

The algorithm we propose for clustering chains is a variant of Lloyd's algorithm (see Algorithm 2), adopted for chains. As pointed out in the previous section, there are some problems associated with using traditional clustering techniques with chains. The distance between two chains and the centroid of a set of chains are both not easily defined. Lloyd's algorithm can be implemented without a distance function for the items that are clustered, but requires the computation of a cluster centroid to which the distances can be efficiently calculated. We discuss next the choice of the centroid, and how this depends on the distance measure we use.

First consider the following general definition of a centroid. Given a set D of items and the class Q of centroids for D, we want to find a $X^* \in Q$, so that

$$X^* = \arg\min_{c \in Q} \sum_{x \in D} d(x, c), \tag{4.3}$$

where $d(x, c)$ is the distance between x and c. Intuitively X^* must thus reside at the "center" of the set D.

We let Q be set of all $|M| \times |M|$ matrices that satisfy the probability constraint ($X(u,v) + X(v,u) = 1$ for all $u, v \in M$). Given a matrix $X \in Q$ and a chain π, the distance $d(\pi, X)$ of π and X is defined by

$$d(\pi, X) = \sum_{(u,v) \in \pi} X(v,u). \tag{4.4}$$

Note that a total order τ can be represented as a matrix $X \in Q$ simply by letting $X(u,v) = 1$ and $X(v,u) = 0$ for all $(u,v) \in \tau$. In this case Equation 4.4 is equivalent to Equation 4.2.

To find the centroid of a given set D of chains, we must find now a matrix $X \in Q$ such that the cost

$$c(X, D) = \sum_{\pi \in D} \sum_{(u,v) \in \pi} X(v,u)$$

is minimized. By writing the sum in terms of pairs of items instead of chains, we obtain

$$c(X, D) = \sum_{u \in M} \sum_{v \in M} C_D(u,v) X(v,u),$$

where $C_D(u,v)$ denotes the number of chains in D where u appears before v. Let U denote the set of all unordered pairs of items from M. Using U the above can be written as

$$c(X, D) = \sum_{\{u,v\} \in U} \big(C_D(u,v) X(v,u) + C_D(v,u) X(u,v) \big).$$

As X must satisfy the probability constraint, this becomes

$$c(X, D) = \sum_{\{u,v\} \in U} \big(C_D(u,v)(1 - X(u,v)) + C_D(v,u) X(u,v) \big).$$

To minimize this it is enough to independently minimize the individual parts of the sum corresponding to the pairs in U. These are convex combinations of $C_D(u,v)$ and $C_D(v,u)$, and will be minimized when we let $X(u,v) = 1$ when $C_D(u,v) > C_D(v,u)$ and $X(u,v) = 0$ (meaning $X(v,u) = 1$) when $C_D(u,v) < C_D(v,u)$. Or, in other terms, we get X by rounding the values in the normalized pair order matrix $\hat{C}_D$ to 0s and 1s. Note that the resulting matrix does not necessarily represent a total order, as cycles can be introduced by the rounding.

A drawback when using X as defined above is that the distances are all integer valued. This may increase the probability that two centroids are of equal distance from a chain, and we have to make the assignment to one of them arbitrarily (see statement inside the **for**-loop of Algorithm 2). This can be avoided by using the matrix $\hat{C}_D$ directly as the cluster centroid. If instead of the distance given in Equation 4.4 we use the definition

$$d'(\pi, X) = \sum_{(u,v)\in\pi} X(v,u)^2, \tag{4.5}$$

$X^* = \hat{C}_D$ will satisfy Equation 4.3. This is easily seen by noting that now we must minimize

$$c(X, D) = \sum_{\{u,v\}\in U} \underbrace{\left(C_D(u,v)(1 - X(u,v))^2 + C_D(v,u)X(u,v)^2\right)}_{c(X,\{u,v\})}.$$

Again the individual parts of the sum can be minimized independently. The first derivative of $c(X, \{u, v\})$ with respect to $X(u, v)$ is

$$-2C_D(u,v)(1 - X(u,v)) + 2C_D(v,u)X(u,v).$$

Setting this equal to zero gives

$$X(u,v) = \frac{C_D(u,v)}{C_D(u,v) + C_D(v,u)} = \hat{C}_D(u,v).$$

4.3 Mappings to vector spaces

So far we have approached the problem of by suggesting ways to make Lloyd's algorithm usable directly with the chains in our input. In this section we use a different strategy and discuss two ways to represent chains in a vector space. This makes it possible to compute the clustering using *any* algorithm that works with vectors. A clustering obtained in this way does obviously not minimize the same objective function as the algorithm discussed in the previous section. If this is not desired, we can use the vector space representation to compute an initial clustering of the chains, and then refine this with Lloyd's algorithm using the centroid and distance function discussed above.

Note that this is only one possible application of the vector representation. In Chapter 5 we make use of the vectors for creating scatterplots of sets of chains. Also other tasks, for example nearest neighbor queries or classification problems, can be solved for inputs that consist of chains using these representations.

The first mapping we describe in Section 4.3 is based on using the adjacency matrices of two graphs with the chains in the input D as vertices. These graphs can be seen as special cases of the so called Planted Partition Model (see e.g. [CK01, ST02]). In Section 4.3 we discuss an alternative technique that maps chains to points on the surface of a high-dimensional hypersphere.

Graph representation

Motivation

We start the discussion by returning to the question of computing the distance between two chains. Both Spearman's footrule and Kendall distances can be modified for chains so that they only consider the common items. If the chains π_1 and π_2 have no items in common, we have to fix some distance between π_1 and π_2. This is done for example in [KF03], where the distance between two chains is given by $1 - \rho$, where $\rho \in [-1, 1]$ is Spearman's rank correlation. For two fully correlated chains the distance becomes in this case 0, and for chains with strong negative correlation the distance is 2. If the chains have no common items we have $\rho = 0$ and the distance is 1.

We could use the same approach also with the Kendall distance: The distance of the chains π_1 and π_2 is the Kendall distance of the permutations induced by the common items in π_1 and π_2 normalized to the interval $[0, 1]$. If there are no common items we set simply the distance to 0.5. Now consider the following example. Let $\pi_1 = (1\ 2\ 3\ 4\ 5)$, $\pi_2 = (6\ 7\ 8\ 9\ 10)$, and $\pi_3 = (4\ 8\ 2\ 5\ 3)$. By definition we have $d_K(\pi_1, \pi_2) = 0.5$, and a simple calculation gives $d_K(\pi_1, \pi_3) = 0.5$ as well. Without any additional information this is a valid approach.

However, as we are clustering the chains, we can take into account the underlying orders (total or partial) that have generated the set of chains D. See Section 2.1 for a description of the model that we assume has generated the set of chains. For example, let us assume that the components are total orders, and that π_1 and π_2 have been emitted by the same component, the total order $(1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10)$, and that π_3 is generated by another component, the total order $(6\ 7\ 9\ 10\ 4\ 8\ 2\ 5\ 3\ 1)$. Under this assumption it no longer appears meaningful to have $d_K(\pi_1, \pi_2) = d_K(\pi_1, \pi_3)$, as the clustering algorithm should separate chains generated by different components from each other. We would like to have $d_K(\pi_1, \pi_2) < d_K(\pi_1, \pi_3)$ in this case. Of course we can a priori not know the underlying components, but when computing a clustering of the set of chains we are implicitly assuming that they exist.

Agreement and disagreement graphs

Next we propose a method for mapping the chains to $\mathbb{R}^m$ so that the distances between the vectors that correspond to π_1, π_2 and π_3 satisfy the inequality above. In general, we would like to have chains that are generated by the same component to have a shorter distance to each other than to chains that are generated by other components. To this end, we define the distance between two chains in D as the distance between their neighborhoods in appropriately constructed graphs. If the neighborhoods are similar, i.e., there are many chains in D that are (in a sense to be formalized shortly) “close to” both π_1 and π_2, we consider also π_1 and π_2 similar to each other. Note that this definition of distance between two chains is dependant on the input D, i.e., we obtain different distances for π_1 and π_2 by changing the other chains in D.

We say that chains π_1 and π_2 *agree* if for some u and v we have $(u, v) \in \pi_1$ and $(u, v) \in \pi_2$. Likewise, the chains π_1 and π_2 *disagree* if for some u and v we have $(u, v) \in \pi_1$ and $(v, u) \in \pi_2$. Note that π_1 and π_2 can simultaneously both agree and disagree. We define two graphs as follows: Let $G_a(D)$ be the *agreement graph* of the set D, and let $G_d(D)$ be the *disagreement graph* of the set D. Both $G_a(D)$ and $G_d(D)$ are undirected graphs with chains in D as vertices. In the agreement graph $G_a(D)$ two vertices are connected by an edge if their respective chains *agree and do not disagree.* In the disagreement graph $G_d(D)$ two vertices are connected by an edge if their respective chains *disagree but do not agree.*

The Planted Partition Model

Consider the following stochastic model for creating a random graph of n vertices. First partition the set of vertices to k disjoint subsets denoted $V_1, \ldots, V_k$. Then, independently generate edges between the vertices as follows: add an edge between two vertices that belong to the same subset with probability p, and add an edge between two vertices that belong to different subsets with probability $q < p$. This specific model was first discussed in [CK01] and subsequently in [ST02].

Under some simplified conditions both the agreement graph $G_a(D)$ and disagreement graph $G_d(D)$ can be seen as instances of the planted partition model. Consider the graph $G_a(D)$, and let the chains be generated by a mixture of k random total orders on the set of items M, so that each chain is the projection of one of the total orders on some l-sized subset of M. In this case we have

$$p = \binom{m}{l}^{-1} \sum_{i=2}^{l} \binom{l}{i} \binom{m-l}{l-i}, \text{ and} \tag{4.6}$$

$$q = \binom{m}{l}^{-1} \sum_{i=2}^{l} \frac{\binom{l}{i}\binom{m-l}{l-i}}{i!}. \tag{4.7}$$

These can be derived as follows. Consider the case where we first pick the chain π_1 from component j. This is a subset of M of size l ordered according to component j. The probability p that a second chain π_2 generated by component j agrees with π_1 is the probability that π_1 and π_2 share at least two items. That is, p is given by the sum of the possible choices of items into π_2 that have at least 2 items in common with π_1, divided by the total number of choices of l-sized subsets of M. In the case of q the chain π_2 is picked from the component $j' \neq j$. It can happen

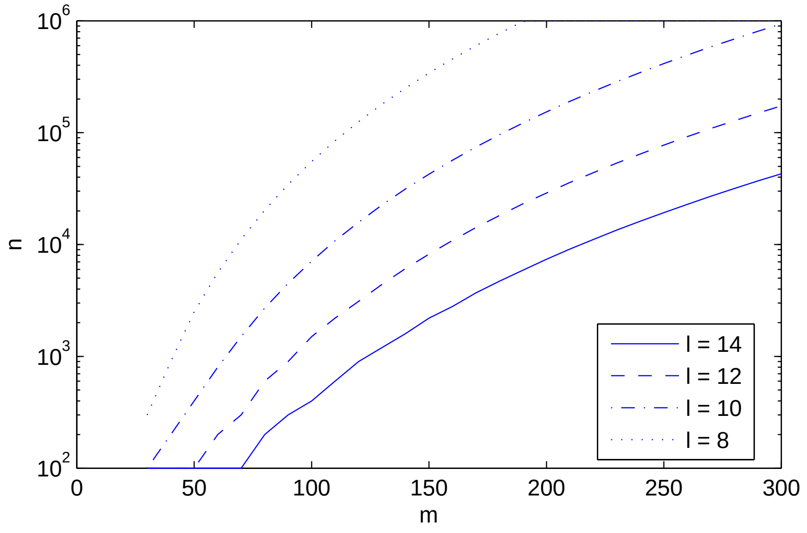


Figure 4.1: Minimum size n of the input as a function of m for different l when using the algorithm of [ST02] for computing the clustering. (Assuming the clusters are of equal size.) The numbers on the y-axis are not absolute, instead the purpose of the curves is to indicate how the minimum input size behaves as m increases.

that π_2 agrees with π_1, but in that case all the common items must be ordered exactly as in π_1. To take this into account we divide the product of the binomial coefficients with the factorial of i, as only one of the possible permutations of the common items is allowed.

Both [CK01] and [ST02] present algorithms that find the correct clustering with high probability, provided the gap $\Delta = p - q$ is not too small. According to [ST02], for input D of size n, the gap Δ must be $\Omega(n^{-\frac{1}{2}+\epsilon})$, for the algorithm of [CK01] to work, and of order $\Omega(kn^{-\frac{1}{2}} \log n)$ for their improved algorithm to find the correct partitioning. Moreover, these bounds hold only for equal sized clusters.

Using Equations 4.6 and 4.7 we can compute the value of the gap $\Delta = p - q$ for different values of m and l. Moreover, we can check for what value of n (size of the input) the algorithm of, say, [ST02] will work with the given value of Δ. In Figure 4.1 we have plotted n as a function of m for different values of l. For instance, if $m = 200$ and $l = 10$ (still a reasonable combination of the parameters in practical applications) we need tens of thousands of chains in the input for the algorithm of [ST02] to find the correct partitioning with high probability. In practice n can be one or two orders of magnitude smaller. The effect of l is also considerable; when m is about 100, n increases by an order of magnitude when l decreases by 2.

We can also study how the gap Δ behaves as a function of m and l. Since we have

$$\Delta = p - q = \frac{\sum_{i=2}^{l} \binom{l}{i}\binom{m-l}{l-i}\left(1 - \frac{1}{i!}\right)}{\binom{m}{l}},$$

where $(1 - \frac{1}{i!})$ is significantly less than 1 only for very small i (say, $i \leq 3$), it is reasonable to bound Δ by using an upper bound for p. We obtain the following theorem:

Theorem 4.3.1 *For the simple model under which Equations 4.6 and 4.7 hold, we have*

$$\Delta < p = O\left(\frac{l^2}{m}\right).$$

Proof See Appendix A. □

This bound partially explains the strong effect of l in Figure 4.1. Also, it gives some intuition how the density of the agreement graph behaves when we modify m and l.

Using $G_a(D)$ and $G_d(D)$

In the agreement graph, under ideal circumstances the chain π is mostly connected to chains generated by the same component as π. Also, it is easy to see that in the disagreement graph the chain π is (again under ideal circumstances) not connected to any of the chains generated by the same component, and only to chains generated by the other components. This latter fact makes it possible to find the correct clustering by finding a k-coloring of $G_d(D)$. Unfortunately this has little practical value as in real data sets we expect to observe noise that will distort both $G_a(D)$ and $G_d(D)$.

Above we argued that representations of two chains emitted by the same component should be more alike than representations of two chains emitted by different components. Consider the case where $k = 2$ and both clusters are of size $n/2$. Let $f_a(\pi) \in \mathbb{R}^n$ be the row of the adjacency matrix of $G_a(D)$ that corresponds to chain π. Let chain π_1 be generated by the same component as π, and let π_2 be generated by a different component. If the similarity s between $f_a(\pi)$ and $f_a(\pi')$ is simply the number of elements where $f_a(\pi) = f_a(\pi') = 1$, we have the following:

$$\begin{aligned} E[s(f_a(\pi), f_a(\pi_1))] &= \frac{n}{2}p^2 + \frac{n}{2}q^2 = \frac{n}{2}(p^2+q^2), \\ E[s(f_a(\pi), f_a(\pi_2))] &= \frac{n}{2}pq + \frac{n}{2}qp = nqp. \end{aligned}$$

If we let $p = cq$, with $c > 1$, it is easy to see that $E[s(f_a(\pi), f_a(\pi_1))] > E[f_a(\pi), f_a(\pi_2)]$. This is true if p and q are defined as in Equations 4.6 and 4.7. Therefore, at least under these simple assumptions the expected distance between two chains from the same component is always less than the expected distance between two chains from different components.

In practice we can combine the adjacency matrices of $G_a(D)$ and $G_d(D)$ to create the final mapping. Let $G_{ad} = G_a(D) - G_d(D)$, where $G_a(D)$ and $G_d(D)$ denote the adjacency matrices of the agreement and disagreement graphs. The representation of the chain π in $\mathbb{R}^n$ is simply the vector of G_{ad} that corresponds to π.

Using the agreement and disagreement graphs has the obvious drawback that the adjacency matrices of $G_a(D)$ and $G_d(D)$ are both of size $n \times n$, and computing one entry takes time proportional to l^2. Even though $G_a(D)$ and $G_d(D)$ have the theoretically nice property of being generated by the Planted Partition Model, using them in practice can be prohibited by the scalability issues. In Section 4.4 we consider an approach where the matrix G_{ad} is not computed fully, but only a fraction of it's columns are used for clustering.

Hypersphere representation

Next we discuss a method for mapping chains to an m-dimensional vector space. The mapping can be computed in time $O(|D|)$. This method has a slightly different motivation than the one discussed above. Let f be the mapping from the set of all chains to $\mathbb{R}^m$ and let d be a distance function in $\mathbb{R}^m$. Furthermore, let π be a chain and denote by π^R the reverse of π, i.e., the chain that orders the same items as π, but in exactly the opposite way. The mapping f and distance d should satisfy

$$\begin{aligned} \pi^R &= \arg\max_{\pi'} d(f(\pi), f(\pi')) \quad \text{for all } \pi, && (4.8) \\ d(f(\pi_1), f(\pi_1^R)) &= d(f(\pi_2), f(\pi_2^R)) \quad \text{for all } \pi_1 \text{ and } \pi_2. && (4.9) \end{aligned}$$

Less formally, we want the reversal of a chain to be furthest away from it in the vector space (4.8), and the distance between π and π^R should be the same for all chains (4.9). We first define a mapping for total orders and then generalize this for chains. In both cases the mappings have a nice geometrical interpretation.

A mapping for total orders

We define a function f that maps total orders to $\mathbb{R}^m$ as follows: Let π be a permutation of M, and let $\pi(u)$ denote the position of $u \in M$ in π. Define the vector $\mathbf{f}_\pi$ so that

$$\mathbf{f}_\pi(u) = -\frac{m+1}{2} + \pi(u) \tag{4.10}$$

for all $u \in M$. We define the mapping f such that $f(\pi) = \mathbf{f}_\pi / \|\mathbf{f}_\pi\| = \hat{\mathbf{f}}_\pi$.

For example, if $M = \{1, \ldots, 8\}$ and $\pi = (5, 1, 6, 3, 7, 2, 8, 4)$, then according to Equation 4.10

$$\mathbf{f}_\pi = (-2.5, 1.5, -0.5, 3.5, -3.5, -1.5, 0.5, 2.5),$$

and as $\|\mathbf{f}_\pi\| = 6.48$, we have

$$f(\pi) = \hat{\mathbf{f}}_\pi = (-0.39, 0.23, -0.08, 0.54, -0.54, -0.23, 0.08, 0.39).$$

When d is the *cosine distance* between two vectors, which in this case is simply $1 - \hat{\mathbf{f}}_\pi^T \hat{\mathbf{f}}_{\pi'}$ as the vectors are normalized, it is straightforward to check that $\hat{\mathbf{f}}_\pi$ satisfies equations 4.8 and 4.9. This mapping has a geometrical interpretation: all permutations are points on the surface of an m-dimensional unit-sphere centered at the origin. Moreover, the permutation π and its reversal π^R are on exactly opposite sides of the sphere. That is, the image of π^R is found by mirroring the image of π at the origin.

A mapping for chains

To extend the above for chains we employ the technique used e.g. in [Cri85] and also in [Zha04]. The idea is to represent a chain τ on M by the set of permutations on M that are compatible with τ. That is, we view τ as a partial order on M and use the set of linear extensions of τ to construct the representation $f(\tau)$. More precisely, we want $f(\tau)$ to be the *center* of the points in the set $\{f(\pi) : \pi \in \mathcal{E}(\tau)\}$, where f is the mapping for permutations given in the definition above, and $\mathcal{E}(\tau)$ is the set of linear extensions of τ. Despite the fact that the size of $\mathcal{E}(\tau)$ is $\binom{m}{l}(m-l)!$, we can compute $f(\tau)$ very efficiently. We start by giving a definition for $f(\tau)$ that is not related to $\mathcal{E}(\tau)$ in any way. Then, in Theorem 4.3.2 we show that this definition indeed leads $f(\tau)$ to be the center of $\mathcal{E}(\tau)$.

We define a function f that maps chains to vectors in $\mathbb{R}^m$ as follows: Let τ be a chain and let $\mathbf{f}_\pi$ be defined as in Equation 4.10. Define the vector $\mathbf{f}_\tau$ so that

$$\mathbf{f}_\tau(u) = \begin{cases} -\frac{|\tau|+1}{2} + \tau(u) & \text{iff } u \in \tau, \\ 0 & \text{iff } u \notin \tau, \end{cases} \tag{4.11}$$

for all $u \in M$. Finally, we define the mapping f such that $f(\tau) = \mathbf{f}_\tau / \|\mathbf{f}_\tau\| = \hat{\mathbf{f}}_\tau$.

It is not trivial to see that a mapping based on Equation 4.11 indeed results in what we outlined above. We give the following theorem.

Theorem 4.3.2 *If the vector $\mathbf{f}_\tau$ is defined as in Equation 4.11, then for some constant Q we have*

$$\mathbf{f}_\tau(u) = Q \sum_{\pi \in \mathcal{E}(\tau)} \mathbf{f}_\pi(u) \tag{4.12}$$

for all $u \in M$.

Before looking at the proof, let us comment what the statement of the theorem means. We want $f(\tau)$ to be the mean of the points that represent the linear extensions of τ, normalized to unit length. Theorem 4.3.2 states that this mean has a simple explicit formula that is given by Equation 4.11. Thus, when normalizing $\mathbf{f}_\tau$ we indeed get the normalized mean vector without having to sum over all linear extensions of τ. This is very important, as $\mathcal{E}(\tau)$ is so large that simply enumerating all its members is not computationally feasible.

Proof (of Theorem 4.3.2) We start by showing that the claim of Equation 4.12 holds for all u that belong to τ. Basically we will show that

$$\sum_{\pi \in \mathcal{E}(\tau)} f_\pi(u) = Q\big(-\frac{|\tau|+1}{2} + \tau(u)\big). \tag{4.13}$$

First, note that $\sum_{\pi \in \mathcal{E}(\tau)} f_\pi(u)$ can be rewritten as follows

$$\sum_{\pi \in \mathcal{E}(\tau)} -\frac{m+1}{2} + \pi(u) = \sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} \#\{\pi(u) = i\}\Big(-\frac{m+1}{2} + i\Big), \tag{4.14}$$

where $\#\{\pi(u) = i\}$ denotes the number of times u appears at position i in the linear extensions of τ. The sum is taken over the range $\tau(u), \ldots, m - |\tau| + \tau(u)$, as $\pi(u)$ can not be less than $\tau(u)$, because the items that appear before u in τ must appear before it in π as well, likewise for the other end of the range.

To see what $\#\{\pi(u) = i\}$ is, consider how a linear extension π of τ is structured. When u appears at position i in π, there are exactly $\tau(u) - 1$ items belonging to τ that appear in the $i - 1$ indices to the left of u, and $|\tau| - \tau(u)$ items also belonging to τ that appear in the $m - i$ indices to the right of u. The ones on the left may choose their indices in $\binom{i-1}{\tau(u)-1}$ different ways, while the ones on the right may choose their indices in $\binom{m-i}{|\tau|-\tau(u)}$ different ways. The remaining items that do not belong to τ are assigned in an arbitrary fashion to the remaining $m - |\tau|$ indices. We have thus,

$$\#\{\pi(u) = i\} = \binom{i-1}{\tau(u)-1}\binom{m-i}{|\tau|-\tau(u)}(m-|\tau|)!.$$

When this is plugged into the right side of (4.14), and after rearranging the terms slightly, we get

$$\sum_{\pi\in\mathcal{E}(\tau)} f_\pi(u) = (m-|\tau|)! \sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} \binom{i-1}{\tau(u)-1}\binom{m-i}{|\tau|-\tau(u)}\Big(-\frac{m+1}{2}+i\Big).$$

This can be written as

$$\sum_{\pi\in\mathcal{E}(\tau)} f_\pi(u) = (m-|\tau|)!(S_1+S_2), \tag{4.15}$$

where

$$\begin{aligned} S_1 &= -\frac{m+1}{2}\sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} \binom{i-1}{\tau(u)-1}\binom{m-i}{|\tau|-\tau(u)}, \quad \text{and} \\ S_2 &= \sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} i\binom{i-1}{\tau(u)-1}\binom{m-i}{|\tau|-\tau(u)}. \end{aligned}$$

Let us first look at S_2. The part $i\binom{i-1}{\tau(u)-1}$ can be rewritten as follows:

$$\begin{aligned} i\binom{i-1}{\tau(u)-1} &= \frac{i\ (i-1)!}{(\tau(u)-1)!(i-\tau(u))!}\cdot\frac{\tau(u)}{\tau(u)} \\ &= \tau(u)\frac{i!}{\tau(u)!(i-\tau(u))!} \\ &= \tau(u)\binom{i}{\tau(u)}. \end{aligned}$$

This gives

$$S_2 = \tau(u)\sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} \binom{i}{\tau(u)}\binom{m-i}{|\tau|-\tau(u)} = \tau(u)\binom{m+1}{|\tau|+1},$$

where the second equality is based on Equation 5.26 in [GKP94]. Next we must show that $\binom{m+1}{|\tau|+1}$ will appear in S_1 as well. We can rewrite the sum as follows:

$$\sum_{i=\tau(u)}^{m-|\tau|+\tau(u)} \binom{i-1}{\tau(u)-1}\binom{m-i}{|\tau|-\tau(u)} = \sum_{i=\tau(u)-1}^{m-|\tau|+\tau(u)-1} \binom{i}{q}\binom{r-i}{p-q},$$

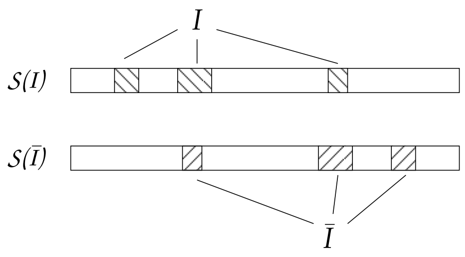


Figure 4.2: Permutations in $S(I)$ have the positions I occupied by items that belong to the chain τ, while permutations in $S(\bar{I})$ have the positions $\bar{I}$ occupied by items of τ. See proof of Theorem 4.3.2.

where $q = \tau(u) - 1$, $r = m - 1$ and $p = |\tau| - 1$. Again we apply Equation 5.26 of [GKP94] to get

$$S_1 = -\frac{m+1}{2}\binom{r+1}{p+1} = -\frac{m+1}{2}\binom{m}{|\tau|},$$

which we multiply by $\frac{|\tau|+1}{|\tau|+1}$ and have

$$S_1 = -\frac{|\tau|+1}{2} \cdot \frac{m+1}{|\tau|+1}\binom{m}{|\tau|} = -\frac{|\tau|+1}{2}\binom{m+1}{|\tau|+1}.$$

When S_1 and S_2 are plugged into (4.15) we have

$$\sum_{\pi \in \mathcal{E}(\tau)} f_\pi(u) = (m - |\tau|)!\Big(-\frac{|\tau|+1}{2}\binom{m+1}{|\tau|+1} + \tau(u)\binom{m+1}{|\tau|+1}\Big),$$

which is precisely Equation 4.13 when we let $Q = (m - |\tau|)!\binom{m+1}{|\tau|+1}$.

To complete the proof we must still show that Equation 4.12 also holds for items u that do not appear in the chain τ. For such u we have $f_\tau(u) = 0$ by definition, and as we showed above that $Q > 0$, we have to show that $\sum_{\pi \in \mathcal{E}(\tau)} f_\pi(u) = 0$ to prove the claim.

It is simple to see that $f_\pi(u) + f_{\pi'}(u) = 0$ when $\pi'(u) = m - \pi(u) + 1$; the values at $\pi(u)$ and $m - \pi(u) + 1$ cancel each other out. We'll partition $\mathcal{E}(\tau)$ to disjoint groups defined by index sets I. Let $S(I)$ denote the set of all linear extensions of τ, where the items that belong to τ appear at indices $I = \{i_1, \ldots, i_{|\tau|}\}$. Furthermore, let $\bar{I} = \{m - i_1 + 1, \ldots, m - i_{|\tau|} + 1\}$. Note that these are the positions that would cancel out the values incurred by an item appearing in a position in I. See Figure 4.2 for an illustration of the structure of $S(I)$ and $S(\bar{I})$.

First, consider the positions *not* in $\{I \cup \bar{I}\}$. Every item $u \notin \tau$ appears equally many times in each of these positions in $S(I)$. To see this, remember that in all permutations of $m - |\tau|$ items, each item appears equally many times at every position. Hence, item u appears as many times at position $\pi(u) \notin \{I \cup \bar{I}\}$ as it appears at position $m - \pi(u) + 1$, meaning that for these positions the terms of $\sum_{\pi \in S(I)} f_\pi(u)$ cancel each other out.

The problem is that in $S(I)$ an item that does not belong to τ an appear at a position in $\bar{I}$ as well, but terms corresponding to these occurrences are not canceled out as the positions in I are occupied by items belonging to τ. Now consider the set $S(\bar{I})$. Here the situation is reversed: items not in τ appear also at positions I, but are not canceled out because the positions $\bar{I}$ are again "blocked" by items in τ. However, it is easy to see that the number of times an item $u \notin \tau$ appears at a position $i \in \bar{I}$ in $S(I)$, is the same as the number of times u appears in position $j \in I$ in $S(\bar{I})$. Thus, the occurrences in $S(I)$ and $S(\bar{I})$ cancel each other out. For an $u \notin \tau$ we can write

$$\sum_{\pi \in \mathcal{E}(\tau)} f_\pi(u) = \frac{1}{2} \sum_{I} \sum_{\pi \in \{S(I) \cup S(\bar{I})\}} f_\pi(u),$$

and we argued above that the inner sum on the right-hand side is always zero, the desired result follows. This concludes the proof of Theorem 4.3.2. □

One advantage of the hypersphere representation over using the mapping based on the agreement and disagreement graphs is speed. To compute the vectors $\mathbf{f}_\pi$ for all chains in the input is of order $O(nl)$, which is considerably less than the requirement of $O(n^2 l^2)$ for the graph based approach. Of course we lose the property of having a shorter distance between chains generated by the same component than between chains generated by different components. The second advantage is size. Storing the full graph representation requires $O(n^2)$ memory, while storing the hyperspere representation needs only $O(nm)$ of storage. This is less than $O(n^2)$ as we usually have $m \ll n$.

4.4 Experiments

In this section we discuss experiments based on both artificial and real data sets. We use the following combinations of techniques:

graph	Compute the graph representation of the input and use any implementation of k-means on this to obtain a clustering.
graph, rf	Same as *graph* but the returned clustering is refined using the iteration described in Section 4.2.
hyper	Compute the hypersphere representation of the input and use any implementation of k-means on this to obtain a clustering.
hyper, rf	Same as *hyper* but the initial clustering is refined using the iteration described in Section 4.2.

With *graph* and *hyper* we used the k-means implementation found in the Statistics Toolbox of Matlab with the cosine distance for both representations.

A comparison of the algorithms

We compare the algorithms first using the real data sets described in Section 2.1. The SUSHI and MLENS data sets are likely to be more difficult to cluster, as the total number of items is larger than in case of DUBLIN and MSNBC. In addition to our own algorithm we also run the same experiments with the algorithms presented in [KA06]. These algorithms, denoted TMSE and EBC, are similar clustering algorithms for sets of chains, but they are based on slightly different distance functions and type of centroid. Our algorithm was implemented in Matlab, while TMSE and ECB are the original C implementations obtained from the authors of [KA06].

We used the algorithms to compute a 2-way clustering of each data set, and computed the clustering error as defined in Equation 4.1. The experiment was repeated 25 times. The results given in Table 4.1 are averages of these. Estimates of the standard deviation of the error are given in parenthesis. The best performing algorithm is indicated in bold for each data set. The *dummy* algorithm on the top row is the error we obtain when all chains are put into the same cluster.

	SUSHI	MLENS	DUBLIN	MSNBC
dummy	94138	71092	21407	36964
graph	(99.90) 86122	(347.9) 66300	(46.40) 18636	(0.000) 34350
graph, rf	(137.0) 85113	(342.7) 66285	(510.9) 17116	(472.1) 32473
hyper	(77.70) 86179	(441.5) 66406	(272.7) 17219	(0.000) 33307
hyper, rf	(136.6) **84996**	(445.7) 66376	(26.30) **16848**	(46.40) **32363**
EBC	(427.6) 85761	(21.70) **65846**	(545.8) 17723	(182.6) 32888
TMSE	(338.4) 85058	(28.30) 65847	(150.6) 17202	(195.5) 32797

Table 4.1: Averages of the clustering cost (of 25 rounds with $k = 2$) for different algorithms and data sets. The smallest error for each data set is indicated in bold.

Of our algorithms *graph, rf* and especially *hyper, rf* outperform both TMSE and EBC in every case except mlens, where TMSE and EBC have a smaller error. Also, the estimated standard deviation is smaller with our algorithms in the cases where they perform better than the algorithms of [KA06].

Correctness of the returned clustering

The notion of "correctness" is difficult to define when it comes to clustering models. With real data we do in general not know the correct clusters. When comparing two different clusterings of the same data one typically considers the clustering with the smallest reconstruction error (as defined in Equation 4.1) as the better model. However, this is not guaranteed to be more correct than any other clustering. With artificial data we can generate the data so that it contains a known clustering and then compare this with the clusterings found by the algorithms.

To this end we need a way of measuring the similarity of two clusterings. One alternative is to compute all pairwise similarities between clusters in the first and second clustering. The similarity of two clusters can be defined simply as the fraction of points they have in common. Then these similarities are used to compute the best matching between the clusters in the first and second clustering. The similarity of the clusterings is given by the weight of this matching. Ideally, when the clusterings are identical and the similarity between two clusters is defined as above, there exists a matching with weight equal to k. This is because the similarity of all pairs of clusters in the matching is 1.

While this is a reasonable approach, it has the drawback of not being very elegant. One has to first compute all pairwise similarities and then find the optimal matching based on these. Alternatively we can compare two clusterings $\mathcal{C}_1$ and $\mathcal{C}_2$ by considering pairs of points $\{x_i, x_j\}$, $x_i, x_j \in D$. Let $P_s(\mathcal{C}) = \{\{x_i, x_j\} | \mathcal{C}(x_i) = \mathcal{C}(x_j)\}$ be the set of pairs where both points belong to the same cluster in clustering $\mathcal{C}$. Likewise, let $P_d(\mathcal{C}) = \{\{x_i, x_j\} | \mathcal{C}(x_i) \neq \mathcal{C}(x_j)\}$ be the set of pairs where the points belong to different clusters in clustering $\mathcal{C}$. We now define the similarity of two clusterings $\mathcal{C}_1$ and $\mathcal{C}_2$ as follows:

$$sim(\mathcal{C}_1, \mathcal{C}_2) = \frac{|P_s(\mathcal{C}_1) \cap P_s(\mathcal{C}_2)| + |P_d(\mathcal{C}_1) \cap P_d(\mathcal{C}_2)|}{0.5n(n-1)}, \tag{4.16}$$

where $n = |D|$. This is the fraction of pairs that behave in the same way in both clusterings. Note that $P_s(\mathcal{C}) + P_d(\mathcal{C}) = 0.5n(n-1)$ for any $\mathcal{C}$.

We create artificial sets of chains with the procedure described in Section 2.1 on page 6. Parameters that we vary are l, the number of items m, and cs, which is the number of chains generated by a single component. The total size of the input is thus $k \cdot cs$. Additional parameters are the number of components k and the number of buckets in a bucket order, denoted b. We fix $k = 2$ and $b = 10$, and let $cs \in \{200, 400, 800\}$, $l \in \{3, 4, \ldots, 9\}$ and $m \in \{20, 50, 100\}$. Of each combination we create a random set of chains, compute a clustering of it using the three different algorithms, and record the similarity of the found clustering with the true clustering as defined in Equation 4.16. This is repeated 25 times for every combination.

Results are shown in Figure 4.3. Performance of the algorithm based on a random initialization (graph representation, hypersphere representation) is plotted with a dotted (dashed, solid) line, respectively. The first observation is that in every case the performance of the algorithms increases considerably as the length of the chains increases. With $l = 9$ the algorithms find the original clustering almost always independent of the values of m and cs. Also m and cs have an effect on the quality of the found clustering. The problem becomes more difficult as m increases, and slightly easier when there is more data available (as cs increases). The second observation is that the hypersphere representation seems to lead to the best results in every case. Also, the effect of the parameters to it's performance is not as strong as it is to the two other approaches. Also, maybe somewhat surprisingly the random initialization leads to equally an good performance as using the graph representation for computing the initial clustering.

Reducing the size of the graph representation

The graph representation requires $O(n^2)$ space if implemented as described in Section 4.3. This will in practice lead to a poor performance since we have to cluster n-dimensional vectors. *Dimension reduction algorithms* are discussed later in Chapter 5, for now we are satisfied with a very simple approach to speed up the clustering.

Recall that G_{ad} is the matrix we obtain when the adjacency matrix of the disagreement graph is subtracted from the adjacency matrix of the agreement graph. Let $\mathbf{x_i}$ be the ith row of G_{ad}; this is the graph representation of the ith chain in the input. Every dimension of $\mathbf{x_i}$ corresponds thus to a chain in the input. We have $\mathbf{x_i}(j) = 1$, $\mathbf{x_i}(j) = -1$, or $\mathbf{x_i}(j) = 0$, if the ith and jth chain of the input D agree, disagree, or have no relation to each other, respectively. When comparing two chains, we compare the vectors $\mathbf{x_i}$ and $\mathbf{x_j}$. We are thus comparing the two chains by considering how they relate to every other chain in the input. An easy way of reducing the dimensionality of $\mathbf{x_i}$ and $\mathbf{x_j}$ is to use only a subset of the input when comparing the chains. That is, we use only a subset of the columns of the matrix G_{ad}. This subset is chosen uniformly at random.

In this experiment we investigate how the running time of the clustering and the quality of the output (in terms of the error function) behave when we decrease the fraction of columns of G_{ad} that are used. The fraction of retained columns

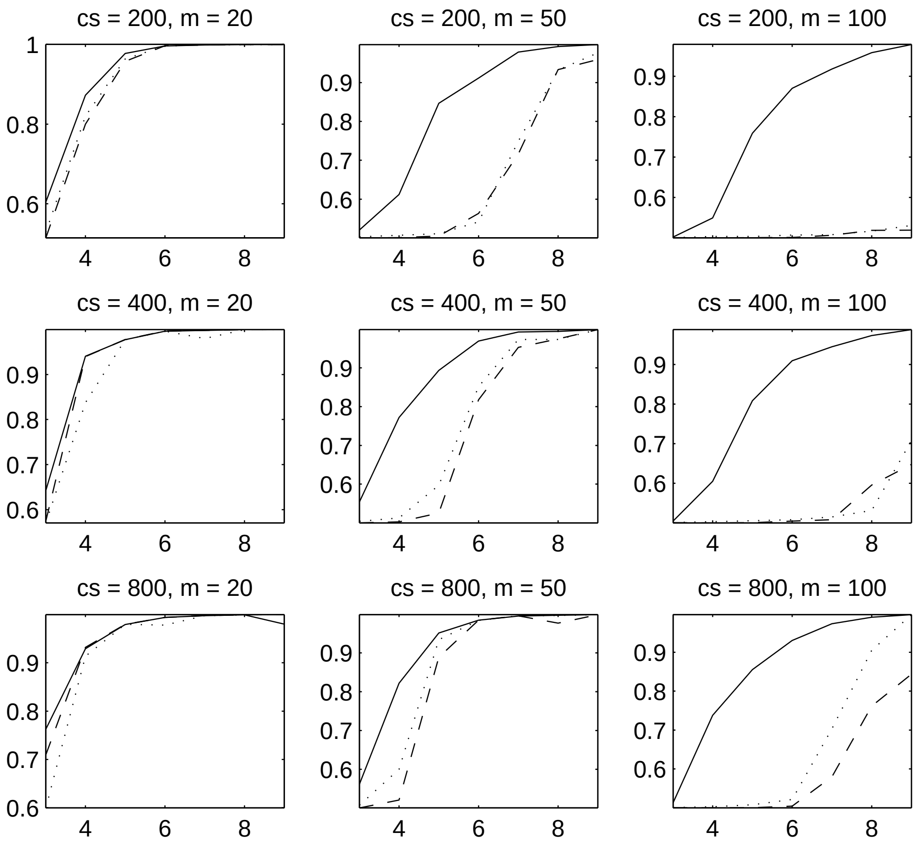


Figure 4.3: Curves showing the similarity of a clustering found by the algorithms with the true clustering (Equation 4.16) as a function of the length of the chain.

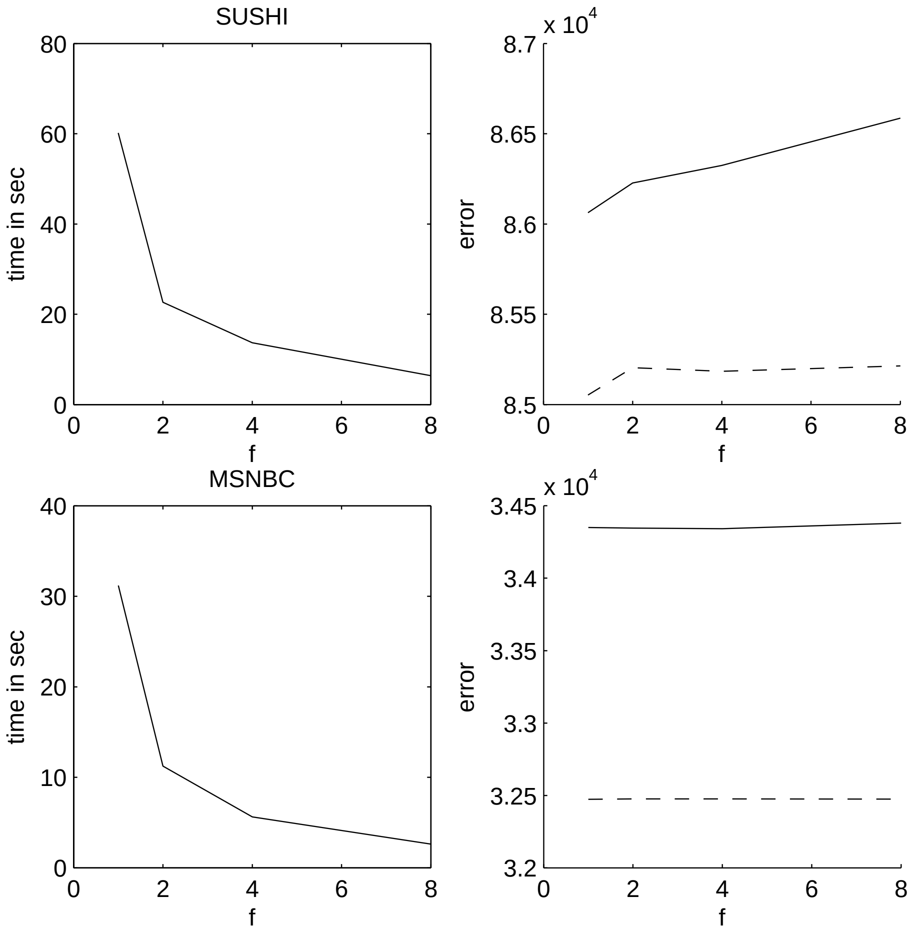


Figure 4.4: Clustering the SUSHI and MSNBC data sets with the graph representation using a random fraction of $1/f$ of the columns of the matrix G_{ad}. Left: Running time of k-means in seconds as a function of f. The time required for constructing G_{ad} is not included. Right: Clustering error as a function of f for using only the graph representation (solid line) and subsequently refining the model with Lloyd's algorithm (dashed line).

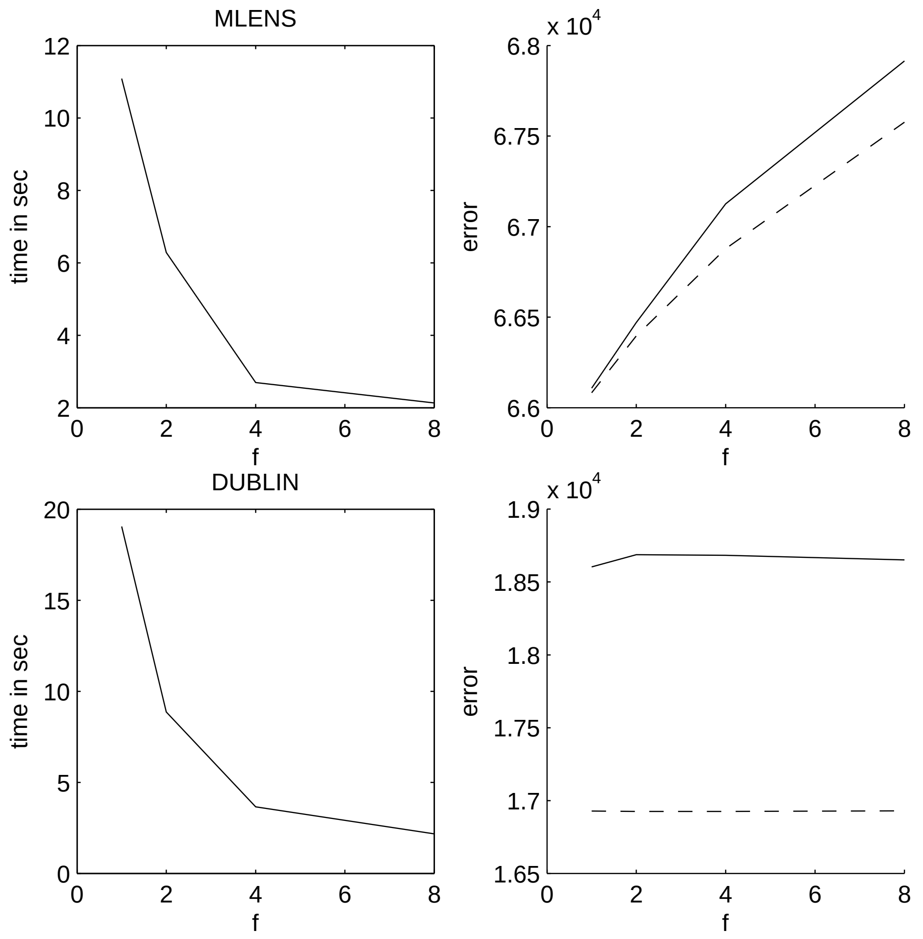


Figure 4.5: Clustering the MLENS and DUBLIN data sets with the graph representation using a random fraction of $1/f$ of the columns of the matrix G_{ad}. Left: Running time of k-means in seconds as a function of f. The time required for constructing G_{ad} is not included. Right: Clustering error as a function of f for using only the graph representation (solid line) and subsequently refining the model with Lloyd's algorithm (dashed line).

is given by $1/f$ where $f \in \{1, 2, 4, 8\}$. We use the two phased algorithm *graph, rf*, where an initial clustering is computed by applying regular k-means to the graph representation of the input, and this is further refined by applying Lloyd's iteration with the distance and centroid described in Section 4.2. The errors of the initial and final clustering are both considered. For each value of f the experiment is repeated 25 times. The reported results are averages over these runs. In Figures 4.4 and 4.5 we can see how the running time and error behave for different values of f with the four data sets described in Section 2.1.

The computation time is reduced by an order of magnitude in every case when the fraction of used columns is decreased to 1/8. We observe three different behaviors of the error. In case of MSNBC and DUBLIN the number of columns seems to have no effect on the resulting clustering. In every case the error of the final clustering is smaller than the error of the initial clustering, but the error is independent of f. With the SUSHI data the error of the initial clustering increases marginally as f increases, whereas the error of the final clustering remains constant (after a small increase at $f = 2$). Finally, with MLENS both errors increase with f.

Using only a random subset of the columns is obviously a very simple way of reducing the dimensionality. Most more sophisticated methods, such as the ones we discuss in Chapter 5, come with a higher computational cost. *Random projections* can be fast to compute, especially if the projection matrix is sparse. The approach we use here can be seen as a random projection that simply discards a number of the original dimensions. For a more elaborate, yet computationally efficient technique see for example [Ach03].

4.5 Conclusion

We have discussed the problem of clustering chains. We considered two approaches and their combination. First we gave simple definitions of a distance and a centroid that can be used together with Lloyd's algorithm for computing a clustering directly using the input D. In Section 4.3 we gave two methods for mapping chains to a high-dimensional vector space. These representations can be used with any clustering algorithm. The output of such an algorithm can still be further refined using the technique of Section 4.2.

Mapping chains to vector spaces is an interesting subject in its own right and can have many other uses in addition to clustering. In the next chapter we will use the mappings to create two dimensional scatterplots of sets of chains. They have also some theoretical interest, as the first mapping we discuss is related to the so called Planted Partition Model [CK01, ST02], for which some clustering algorithms have been developed. We gave some theoretical arguments why these algorithms might fail when the input is based on sets of chains. However, studying experimentally whether this really is the case may also be of interest.

Chapter 5

Visualizing sets of chains

5.1 Introduction

The human visual system is very efficient in finding patterns and structure from appropriately constructed images. Visual problems that are very tricky to solve using a computer are sometimes remarkably easy for humans. For example, the idea of a *captcha* [vBHL03] is based on the fact that it is hard to algorithmically recognize digits and letters that have been distorted in a certain way, while humans can still (fairly) easily see what the displayed characters are. Captcha's are used by many online services to prevent automated robots to, e.g., create user accounts or perform other possibly malicious tasks.

Also, what is understood by visualization in the first place depends on the task at hand. A graph with the price of a stock over some period of time is an obvious example, but one can also consider the fuel level indicator in a car as a visualization of the amount of gasoline in the tank. We use the word visualization to mean *a single static figure that represents a finite set of data.* The stock price example fits this definition, while the fuel level indicator doesn't.

Visualization can play an important part in explorative data analysis. For example, given a set of points in a vector space, a human can easily see from a suitable visualization if the points from clusters and how many clusters there are in total. These questions are rather nontrivial to answer even using elaborate algorithmic techniques.

In the previous chapter we looked at the problem of clustering a set of chains. To this end we proposed in Section 4.3 two techniques for representing chains in a vector space. In this chapter we make use of these representations to create visualizations of sets of chains. Given a set of chains, the task is to create a two dimensional scatterplot where each chain is represented by a single point, and points corresponding to similar rankings are plotted close to each other.

C_1	C_2
8 9 10 5 12 6	5 10 4 12 1 11
9 10 5 3 11 6	5 2 3 4 12 11
8 9 3 5 11 6	6 3 8 4 11 7
9 8 5 11 12 2	5 2 4 8 1 9
7 9 10 5 3 6	2 12 8 7 9 1
7 1 5 2 6 4	2 5 4 3 7 1
1 9 3 5 2 12	5 2 6 4 12 1
9 5 2 11 12 4	2 6 10 4 9 1
7 8 3 12 2 4	5 10 3 9 7 11
9 8 3 5 2 4	5 6 10 12 9 7
9 10 3 12 2 11	5 3 8 9 11 1
7 10 2 12 11 6	5 10 3 4 1 11
7 6 12 2 11 4	2 6 3 8 9 1
1 8 11 12 6 4	2 6 8 12 7 11
7 9 8 5 6 4	5 10 12 7 1 11

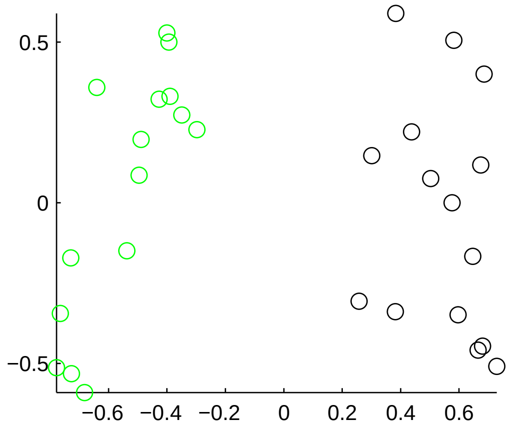


Figure 5.1: Example visualization of the chains shown on the left. The rankings are generated by a model with two components (C_1 and C_2). In the visualization rankings originating from the same component are indicated by the same color. The two components appear as two clouds of points in the scatterplot. Figure from [Ukk07].

In short our approach can be summarized as follows: map all chains in D to a high-dimensional space with one of the mappings of the previous chapter, and then apply some known dimension reduction algorithm to create a set of two dimensional points, one point for each chain. These points are plotted on a plane to construct the final visualization. See Figure 5.1 for an example. Here the set of items is the set of integers $1, 2, \ldots, 12$, and each chain contains 6 items. The chains in column C_1 tend to have a 7 or a 9 in the first position and a 6 or a 4 in the last. On the other hand, the chains in column C_2 have typically a 2 or a 5 in the first position and mostly end with a 1 or an 11. Clearly the chains can be divided to two clusters. The scatterplot in Figure 5.1 shows two well separated groups of circles, which correspond to the chains in columns C_1 and C_2. The data in Figure 5.1 was generated using the model discussed in Section 2.1.

5.2 Dimension reduction techniques

Let $X = \{\mathbf{x_1}, \ldots, \mathbf{x_n}\}$ be a set of vectors in $\mathbb{R}^m$. In dimension (or dimensionality) reduction the task is to construct a mapping $g : \mathbb{R}^m \to \mathbb{R}^k$, where $k \ll m$, that tries to minimize some distortion measure $\delta(X, g(X))$ between X and $g(X)$. We abuse notation slightly and denote by $g(X)$ the set of k-dimensional vectors obtained when g is applied to every vector $\mathbf{x} \in X$. Some typical choices for δ are

$$\delta(X, g(X)) = \sum_{i=1}^{n} \sum_{j=1}^{n} (d_1(\mathbf{x_i}, \mathbf{x_j}) - d_2(g(\mathbf{x_i}), g(\mathbf{x_j})))^2, \tag{5.1}$$

or

$$\delta(X, g(X)) = Var[X] - Var[g(X)], \tag{5.2}$$

where $Var[X]$ is defined as the trace of the covariance matrix of X. Using Equation 5.1 we are basically trying to find a mapping that preserves the interpoint distances as well as possible. Here d_1 and d_2 are some distance measures in $\mathbb{R}^m$ and $\mathbb{R}^k$, respectively. Multidimensional scaling (MDS) is an example of a dimension reduction technique that minimizes (5.1). In case of Equation 5.2 the mapping tries to preserve as much of the variance in X as possible. In dimension reduction based on *Principal Component Analysis* (PCA) we construct a mapping that minimizes Equation 5.2.

Another way of classifying dimension reduction algorithms is to consider if the mapping g is *linear* or *nonlinear.* A linear mapping can be written as

$$g(\mathbf{x}) = W\mathbf{x}, \tag{5.3}$$

where W is an $k \times n$ matrix. PCA is an example of a linear mapping, whereas MDS uses a nonlinear mapping. Below we discuss PCA, MDS and some other dimension reduction techniques in more detail.

Principal Component Analysis

Principal Component Analysis (PCA) (see for example [Sha95]) is a traditional method in statistics to reduce the number of variables in a data set. The task is to find a (preferably small) set of new variables that are linear combinations of the original variables and form an orthogonal basis, so that when the data is projected onto the new variables as much as possible of the variance is preserved as stated in Equation 5.2. In practice this is done by first computing the covariance matrix of the data and using the eigenvectors of this as the basis. Thus, the mapping g is formed by picking the k eigenvectors that correspond to the k largest eigenvalues of the covariance matrix and using them as the rows of the matrix W in Equation 5.3. The amount of variance preserved by the projection can be computed by summing the eigenvalues that correspond to the chosen eigenvectors and comparing this to the total variance $Var[X]$ given by the trace of the original covariance matrix.

Multidimensional Scaling and some variants
Multidimensional scaling (MDS) [BG97] is another traditional technique for creating low dimensional representations of data sets. In its simplest form MDS finds a mapping g such that the distortion defined by Equation 5.1 is minimized. The framework can be easily extended by modifying the cost function. Curvilinear Component Analysis (CCA) [DH97] is a method very similar to MDS. It introduces an additional component in the objective function that can be used to adjust how distances of different magnitudes affect the final outcome. The function minimized is of the form

$$\sum_i \sum_j \big(d_1(\mathbf{x_i}, \mathbf{x_j}) - d_2(g(\mathbf{x_i}), g(\mathbf{x_j}))\big)^2 F(d_2(g(\mathbf{x_i}), g(\mathbf{x_j})), \lambda_y), \tag{5.4}$$

where F is a bounded and monotonically decreasing (in $d_2(g(\mathbf{x_i}), g(\mathbf{x_j}))$) function. The idea is to reduce the effect of long distances in the output space to enhance the conservation of local neighborhoods. The parameter λ_y can be constant or varied over time.

Continuing the work of [DH97], the authors of [VK06] recently proposed a parameterized version of Equation (5.1). This is

$$\sum_i \sum_j \big(d_1(\mathbf{x_i}, \mathbf{x_j}) - d_2(g(\mathbf{x_i}), g(\mathbf{x_j}))\big)^2 \big((1-\lambda) F(d_2(g(\mathbf{x_i}), g(\mathbf{x_j})), \sigma_i) + \lambda F(d_1(\mathbf{x_i}, \mathbf{x_j}), \sigma_i)\big), \tag{5.5}$$

where F is again bounded and monotonically decreasing. The parameter λ can be used to control whether the neighborhood of a point in the projection or input space is considered more important. The size of the neighborhood is adjusted by the parameter σ_i, which is typically slowly decreased during the optimization. The method that optimizes Equation (5.5) for a given value of λ is called Local MDS [VK06]. Note that when $\lambda = 0$, equations 5.5 and 5.4 are the same.

Isomap
Isomap [TdL00] employs a technique called *manifold embedding.* Where PCA and other linear projections assume that the data in fact resides on a hyperplane in the high dimensional space, Isomap assumes that this low dimensional subspace is not a (hyper)plane, but a manifold with an arbitrary structure. An easy example is to consider a set points on a piece of paper that resides in a three dimensional world. When the paper lies flat on a table, the points occupy a planar two dimensional subspace of the three dimensional space. If we take the piece of paper and wrap it to a roll, the points still occupy a two dimensional subspace, but it is no longer a plane. The idea of a manifold embedding is to discover how the paper has been rolled and then "unfold" it.

Isomap does this by first constructing the k-nearest neighbors graph, and then optimizing a function similar to Equation (5.1) where $d_1(\mathbf{x_i}, \mathbf{x_j})$ is determined by the graph distance (shortest path) between data points $\mathbf{x_i}$ and $\mathbf{x_j}$ in the k nearest neighbors graph.

A number of algorithms for dimension reduction exist in the literature ([BN02, HR02, Koh82, QY04, RS00, WSZS07] to mention a few), but most of them employ same techniques as the ones discussed here. For a very nice summary of different dimension reduction techniques the reader is referred to [Ven07].

5.3 Example visualizations

At this point we show some examples that are obtained by the algorithms described above. The data sets are the ones described in Section 2.1. We first computed a clustering with $k = 3$ using the algorithm described in Chapter 4. The clusters were initialized at random, that is, the vector representations where not used when computing the clustering. This clustering is indicated by the colors red, green and blue in the figures below. In this section we evaluate the scatterplots qualitatively. A quantitative approach is taken in the next section.

Figures 5.2 and 5.3 show plots created from the data sets SUSHI and MLENS. With these the clusters seem to be mostly overlapping. None of the methods preserve global structure, but the Local MDS based approaches show some preservation of local structure. Chains belonging to the same cluster form several small homogenous groups, that seem to be located almost randomly next to each other. Even though the clusters are not separated, this kind of visualizations may be useful if the task is to "classify" unlabeled examples based on the nearest neighbors of an unlabeled point in the visualization. In the LMDS 0.0 based scatterplots a large number of the closest neighbors of a point belong to the same cluster as the point itself. Note that the reason for overlapping clusters may also be a problem with the clustering itself. Both SUSHI and MLENS are data sets where the length of the chains is short when compared with the total number of items. With this kind of data sets Lloyd's algorithm as started from a random initialization may lead to suboptimal results as discussed in Chapter 4.

Figures 5.4 and 5.5 show the data sets MSNBC and DUBLIN. Here the clusters are still slightly overlapping, but nonetheless much better separated than with the two other data sets. Almost all methods seem to work equally well. In some cases, such as MSNBC with PCA (hypersphere representation), or DUBLIN with Isomap (hypersphere representation), the clusters are even separated reasonably well.

5.4 Evaluating visualizations

Factors that affect the quality of a visualization depend strongly on the intended application of the visualization. If a visualization is supposed to provide accurate

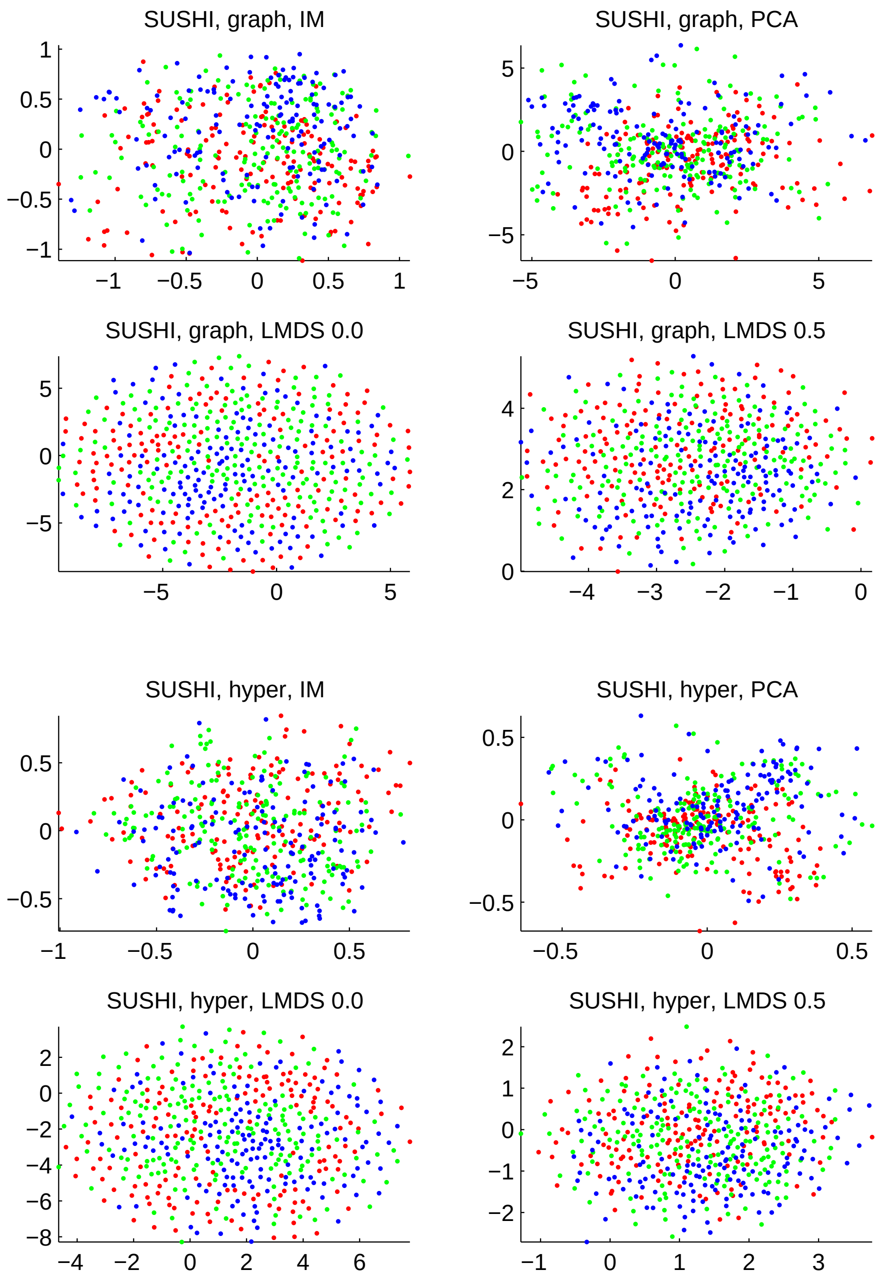


Figure 5.2: Scatterplots for the SUSHI data based on various algorithms.

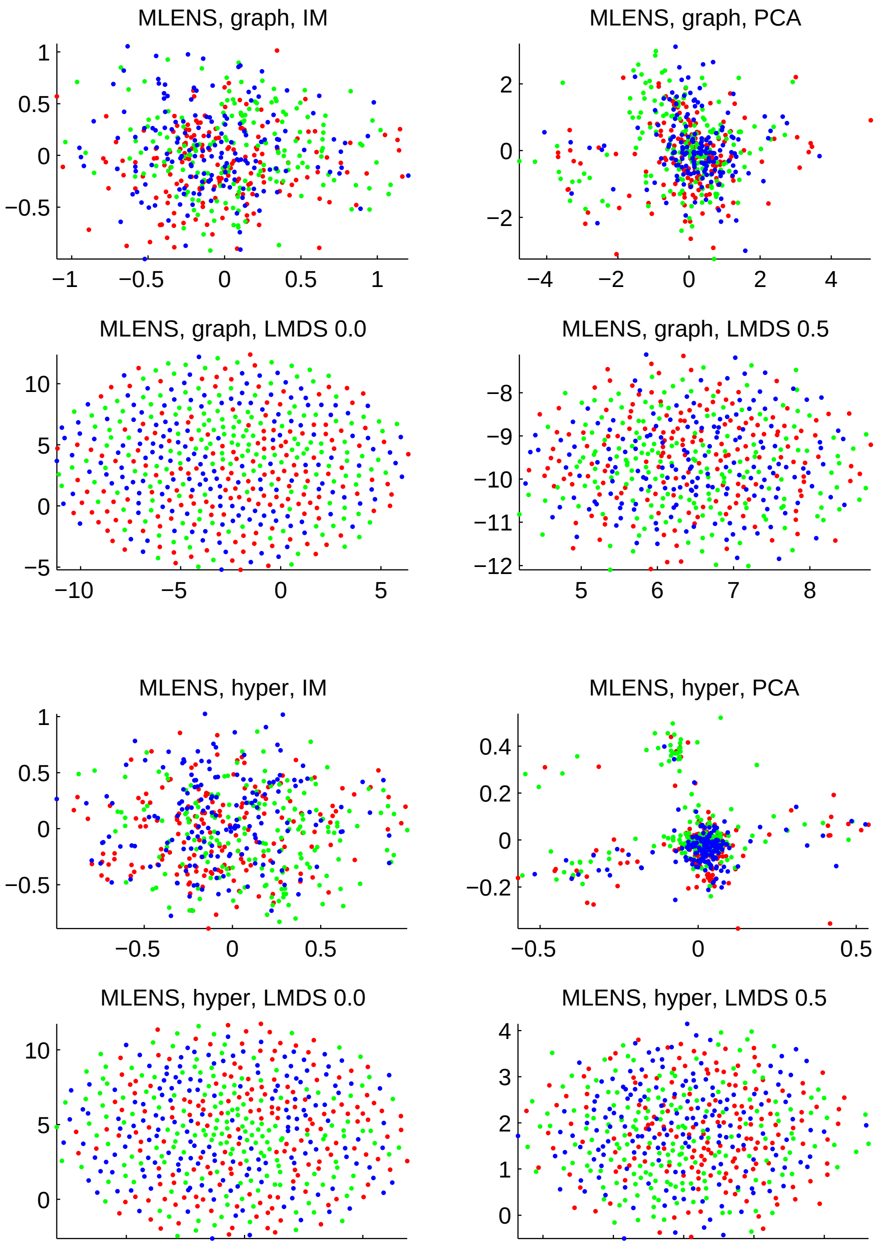


Figure 5.3: Scatterplots for the MLENS data based on various algorithms.

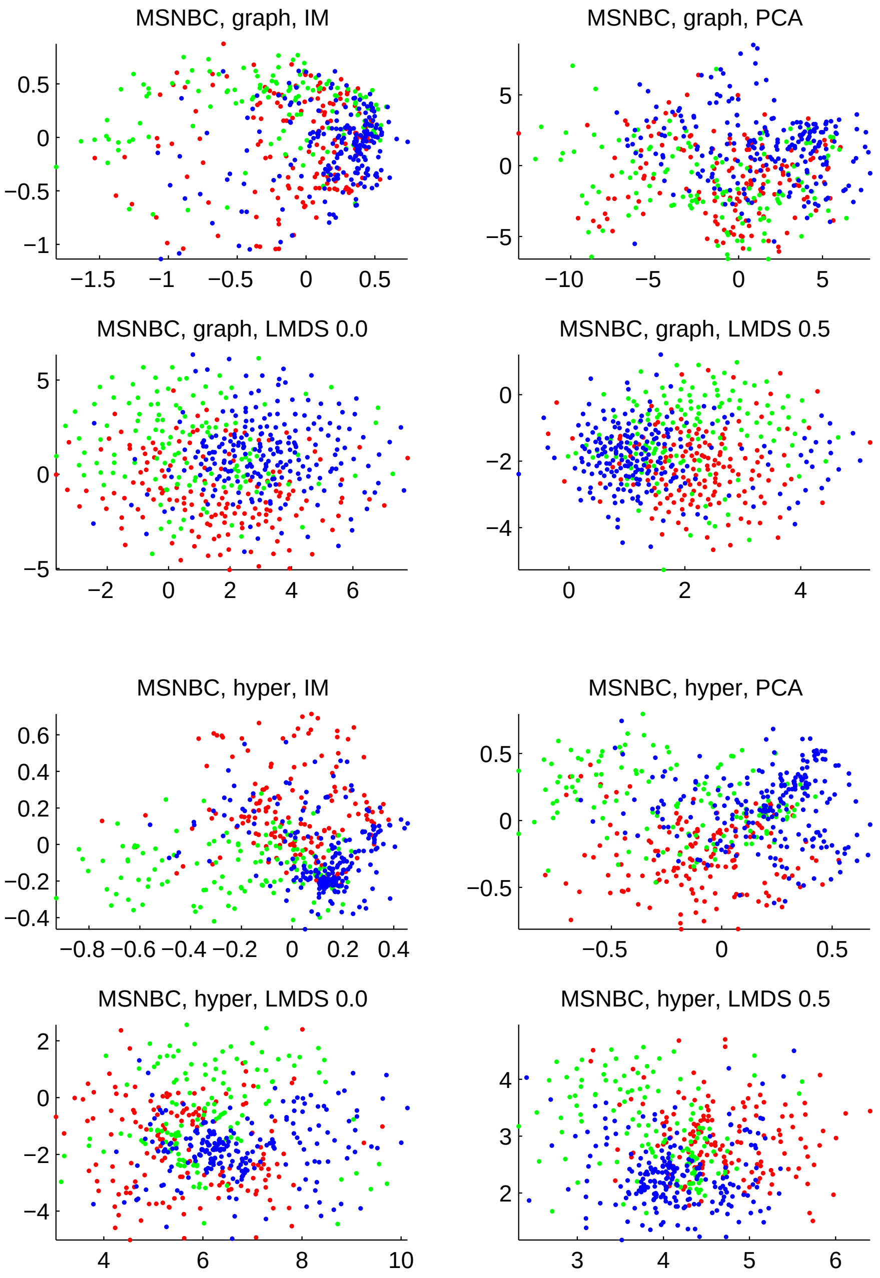


Figure 5.4: Scatterplots for the msnbc data based on various algorithms.

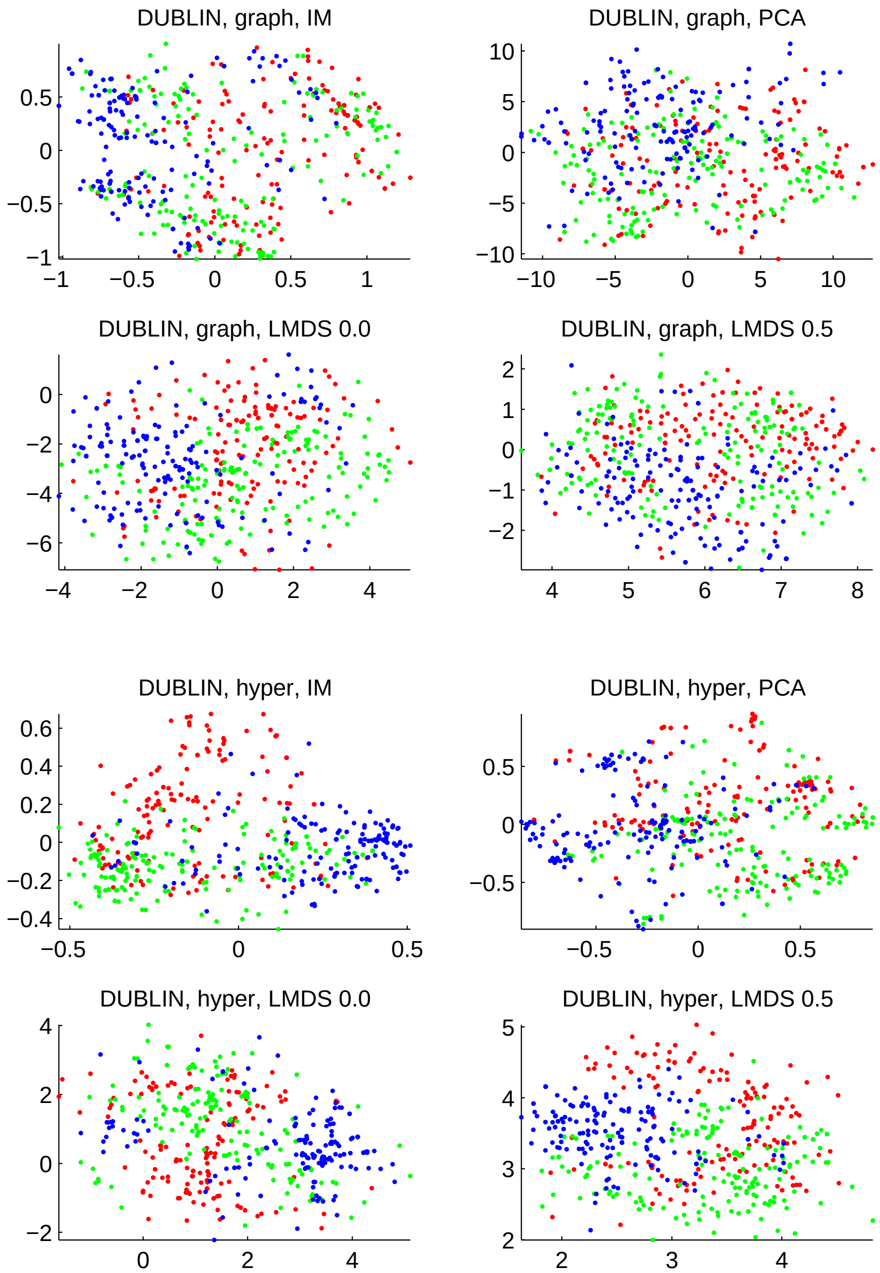


Figure 5.5: Scatterplots for the DUBLIN data based on various algorithms.

real-time information about a relatively simple phenomenon, such as the speed of a car, the evaluation criteria must be different than when assessing a visualization of a system that is more complex, such as the structure of a large software application. Naturally in both cases the information presented in the visualization must be accurate and truthful. However, in the latter case we can expect the user of the visualization to be prepared to aquire additional knowledge in order to properly understand the visualization. If understanding the speedometer of a car requires training, the visualization (the meter) is badly designed, but even well crafted UML-diagrams (Unified Modeling Language, a set of visual tools for representing e.g. the structure and operation of computer programs, see [ISO05]) require the user to have knowledge of UML.

One can look at the quality of a visualization from different perspectives. On one hand we can investigate how well a visualization performs in terms of qualities related to human perception and truthfulness with respect to the data. This is mostly related to the practical preparation of a figure: whether to use colors or not, what shapes to use, how to lay out the figure, etc., but also how well the contents match the actual data. There exist measures proposed by Tufte such as *lie-factor* and *data-ink ratio* [Tuf01] that can be used to assess visualizations from this point of view, but they should be considered more as general guidelines and thought experiments instead of rigorous quality measures. In practice evaluating the perceptual quality of a visualization is not trivial.

On the other hand we can evaluate how well a visualization fits the purpose it is intended for. Let us imagine we have two visualizations for the daily temperature outside during the last three months. The first one is a graph with time on the x-axis and temperature on the y-axis. The second one is a pie-chart, with one slice for every day, with the size of the slice being proportional to the actual temperature. Both figures can be prepared well in terms of Tufte's qualities mentioned above, but still most would consider the pie-chart a very bad visualization for the task of depicting trends in temperature.

We evaluate the visualizations more from the latter perspective. In exploratory data analysis the purpose of the visualization is to present the possibly large data set in a concise way. Most importantly the two (or in some cases three) dimensional visualization should be as similar as possible to the original high dimensional data. As we are visualizing a set of points in a high dimensional space with a set of points in a low dimensional space, a simple approach would be to use one of the distortion measures presented earlier. They are mostly rather difficult to interpret, however. For example, consider Equation 5.1. If this error is zero for some mapping g, we can be sure the resulting visualization has preserved all interpoint distances. This is very unlikely to happen. In a realistic setting we observe some error given by Equation 5.1, but this value might not have anything to do with the problem the user tries to solve using the visualization.

Visualization as Information Retrieval

Scatterplots can be used for identifying the number and shape of clusters, to detect outliers, and to study the neighborhood of a given point. We are interested in the last task. Suppose we are given a set of chains and create a scatterplot using the mapping g. Furthermore, suppose that we have additional information related to each chain. Given a new chain π without this information, we can use the plot to see to what "region" π is mapped by g. For this process to be reliable, the neighborhood of a point in the visualization must be similar to its neighborhood in the high dimensional space.

This can be seen as an information retrieval (IR) task, as done in [VK07] for example. Given a *query point*, we map it to the low dimensional space using g and observe its neighborhood. The set of *relevant points* is defined by the query point's neighborhood in the high dimensional space, while the set of *retrieved points* is defined by it's neighborhood in the visualization. Traditionally in IR one wants the set of retrieved documents (points, in this case) to contain every relevant document, and no irrelevant documents. In practice one can only retrieve a subset of the relevant documents, and the results contain also irrelevant documents.

We follow the approach of [VK07] and define the sets of relevant and retrieved points as follows: The set $H_r(\pi)$ is the set of r closest neighbors of π in the original high dimensional space. These are the relevant points. The set $P_k(\pi)$ is the set of k closest neighbors of π in the low dimensional projection. These are the retrieved points.

Precision and recall

In a good visualization $P_k(\pi)$ contains most of $H_r(\pi)$ and not much else. This can be measured with *precision* and *recall*. Precision is the fraction of relevant documents in the set of retrieved documents, while recall is the fraction of relevant documents returned. Obviously we could design a system that always returns every document in the collection and hence achieves the best possible score in terms of recall, but has a very low precision as the results contain mostly irrelevant documents. We define precision and recall as

$$\begin{aligned}\mathrm{prec}_k(\pi) &= \frac{|H_r(\pi) \cap P_k(\pi)|}{|P_k(\pi)|}, && (5.6)\\ \mathrm{recall}_k(\pi) &= \frac{|H_r(\pi) \cap P_k(\pi)|}{|H_r(\pi)|}. && (5.7)\end{aligned}$$

These definitions are for a single chain π. In practice when evaluating the visualizations we compute averages of both precision and recall over all chains and denote the measures prec_k and recall_k.

An established way of combining precision and recall is to use the F-measure. This is the harmonic mean of prec_k and recall_k, and it is defined as

$$F_\alpha = \frac{(1+\alpha)\cdot \text{prec}_k \cdot \text{recall}_k}{\alpha \cdot \text{prec}_\text{k} + \text{recall}_\text{k}}, \tag{5.8}$$

where α is a parameter that can be used to adjust the effect of precision and recall. Setting $\alpha = 1$ weights both evenly.

We can also study the dependency between precision and recall by using precision-recall -curves. They are constructed by computing prec_k and recall_k for all values of k and plotting a curve with recall on the x-axis and precision on the y-axis.

ROC curves

As alternatives to precision and recall we also use so called ROC (Receiver Operating Characteristic) curves (see e.g. pages 173–174 in [TK03]) for evaluating the visualizations. They show the *true positive rate* (TPR) at a given *false positive rate* (FPR). In case of dimension reduction these can be used like precision and recall to study how the neighborhoods are preserved by the projection. The true positive rate (false positive rate) tells us what fraction of the relevant (irrelevant) points is found in the set of retrieved points. The set of retrieved points corresponds to the neighborhood of the query point in the visualization. They are defined using the parameter k as follows. Remember that n is the total number of points, r is the number of relevant points and k is the number of retrieved points. We let

$$\text{TPR}_k(\pi) = \frac{|H_r(\pi) \cap P_k(\pi)|}{|H_r(\pi)|}, \tag{5.9}$$

$$\text{FPR}_k(\pi) = \frac{k - |H_r(\pi) \cap P_k(\pi)|}{n-r}. \tag{5.10}$$

Note that TPR_k is the same as recall_k, but FPR_k is not equivalent to precision. Just as with precision and recall there is a trade-off between these measures. Obviously we aim at a high true positive rate under a low false positive rate, but in general when we increase k to have a higher TPR we also increase the false positive rate. The ROC curve of a single point is obtained by computing $\text{TPR}_k(\pi)$ and $\text{FPR}_k(\pi)$ for all values of k. The curve of the entire visualization is the average of the ROC curves of every point.

ROC curves of two visualizations can be compared simply by inspecting the figures. Note that a “dummy” algorithm that simply places the points to two dimensions at random will have a ROC curve that is a straight line from the origin to $(1, 1)$. To obtain a single number for a visualization we can compute the so called AUC (Area Under Curve) statistic. This is simply the size of the area between the ROC curve and the x-axis. As both TPR and FPR range from 0 to 1, the maximum AUC is 1. This is only obtained when $\mathrm{TPR}_k = 1$ at a zero false positive rate. For the dummy algorithm AUC is obviously 0.5.

5.5 Experiments

In this section we describe empirical results with both artificial and real data sets. The artificial data was generated with the model described in Section 2.1. The real data sets are the ones described in Section 2.1. The algorithms we consider are Isomap (IM), Principal Component Analysis (PCA) and Local MDS (LMDS) with $\lambda = 0.0$ (which corresponds to CCA) and $\lambda = 0.5$. We test every algorithm with both the graph and hypersphere representations (see Section 4.3) giving eight different combinations in total. Implementations for Isomap[1] and LMDS[2] are obtained from the authors of [TdL00] and [VK06], respectively.

Results on artificial data

We generated four artificial data sets described in the table below:

	n	m	l	b	k
EASYL7	5000	100	7	100	2
EASYL10	5000	100	10	100	2
HARDL7	5000	100	7	2	2
HARDL10	5000	100	10	2	2

Thus, we say a data set is "easy" if it was generated by a permutation ($b = m$), and "hard", if it was generated by a bucket order with two buckets only. Also based on the clustering results we assume that data sets with $l = 7$ are in some sense harder to visualize than the ones with $l = 10$.

In Figure 5.6 are the precision-recall curves for the different artificial data sets. In general the results show that preserving the neighborhoods is not easy with any of the four data sets. The choice of the projection technique (graph vs. hyper) seems to have only a marginal effect. Overall the best performing algorithm is LMDS with $\lambda = 0.0$ when we are interested in having a high precision. If recall is more important the other algorithms seem to have a small advantage. Especially pronounced this difference is in the case of EASYL10 and the graph representation. In this case LMDS with $\lambda = 0.5$ has a recall over 0.75 with precision above 0.4.

We also use the F_1 measure for comparing the algorithms. We compute the F_1 measure for each of the eight combinations with $k = 10$, $k = 100$ and $k = 250$.

[1]`http://isomap.standord.edu/` (24.4.2008)

[2]`http://www.cis.hut.fi/projects/mi/software/dredviz/` (24.4.2008)

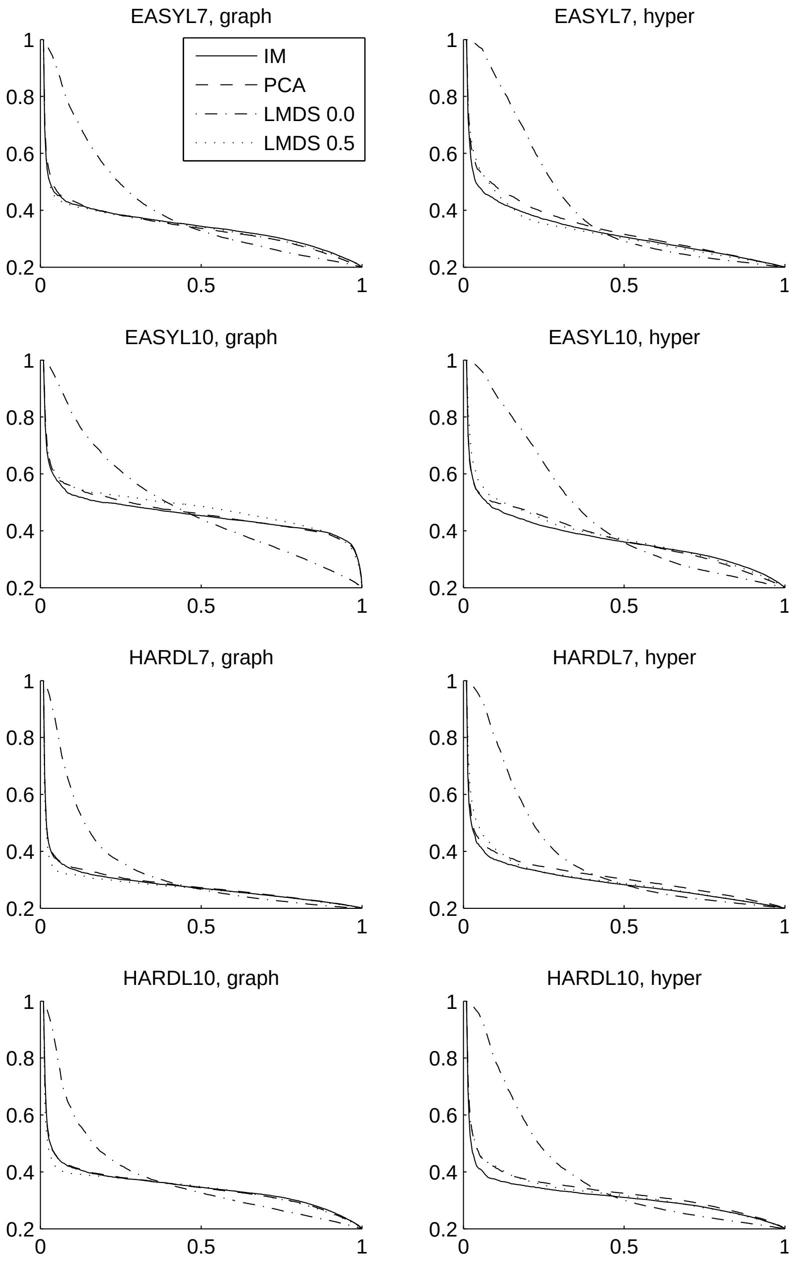


Figure 5.6: Precision-Recall curves for different artificial data sets and visualization techniques.

Table 5.1: Values of F_1 when $k = 10$ for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.086	0.105	0.071	0.084
PCA, hs	0.097	0.096	0.082	0.086
IM, g	0.084	0.104	0.070	0.084
IM, hs	0.088	0.096	0.078	0.076
LMDS00, g	0.145	0.154	0.130	0.127
LMDS00, hs	**0.163**	**0.164**	**0.153**	**0.153**
LMDS05, g	0.082	0.107	0.064	0.077
LMDS05, hs	0.098	0.102	0.089	0.085

With $k = 10$ we only look at a very small neighborhood around the query point, whereas with $k = 250$ the idea is to consider the "general region" where the query point is located. These results are given in tables 5.1, 5.2 and 5.3. In every table the best performing combination is indicated in bold for every data set.

In general when looking at the numbers we observe that the data sets with $l = 10$ are easier to visualize than the ones with $l = 7$. Also, when l is fixed the data sets labeled as EASY have higher values of F_1 than the ones labeled HARD. However, it seems that EASYL7 is in many cases at least as hard as HARDL10, indicating that the length of the chains has a stronger effect on the quality of the visualizations than the number buckets in the generating components.

When $k = 10$ LMDS with $\lambda = 0.0$ combined with the hypersphere representation clearly outperforms the other approaches. As k is increased to 100 the differences between the algorithms become smaller and it is not obvious that they are statistically significant. With $k = 250$ Local MDS is no longer the best performer, Isomap using the graph representation outperforms the other algorithms with three data sets. However, the difference to using the graph representation and LMDS with $\lambda = 0.5$ is in practice negligible. Interestingly PCA gives the best result with $k = 250$ in case of the HARDL7 data which we consider the most difficult case.

We now turn our attention to the ROC curves in Figure 5.7. Again we observe that the visualization task is not easy. Recall, that for a dummy algorithm the ROC curve would be a straight line from the origin to the upper right corner. Especially with the HARDL7 data the algorithms have only a marginally better performance than this. In general the curves indicate that the algorithms behave very much alike, with LMDS ($\lambda = 0.0$) being an exception. In every case it shows a slightly better true positive rate at very small false positive rates than the other algorithms. If higher false positive rates are allowed, LMDS 0.0 has consistently a lower true positive rate. When we look at the AUC statistic in

Table 5.2: Values of F_1 when $k = 100$ for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.360	0.464	0.299	0.365
PCA, hs	0.355	0.396	0.330	0.347
IM, g	0.364	0.459	0.296	0.364
IM, hs	0.342	0.384	0.313	0.330
LMDS00, g	**0.382**	0.462	0.321	0.366
LMDS00, hs	0.371	0.418	**0.349**	**0.369**
LMDS05, g	0.361	**0.488**	0.290	0.363
LMDS05, hs	0.335	0.393	0.315	0.337

Table 5.3: Values of F_1 when $k = 250$ for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.422	0.531	0.366	0.433
PCA, hs	0.394	0.432	**0.390**	0.415
IM, g	**0.429**	**0.535**	0.363	**0.438**
IM, hs	0.388	0.441	0.374	0.405
LMDS00, g	0.391	0.451	0.350	0.397
LMDS00, hs	0.364	0.395	0.357	0.372
LMDS05, g	0.422	0.532	0.363	0.431
LMDS05, hs	0.382	0.436	0.376	0.403

Table 5.4 we see that LMDS 0.0 has typically a significantly smaller AUC than the other algorithms. This, however, does not mean that we should always prefer for instance Isomap to Local MDS ($\lambda = 0.0$). In some applications we might only consider the few points that are closest to the query point, and need to have a high true positive rate in this setting.

Tables 5.5, 5.6, and 5.7 show the true positive rate at three different levels of FPR. When we fix FPR = 0.05 (Table 5.7) Local MDS (with $\lambda = 0.0$) has a considerably higher performance than any of the other algorithms. Also, in every case the hypersphere representation leads to a better preservation of the neighborhoods. For FPR = 0.1, LMDS 0.0 still has a higher true positive rate, but now the graph representation leads to better results in most cases. For a false positive rate of 0.2 the differences between the best performing algorithms are very small, and most likely not statistically significant. The easyl10 data seems to be an exception in this respect.

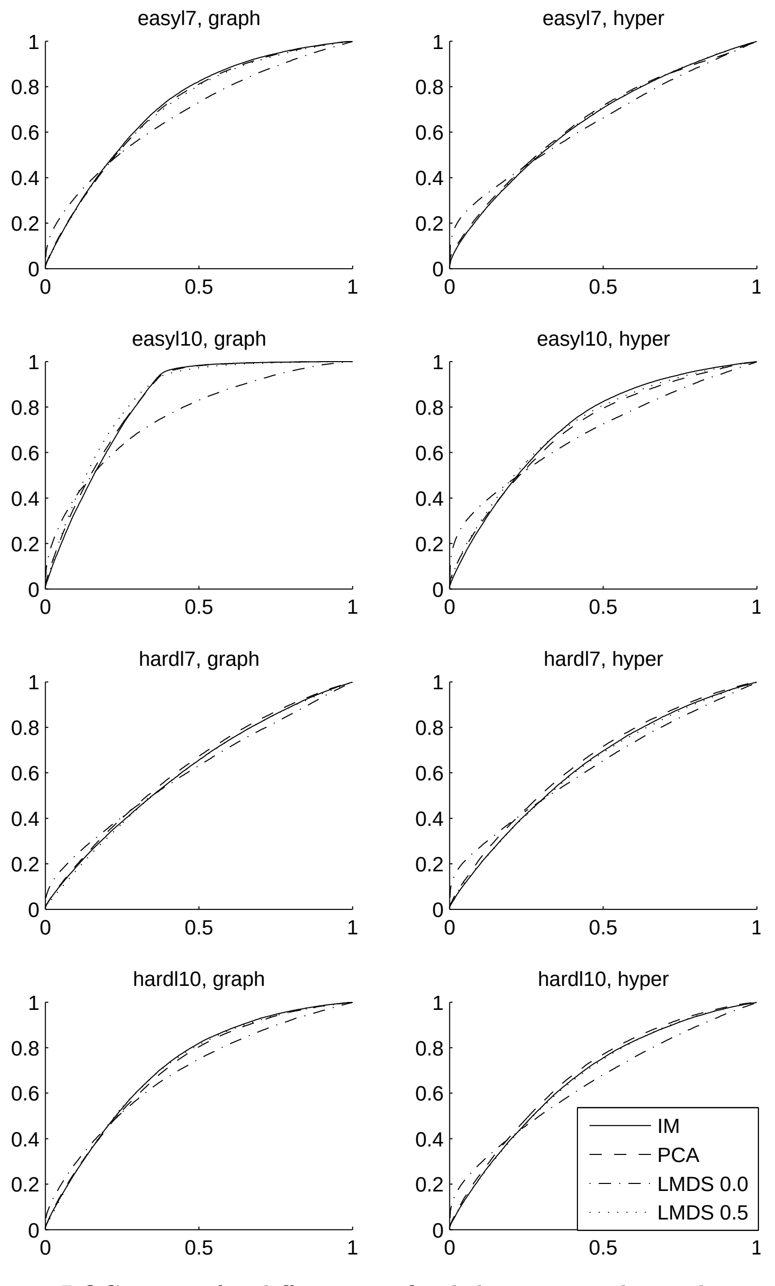


Figure 5.7: ROC curves for different artificial data sets and visualization techniques.

Table 5.4: AUC statistic for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.713	0.827	0.621	0.707
PCA, hs	0.652	0.706	**0.651**	0.686
IM, g	**0.718**	0.822	0.609	**0.714**
IM, hs	0.647	0.719	0.634	0.673
LMDS00, g	0.682	0.759	0.606	0.686
LMDS00, hs	0.641	0.688	0.628	0.647
LMDS05, g	0.708	**0.835**	0.605	0.712
LMDS05, hs	0.650	0.718	0.630	0.671

Table 5.5: True positive rate (at false positive rate of 0.2) for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.451	0.621	0.337	0.442
PCA, hs	0.391	0.457	0.378	0.414
IM, g	**0.458**	0.604	0.325	0.449
IM, hs	0.381	0.463	0.351	0.398
LMDS00, g	0.455	0.570	0.352	0.450
LMDS00, hs	0.407	0.475	**0.383**	0.408
LMDS05, g	0.449	**0.671**	0.312	**0.453**
LMDS05, hs	0.387	0.482	0.349	0.393

Table 5.6: True positive rate (at false positive rate of 0.1) for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.264	0.374	0.192	0.260
PCA, hs	0.245	0.285	0.224	0.243
IM, g	0.262	0.346	0.185	0.255
IM, hs	0.234	0.271	0.200	0.230
LMDS00, g	**0.318**	**0.407**	0.239	**0.295**
LMDS00, hs	0.308	0.365	**0.274**	0.291
LMDS05, g	0.258	0.402	0.170	0.256
LMDS05, hs	0.232	0.296	0.202	0.228

Table 5.7: True positive rate (at false positive rate of 0.05) for different artificial data sets and visualization techniques.

	EASYL7	EASYL10	HARDL7	HARDL10
PCA, g	0.153	0.218	0.110	0.149
PCA, hs	0.156	0.177	0.129	0.144
IM, g	0.147	0.193	0.106	0.144
IM, hs	0.146	0.155	0.114	0.131
LMDS00, g	0.228	0.291	0.169	0.195
LMDS00, hs	**0.245**	**0.293**	**0.212**	**0.218**
LMDS05, g	0.146	0.227	0.090	0.138
LMDS05, hs	0.150	0.185	0.120	0.134

Overall the results on artificial data indicate that visualizing sets of chains in this way is not easy, at least not in terms of the chosen performance measures. There is always a clear difference between the easiest (EASYL10) and hardest (HARDL10) data sets, but this difference is not as big as one might expect. Especially surprising is the relatively poor performance of all algorithms even with the EASYL10 data. In practice we would expect all real data sets to have less structure than EASYL10, and hence to be more difficult to visualize.

Results on real data

We conducted the same set of experiments also with the data sets SUSHI, MLENS, MSNBC and DUBLIN described in Section 2.1. Again it is not clear what algorithm is the best and what the worst; the results seem to have a strong dependence on the data set in question.

In Figure 5.8 are the precision-recall curves. First we note that MLENS is difficult for all algorithms. The SUSHI data is slightly easier, but leads to worse performance than DUBLIN or MSNBC. We believe this is largely a consequence of the fact that the size of M is considerably smaller in DUBLIN and MSNBC than SUSHI and MLENS.

LMDS ($\lambda = 0.0$) yields the best results with all data sets and both representations if we prefer precision. If recall is more important Isomap and PCA seem to have an advantage, although this only occurrs in case of DUBLIN and to some extent with MSNBC. Also, with these two data sets the difference between setting $\lambda = 0.0$ and $\lambda = 0.5$ with LMDS leads to a considerably smaller difference than in case of SUSHI and MLENS.

Tables 5.8, 5.9, and 5.10 show the F_1 measure for $k = 10$, $k = 100$, and $k = 250$. When $k = 10$ (Table 5.8) LMDS 0.0 outperforms the other approaches by a clear margin. Also, the difference between using the graph and hypersphere

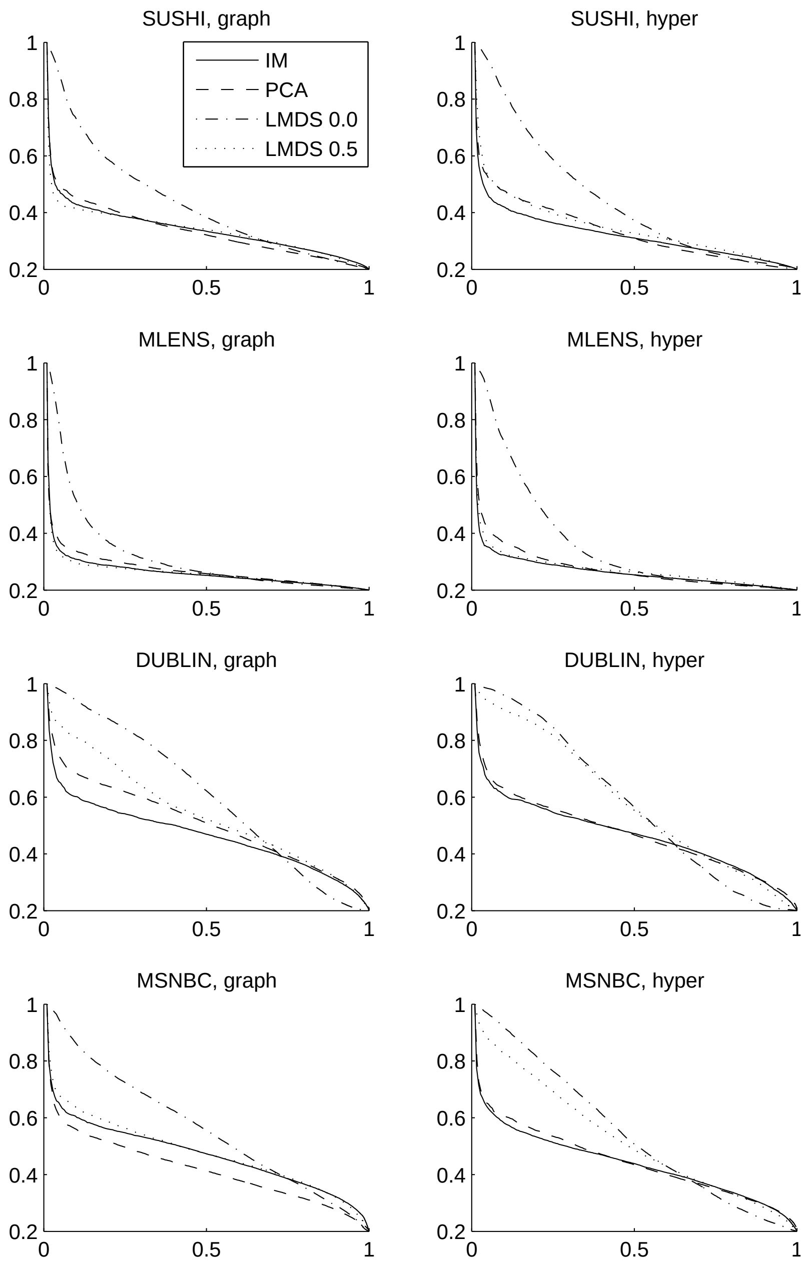


Figure 5.8: Precision-Recall curves for different real data sets and visualization techniques.

Table 5.8: Values of F_1 when $k = 10$ for different data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.090	0.071	0.128	0.107
PCA, hs	0.095	0.078	0.119	0.115
IM, g	0.087	0.066	0.115	0.114
IM, hs	0.086	0.066	0.116	0.113
LMDS00, g	0.141	0.119	0.172	0.160
LMDS00, hs	**0.155**	**0.143**	**0.175**	**0.168**
LMDS05, g	0.080	0.064	0.149	0.121
LMDS05, hs	0.096	0.071	0.166	0.153

Table 5.9: Values of F_1 when $k = 100$ for different data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.358	0.288	0.506	0.434
PCA, hs	0.363	0.290	0.477	0.452
IM, g	0.362	0.276	0.477	0.480
IM, hs	0.342	0.283	0.478	0.453
LMDS00, g	**0.426**	0.310	**0.562**	**0.532**
LMDS00, hs	0.425	**0.338**	0.531	0.504
LMDS05, g	0.363	0.273	0.516	0.480
LMDS05, hs	0.360	0.287	0.528	0.492

representations is more pronounced for SUSHI and MLENS when this algorithm is used. In general the hypersphere representation leads to better results for $k = 10$. When k is increased to 250 (Table 5.10) the situation is reversed, and the graph representation gives almost always a better score. Finally, when we compare $k = 100$ (Table 5.9) and $k = 250$, we observe that for "medium-sized" neighborhoods ($k = 100$) Local MDS tends to work better when $\lambda = 0.0$, whereas for very large neighborhoods ($k = 250$) $\lambda = 0.5$ should be preferred.

The ROC curves in Figure 5.9 indicate a similar behavior. All curves for the MLENS data are almost straight, indicating that the projection is almost the same that one would obtain with the dummy algorithm. Just as with artificial data LMDS 0.0 has a slightly higher true positive rate at the very small false positive rates, and a slightly lower true positive rate at the higher false positive rates. When we look at the AUC statistics in Table 5.11 the differences between the best performing algorithms are again small. Also the difference between the best and worst algorithm is in most cases not very big. For the more difficult data

Table 5.10: Values of F_1 when $k = 250$ for different data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.393	0.352	0.486	0.454
PCA, hs	0.378	0.341	0.478	0.467
IM, g	0.413	0.346	0.483	**0.490**
IM, hs	0.392	0.347	0.482	0.470
LMDS00, g	0.409	0.347	0.457	0.477
LMDS00, hs	0.392	0.349	0.434	0.442
LMDS05, g	**0.416**	0.350	**0.490**	**0.490**
LMDS05, hs	0.405	**0.358**	0.473	0.466

sets (sushi and mlens) LMDS 0.0 also has a higher AUC. For these data sets it seems that LMDS 0.0 is consistently better than the others.

When looking at Tables 5.12, 5.13, and 5.14 that show the true positive rate at a false positive rate of 0.2, 0.1, and 0.05, respectively, the results are remarkably consistent. In every case LMDS 0.0 leads to the best results. Moreover, the hypersphere representation outperforms the graph representation usually by a clear margin, with the data set dublin being an exception as there the situation is reversed.

5.6 Conclusion

We investigated how the vector representation techniques of Section 4.3 can be used for creating two dimensional scatterplots of sets of chains when combined with a dimension reduction algorithm. The algorithms we employed were PCA, Isomap, and Local MDS with two different parameter settings. We used precision and recall, together with ROC curves as a quantitative performance measure for evaluating the obtained scatterplots. Effectively we measured how well the neighborhoods of individual chains are preserved by the projection to two dimensions.

The main result is that the proposed techniques are suitable for the purpose of visualization, but the quality of the resulting scatterplots strongly depends on the characteristics of the input. Especially crucial is the total number of items and the length of the chains. Shorter chains are harder to visualize while having a smaller number of items in total leads to better performance, at least in terms of the chosen measures.

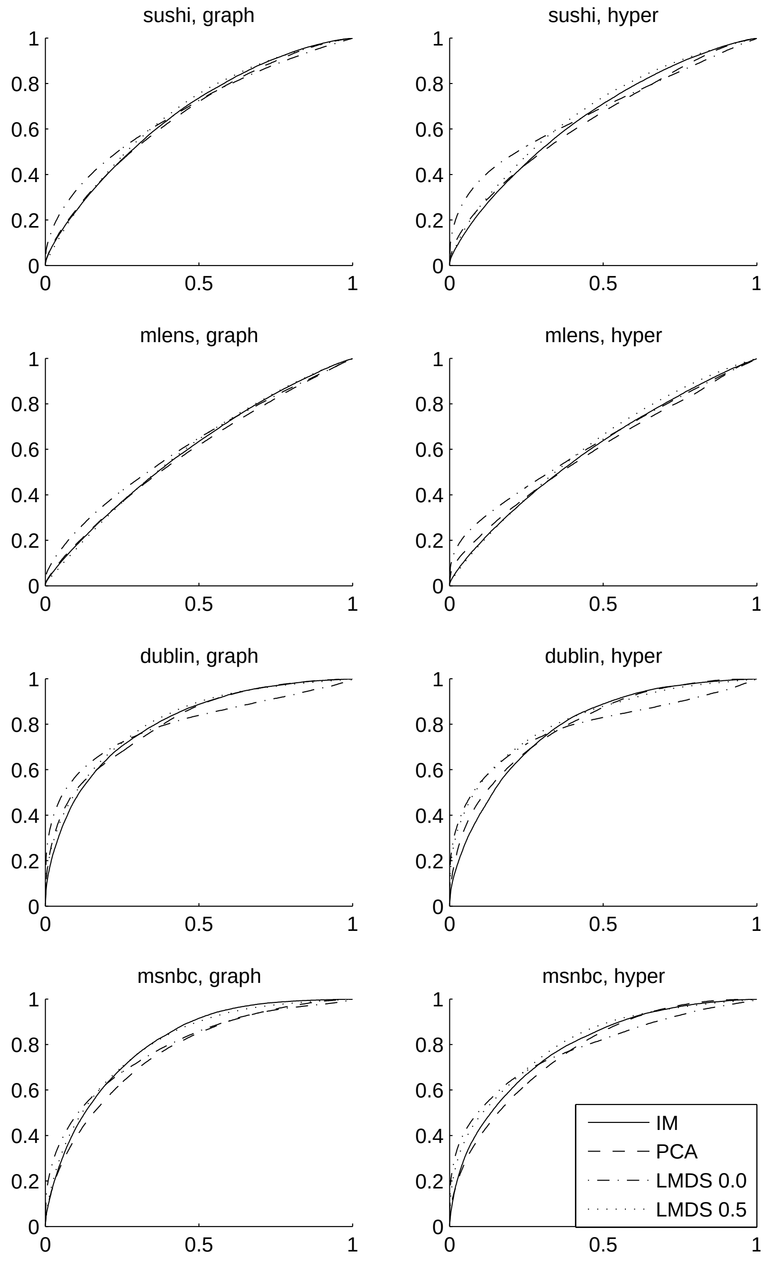


Figure 5.9: ROC curves for different real data sets and visualization techniques.

Table 5.11: AUC statistic for different real data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.662	0.587	0.804	0.765
PCA, hs	0.640	0.595	0.796	0.771
IM, g	0.668	0.597	0.803	**0.807**
IM, hs	0.653	0.599	0.790	0.786
LMDS00, g	**0.681**	0.616	0.797	0.789
LMDS00, hs	0.674	**0.624**	0.784	0.782
LMDS05, g	0.675	0.597	0.814	0.805
LMDS05, hs	0.672	0.612	**0.815**	0.804

Table 5.12: True positive rate (at false positive rate of 0.2) for different real data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.399	0.313	0.634	0.564
PCA, hs	0.390	0.339	0.621	0.563
IM, g	0.398	0.309	0.645	0.629
IM, hs	0.385	0.323	0.606	0.602
LMDS00, g	0.462	0.365	**0.683**	0.628
LMDS00, hs	**0.483**	**0.388**	0.669	**0.641**
LMDS05, g	0.405	0.301	0.663	0.639
LMDS05, hs	0.415	0.321	0.676	0.630

Table 5.13: True positive rate (at false positive rate of 0.1) for different real data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.245	0.183	0.508	0.393
PCA, hs	0.258	0.219	0.469	0.399
IM, g	0.239	0.177	0.475	0.434
IM, hs	0.236	0.188	0.408	0.434
LMDS00, g	0.329	0.237	**0.572**	0.488
LMDS00, hs	**0.375**	**0.287**	0.545	**0.513**
LMDS05, g	0.236	0.161	0.502	0.456
LMDS05, hs	0.261	0.184	0.535	0.487

Table 5.14: True positive rate (at false positive rate of 0.05) for different real data sets and visualization techniques.

	SUSHI	MLENS	DUBLIN	MSNBC
PCA, g	0.153	0.109	0.391	0.264
PCA, hs	0.172	0.151	0.343	0.280
IM, g	0.145	0.103	0.328	0.280
IM, hs	0.143	0.112	0.272	0.305
LMDS00, g	0.233	0.158	**0.475**	0.374
LMDS00, hs	**0.289**	**0.221**	0.439	**0.417**
LMDS05, g	0.131	0.086	0.376	0.307
LMDS05, hs	0.161	0.105	0.418	0.377

Chapter 6

Sampling sets of chains

6.1 Introduction

So far we have looked at techniques for analyzing sets of chains. In this chapter we address the question of the validity of the found results. Our work falls in the field of *randomization testing*, which is a useful approach for evaluating the results of data-analysis algorithms [GMMT07, HMT07, UM07, OVK$^+$08]. In short, the idea is to compare the results obtained from real data with those that are obtained (using the same algorithm) from random data. It is important that the random data share some well defined properties with the real data. Having considerably different results for real data than random data suggests that the phenomenon we observe in real data is statistically significant.

Unlike in *Bootstrap* methods, we are not only sampling the original data with replacement, but have to generate a truly different data set that satisfies a number of given properties. This can be rather trivial, if we can assume the data follows some known distribution. In this case the process of randomizing a new data set is a matter of estimating the parameters of the distribution, and then picking a sample from the distribution given the estimated parameters. However, if no assumptions of the distribution can be made, we have to employ some other techniques.

To do this we must define the properties of the data we want to preserve by some other means. For example, consider market basket data represented as a 0–1 matrix. Each row represents a shopping basket, while each column represents a product. If an entry of the matrix is 1, the shopping basket represented by the row contains the product represented by the column. This type of data is typically used when mining frequent itemsets and association rules [AIS93, MTV94]. In this case it might be interesting to preserve the sizes of the shopping baskets, and the "popularities" of individual products. More formally, we want to preserve the row and column frequencies (sums) of the 0–1 matrix. Note that this approach does not preserve itemsets. To do randomization testing, we must thus pick uniformly at random a number of matrices from the set of all 0–1 matrices that share the row and column sums with the real data matrix. A Markov Chain Monte Carlo algorithm for this was proposed in [CC03]. This approach was extended to real valued matrices in [OVK$^+$08], where the properties to maintain are the means and variances of every row and column.

Furthermore, we must also define the *test statistic* that we compare across the real and random data sets. This obviously depends on the model we are using. For example in [GMMT07] the authors used the number of frequent itemsets as a statistic when assessing the significance of the frequent itemsets, and the reconstruction error of k-means when testing clustering models.

In this work we consider the randomization of sets of chains. The properties we want to preserve are the number of chains, the lengths of all chains, occurrence frequencies of every itemset, and most importantly the unnormalized pair order matrix C_D. Given the set of chains D, the problem is to uniformly sample from the set of sets of partial rankings that share the above properties with D. We use the framework for testing the clusterings found using the algorithms in Chapter 4.

6.2 Randomization testing for chains

Empirical p-values

We start with some definitions. Let D be a set of chains as usual, and denote by $\mathcal{A}$ a data analysis algorithm that takes D as the input and produces some output, denoted $\mathcal{A}(D)$. We can assume that $\mathcal{A}(D)$ is in fact the value of the test statistic that we are interested in. Denote by $\tilde{D}_1, \ldots, \tilde{D}_h$ a sequence of random sets of chains that share certain properties with D. These will be defined more formally later.

If the value $\mathcal{A}(D)$ considerably deviates from the values $\mathcal{A}(\tilde{D}_1), \ldots, \mathcal{A}(\tilde{D}_h)$, we have some evidence for the results obtained by $\mathcal{A}$ to be statistically significant. In practice this means we can rule out the common properties of the real and random data sets as the sole causes for the results found. As usual in statistical testing we can speak of a *null hypothesis* H_0 and an *alternative hypothesis* H_1. These are defined as follows:

$$\begin{aligned} H_0 : \mathcal{A}(D) &= E[\mathcal{A}(\tilde{D}_i)] \\ H_1 : \mathcal{A}(D) &\neq E[\mathcal{A}(\tilde{D}_i)], \end{aligned}$$

where $E[\mathcal{A}(\tilde{D}_i)]$ denotes the empirical mean estimated from the values $\mathcal{A}(\tilde{D}_1), \ldots,$ $\mathcal{A}(\tilde{D}_h)$.

In statistics the *p-value* of a test usually refers to the probability of making an error when rejecting H_0 (and accepting H_1). In order to determine the p-value one typically needs to make some assumptions of the distribution of the test statistic. In general, if we cannot, or do not want to make such assumptions, we can compute the *empirical p-value* based on the randomized data sets. This is defined simply as the fraction of cases where the value of $\mathcal{A}(\tilde{D}_i)$ is more extreme than the value $\mathcal{A}(D)$. Or more formally, for the one-tailed case where $\mathcal{A}(D)$ is expected to be small according to H_1, we have

$$\hat{p} = \frac{|\{\tilde{D}_i : \mathcal{A}(\tilde{D}_i) \leq \mathcal{A}(D)\}| + 1}{h + 1}. \tag{6.1}$$

The one-tailed case with $\mathcal{A}(D)$ being large is obtained simply by flipping the $\leq$ sign above. One problem with using $\hat{p}$ is that in order to get a reasonable estimate the number of randomized data sets must be fairly high. For instance, to have $\hat{p} = 0.001$ we must sample at least 999 data sets. Depending on the complexity of generating one random data set this may be difficult. Of course, already with 99 data sets we can obtain an empirical p-value of 0.01 if all random data sets have a larger value of the test statistic. This should be enough for most practical applications.

Equivalence classes of sets of chains

The random data sets must share some characteristics with the original data D. Given D, we define an equivalence class of sets of partial rankings, so that all sets belonging to this equivalence class have the same properties as D.

Let D_1 and D_2 be two sets of chains on items of the set M. We say that D_1 and D_2 belong to the same equivalence class whenever

1. The number of chains of length l is the same in D_1 as in D_2 for all l.
2. Frequencies of all itemsets in D_1 and D_2 are the same.
3. The pair order matrices C_{D_1} and C_{D_2} are equivalent.

Here an itemset has the same meaning as in frequent pattern mining literature. An itemset I is a subset of M, and the frequency of I is the number of chains that contain I as a subset (the order of the items is not considered).

Given a set D of chains, we denote the equivalence class specified by D with $\mathcal{C}(D)$. To use the framework of randomization testing for testing some property of D, we must sample h sets of chains uniformly from $\mathcal{C}(D)$. In the next subsection we discuss an algorithm that does this. But first, let us elaborate why it is useful to maintain the properties listed above when testing the significance of $\mathcal{A}(D)$.

When we apply algorithm $\mathcal{A}$ on the set of chains D, we are essentially investigating the rankings, i.e., the result should be a consequence of the *ordering information* present in D. This is only one property of D, however. Others are those that we mention in the conditions. Condition 1 is used to rule out the possibility that the value of $\mathcal{A}(D)$ is somehow caused only by the length distribution of the chains in D. Note that this requirement also implies that D_1 and D_2 are of the same size. Likewise, condition 2 should rule out the possibility that the result is not a consequence of the rankings, but simply the co-occurrences of the items.

Maintaining the pair order matrix is motivated from a slightly different point of view. If D contained real-valued vectors instead of chains, it would make sense to maintain the empirical mean of the observations. When we require that C_D is preserved, we are making a similar requirement. Recall that the rank aggregation problem can be solved by formulating a minimum feedback arc set problem given the pair order matrix (see Chapter 2). Hence, the randomized data sets $\tilde{D}_i$ will have the same "mean" as D. One way of seeing this is to view D as a set of points in the space of chains. The random data sets should be located in the same region of this space as D.

An MCMC algorithm for sampling from $\mathcal{C}(D)$

Next we will discuss a Markov chain Monte Carlo algorithm that samples uniformly from $\mathcal{C}(D)$ given D. Under suitable conditions the samples obtained are independent, which is important if we want to use the empirical p-value for assessing the significance of $\mathcal{A}(D)$.

Overview of the algorithm

The MCMC algorithm we propose can be seen as a random walk on an undirected graph with $\mathcal{C}(D)$ as the set of vertices. Denote this graph by $G(D)$. The vertices D_1 and D_2 of $G(D)$ are connected by an edge if we obtain D_2 from D_1 by performing a small local change in D_1 (and vice versa). We call this local change a *swap* and will define it later in detail. First, let us look at a higher level description of the algorithm.

In general, when using MCMC to sample from a distribution, we must construct the Markov Chain so that its *stationary distribution* equals the target distribution we want to sample from. In case of random walks on undirected graphs it is known that if we pick the next vertex uniformly at random from the set of neighboring vertices of the current vertex, the chain will converge to a distribution where the probability of vertex D_i is proportional to its degree, i.e., number of neighbors.

If all vertices are of equal degree, the stationary distribution will be the uniform distribution. As we want to sample uniformly from $\mathcal{C}(D)$, this would be optimal. However, it turns out that the way we define the graph $G(D)$ does not result in the vertices having the same number of neighboring vertices. To remedy this, we use the *Metropolis-Hastings* algorithm (see e.g. [GCSR04]) for picking the next state. Denote by $N(D_i)$ the set of neighbors of the vertex D_i. When the chain is at D_i, we pick uniformly at random the vertex D_{i+1} from $N(D_i)$. The chain jumps to D_{i+1} with probability

$$\min(\frac{|N(D_i)|}{|N(D_{i+1})|}, 1), \tag{6.2}$$

that is, the jump is accepted always when D_{i+1} has a smaller degree, and otherwise we jump with a probability that decreases as the degree of D_{i+1} increases. If the chain does not jump, it stays at the state D_i and attempts to jump again (possibly to some other neighboring vertex) in the next step.

It is easy to show that this modified random walk has the desired property of converging to a uniform distribution over the set of vertices. Denote by $p(D_i)$ the *target distribution* we want to sample from. In this case $p(D_i)$ is the uniform distribution over all i. Hence, we have $p(D_i) = p(D_{i+1})$ for all i. The Metropolis-Hastings algorithm jumps to the next state D_{i+1} with probability $\min(r, 1)$, where

$$r = \frac{p(D_{i+1})/J(D_{i+1}|D_i)}{p(D_i)/J(D_i|D_{i+1})}. \tag{6.3}$$

Above $J(\cdot|\cdot)$ is a *proposal distribution*, which in this case is simply the uniform distribution over the neighbors of D_i for all i. That is, we have $J(D_{i+1}|D_i) = |N(D_i)|^{-1}$ and $J(D_i|D_{i+1}) = |N(D_{i+1})|^{-1}$. When this is substituted into Equation 6.3 along with the fact that $p(D_i) = p(D_{i+1})$ we obtain Equation 6.2.

Given D, a simple procedure for sampling one $\tilde{D}$ uniformly from $\mathcal{C}(D)$ works as follows: we start from $D = D_0$, make a series of local changes to the data resulting in a new slightly modified set of chains on every step, call these $D_1, D_2, \ldots$. After s steps we are at the set D_s, which we let to be our sample $\tilde{D}$. It is important to make enough transformations, i.e., have a large enough s, so that $\tilde{D}$ will not depend on the starting point D. We will discuss a method for determining the correct number steps later. If a larger number of samples is desired, we can simply continue running the Markov chain for another set of s transformations from each sample onwards until enough samples have been obtained.

Note that this procedure for sampling multiple $\tilde{D}$ is valid only when s is large enough to make the individual samples independent. Typically this will not be the case, or at least it is very complicated to determine if the samples are truly independent. However, if we require the samples only to be *exchangeable*, a slight modification to the above procedure will yield a method that does not suffer from the difficulties associated with the simple sequential approach.

We now describe this better approach that will let us sample h sets of chains from $\mathcal{C}_D$ so that the samples satisfy the exchangeability condition. The approach is originally proposed in [BC89]. We first start the Markov chain from D and run it backwards for s steps. In practice the way we define our Markov chain, running it backwards is equivalent to running it forwards. This gives us the set $\tilde{D}_0$. Next, we run the chain forwards $h-1$ times for s steps, each time starting from $\tilde{D}_0$. This way the samples can no longer be dependent on each other, but only on $\tilde{D}_0$. And since we obtained $\tilde{D}_0$ by running the Markov chain backwards from D, the dependence of the samples of $\tilde{D}_0$ is the same as the dependence of D on $\tilde{D}_0$. Note that a somewhat more efficient approach is proposed in [BC91].

The Swap

When running the chain we must make sure the data sets $D_1, D_2, \ldots D_s$ all belong to $\mathcal{C}(D)$. This is achieved when the local change is defined so that if D_i belongs to $\mathcal{C}(D)$, then D_{i+1} belongs to $\mathcal{C}(D)$ as well, independent of i. We call this local change a *swap* for reasons that will become apparent shortly.

Formally we define a swap as the tuple (π, τ, i, j), where π and τ are chains in D, i is an index of π, and j and index of τ. To do the swap (π, τ, i, j) we transpose the items at positions i and $i+1$ in π, and at positions j and $j+1$ in τ. For example, if $\pi = (1, 2, 3, 4, 5)$ and $\tau = (3, 2, 6, 4, 1)$, the swap $(\pi, \tau, 2, 1)$ will result in the chains $\pi' = (1, 3, 2, 4, 5)$ and $\tau' = (2, 3, 6, 4, 1)$. The positions of items 2 and 3 are changed in both π and τ.

Clearly this swap does not affect the number of chains, lengths of any chain, nor the occurrence frequencies of any itemset as items are not inserted or removed. To guarantee that also the pair order matrix C_D will be preserved, we must pose one additional requirement for a swap. Note that when transposing two adjacent items in the chain π, say, u and v with u originally before v, only two elements of C_D are modified. These are $C_D(u, v)$, which is decremented by one as there is one instance less of u preceding v after the transposition, and $C_D(v, u)$, which is incremented by one as now there is one instance more where v precedes u.

Obviously, if the swap would change only π, the resulting data set would no longer belong to $\mathcal{C}(D)$ as C_D was changed. Hence, we must carry out a second transposition in another chain τ that cancels out the effect the first transposition had on C_D. I.e., when we first transpose u and v in π where u precedes v, we must immediately after this transpose u and v in τ where v precedes u.

Definition Let D be a set of chains and let π and τ belong to D. The tuple (π, τ, i, j) is a *valid swap* for D, if the item at the ith position of π is the same as the item at the $j+1$th position of τ, and if the item at $i+1$th position of π is the same as the item at the jth position of τ.

The swap we made in the example above is thus a valid swap.

Given the data D, we may have several valid swaps to choose from. To see how the set of valid swaps evolves in a single step of the algorithm, consider the following example. Let D_i contain the three chains below:

$$\begin{aligned} \pi_1 :& \quad (1,2,3,4,5) \\ \pi_2 :& \quad (7,8,4,3,6) \\ \pi_3 :& \quad (3,2,6,4,1) \end{aligned}$$

The valid swaps in this case are $(\pi_1, \pi_3, 2, 1)$ and $(\pi_1, \pi_2, 3, 3)$. If we apply the swap $(\pi_1, \pi_2, 3, 3)$ we obtain the chains

$$\begin{aligned} \pi_1' :& \quad (1,2,4,3,5) \\ \pi_2' :& \quad (7,8,3,4,6) \\ \pi_3 :& \quad (3,2,6,4,1) \end{aligned}$$

Obviously $(\pi_1, \pi_2, 3, 3)$ is still a valid swap, as we can always revert the previous swap. But notice that $(\pi_1, \pi_2, 2, 1)$ is no longer a valid swap as the items 2 and 3 are not adjacent in π_1'. Instead $(\pi_2', \pi_3, 4, 3)$ is introduced as a new valid swap since now 4 and 6 are adjacent in π_2'.

Given this definition of the swap, an interesting open question is whether or not $\mathcal{C}(D)$ is connected with respect to the valid swaps? Meaning, can we reach every member of $\mathcal{C}(D)$ starting from D? Obviously this should be the case if we want to sample uniformly from all of $\mathcal{C}(D)$, but showing this is not easy.

Convergence

Above it was mentioned that we must let the chain run long enough to make sure $\tilde{D}_s$ is not correlated with the starting state D_0. This means that the chain should have *mixed*, meaning that when we stop it the probability of landing at D_s actually corresponds to the probability D_s has in the stationary distribution of the chain. Making sure a Markov Chain has converged to the stationary distribution is a difficult problem.

Hence we resort to a fairly simple heuristic for the problem of assessing the convergence of the algorithm. An indicator of the current sample D_i being uncorrelated from $D_0 = D$ is the following measure:

$$\delta(D, D_i) = |D|^{-1} \sum_{j=1}^{|D|} d_K(D(j), D_i(j)), \tag{6.4}$$

where $D(j)$ is the jth chain of D. This is always defined, as the chain $D_i(j)$ is always a permutation of $D(j)$. The distance defined in Equation 6.4 is thus the average Kendall distance between the permutations in D and D_i. To assess the convergence we see how $\delta(D, D_i)$ behaves as i grows. When $\delta(D, D_i)$ has converged to some value or is not considerably increasing, we assume the current sample is not correlated to D more strongly than to most other members of $\mathcal{C}(D)$.

Note that here we are assuming that the chains in D are *labeled*. To see what this means consider the following example with the sets D and D_i both containing four chains.

$$\begin{array}{ll} D(1) : 1,2,3 & D_i(1) : 2,1,3 \\ D(2) : 4,5,6 & D_i(2) : 6,5,4 \\ D(3) : 2,1,3 & D_i(3) : 1,2,3 \\ D(4) : 6,5,4 & D_i(4) : 4,5,6 \end{array}$$

Here we have obtained D_i from D with the multiple swap operations. The distance $\delta(D, D_i)$ is 2 even though D and D_i clearly are identical as sets. Hence, Equation 6.4 can not be used for testing this identity. To do this we should compute the Kendall distance between $D(j)$ and $D_i(h(j))$, where h is a bijective mapping between chains in D and D_i that minimizes the sum of the pairwise distances. However, if we associate with each chain a label that is not modified by the swap, then using the mapping h is no longer motivated. If we have some additional data associated with each chain, it is important to preserve the mapping between chains in D and D_r. This can happen for the chains arise for example from a survey where the respondents have been asked to rank some items, but we also have demographic information of each respondent.

Implementation issues

Until now we have discussed the approach at a fairly general level. There's also a practical issue when implementing the proposed algorithm. The number of valid swaps at a given state is of order $O(m^2|D|^2)$ in the worst case, which can get prohibitively large for storing each valid swap as a tuple explicitely. Hence, we do not store the tuples, but only use the structures A_D and S_D, defined next. We have,

$$A_D = \{\{u, v\} \mid uv \in \pi_1 \wedge vu \in \pi_2 \wedge \pi_1, \pi_2 \in D\}, \tag{6.5}$$

where $uv \in \pi$ denotes that u and v are adjacent in π with u before v. This is the set of *swappable pairs*. The size of A_D is of order $O(m^2)$ in the worst case. In addition, we also have the sets

$$S_D(u, v) = \{\pi \in D \mid uv \in \pi\} \tag{6.6}$$

for all (u, v) pairs. This is simply a list that contains the set of chains where we can transpose u and v. Note that $S_D(u, v)$ and $S_D(v, u)$ are not the same set. In $S_D(u, v)$ we have chains where u appears before v, while in $S_D(v, u)$ are chains where v appears before u. The size of each $S_D(u, v)$ is of order $O(|D|)$ in the worst case, and the storage requirement for A_D and S_D is hence only $O(m^2|D|)$, a factor of $|D|$ less than storing the tuples explicitely.

Do A_D and S_D fully represent all possible tuples of valid swaps? A valid swap is constructed from A_D and S_D by first picking a swappable pair $\{u, v\}$ from A_D, and then picking two chains, one from $S_D(u, v)$ and the other from $S_D(v, u)$. It is easy to see that a swap constructed this way must be a valid swap. Also, there are no valid swaps that are not described by A_D and S_D. Let (τ_1, τ_2, i, j) be such a valid swap. This would mean that there exist τ_1 and τ_2 in D so that item u appears at position i in τ_1 and at position $j + 1$ in τ_2, and item v appears at position $i + 1$ in τ_1 and at position j in τ_2. Obviously this means that $uv \in \tau_1$ (hence $\tau_1 \in S_D(u, v)$) and $vu \in \tau_2$ (meaning $\tau_2 \in S_D(v, u)$), and since both τ_1 and τ_2 belong to D, the pair $\{u, v\}$ must be a swappable pair and belong to A_D. Thus, (τ_1, τ_2, i, j) is represented by A_D and S_D, and the assumption of this not being the case is incorrect.

This leaves us with two additional concerns. Recall, that we want to use the Metropolis-Hastings approach to sample from the uniform distribution over $\mathcal{C}(D)$. In order to do this we must be able to sample uniformly from the neighbors of D_i, and we have to know the precise size of D_i's neighborhood. The size of the neighborhood $N(D_i)$ is precisely the number of valid swaps at D_i, and is given by

$$|N(D_i)| = \sum_{\{u,v\} \in A_{D_i}} |S_D(u, v)| \cdot |S_D(v, u)|, \tag{6.7}$$

which is easy to compute given A_{D_i} and S_{D_i}.

To sample a neighbor uniformly at random using A_{D_i} and S_{D_i}, we first pick the swappable pair $\{u, v\}$ from A_{D_i} with the probability

$$Pr(\{u, v\}) = \frac{|S_{D_i}(u, v)| \cdot |S_{D_i}(v, u)|}{N(D_i)}, \tag{6.8}$$

which is simply the fraction of valid swaps in $N(D_i)$ that affect items u and v. Then π and τ are sampled uniformly from $S_D(u, v)$ and $S_D(v, u)$ with probabilities $|S_D(u, v)|^{-1}$ and $|S_D(v, u)|^{-1}$, respectively. Thus we have

$$Pr(\{u, v\}) \cdot |S_D(u, v)|^{-1} \cdot |S_D(v, u)|^{-1} = \frac{1}{N(D_i)}$$

as required.

The final algorithm that we call SWAP-PAIRS is given in Algorithm 3. It takes as arguments the data D and the integer n that specifies the number of rounds

Algorithm 3: The SWAP-PAIRS algorithm for sampling uniformly from $\mathcal{C}(D)$.

```
1: SWAP-PAIRS($D$, $n$)
2: $A_D \leftarrow \{\{u, v\} \mid uv \in \pi_1 \wedge vu \in \pi_2 \wedge \pi_1, \pi_2 \in D\}$
3: for all $\{u, v\} \in A_D$ do
4:     $S_D(u, v) \leftarrow \{\pi \in D \mid uv \in \pi\}$
5:     $S_D(v, u) \leftarrow \{\pi \in D \mid vu \in \pi\}$
6: end for
7: $i \leftarrow 0$
8: while $i < n$ do
9:     $\{u, v\} \leftarrow$ SAMPLE-PAIR($A_D, S_D$)
10:    $\pi \leftarrow$ SAMPLE-UNIFORM($S_D(u, v)$)
11:    $\tau \leftarrow$ SAMPLE-UNIFORM($S_D(v, u)$)
12:    $s \leftarrow (\pi, \tau, \pi(u), \tau(v))$
13:    $N_a \leftarrow \sum_{\{u,v\} \in A_D} |S_D(u, v)| \cdot |S_D(v, u)|$
14:    APPLY-SWAP($s, D, A_D, S_D$)
15:    $N_b \leftarrow \sum_{\{u,v\} \in A_D} |S_D(u, v)| \cdot |S_D(v, u)|$
16:    if RAND() $\geq \frac{N_a}{N_b}$ then
17:        APPLY-SWAP($s, D, A_D, S_D$)
18:    end if
19:    $i \leftarrow i + 1$
20: end while
21: return $D$
```

the algorithm is ran. On lines 2–6 we initialize the A_D and S_D structures, while lines 8–20 contain the main loop. First, on line 9 the pair $\{u, v\}$ is sampled from A_D with the probability given in Equation 6.8. The SAMPLE-UNIFORM function simply samples an element from the set it is given as the argument. On lines 13 and 15 we compute the neighborhood sizes before and after the swap, respectively. The actual swap is carried out by the APPLY-SWAP function, that modifies π and τ in D and updates A_D and S_D accordingly. Lines 16–18 implement the Metropolis-Hastings step. Note that it is easier to simply perform the swap and backtrack if the jump should not have been accepted. A swap can be canceled simply by applying it a second time. The function RAND() simply returns a uniformly distributed number from the interval $[0, 1]$.

6.3 Experiments

We can use the random data sets to test if the clusterings we obtained in Chapter 4 are valid. For a description of the data sets, please see Section 2.1. The validity

	SUSHI	MLENS	DUBLIN	MSNBC
n	5000	2191	5000	5000
m	100	207	12	17
min. l	10	6	4	6
avg. l	10	13.3	4.8	6.5
max. l	10	15	6	8
t (sec)	297	670	73	81

Table 6.1: Properties of different data sets together with the time t that it takes for SWAP-PAIRS to perform 10^7 swaps.

of a clustering can be questioned if the clusterings we compute from the random data sets are as good or almost as good as the one we computed from the real data. We measure the goodness of a clustering with the cost function of Equation 4.4 that the clustering algorithms try to minimize. That is, we let $\mathcal{A}(D)$ to be the clustering error.

Estimating the convergence

We must first estimate how many swaps must we make between individual samples to obtain an uncorrelated set of random data sets. To this end we run the SWAP-PAIRS algorithm for 10^7 swaps on each data set and measure $\delta(D, D_i)$ every 10^5 swaps.

From Table 6.1 we can see the characteristics of the different data sets and the time it takes SWAP-PAIRS to perform 10^7 swaps. Obviously n, the number of chains in the input, does not affect the time t, but the number of items m plays a significant part.

In Figure 6.1 are four plots that show how the distance $\delta(D, D_i)$ develops with the number of swaps i for the data sets. The assumption is that when $\delta(D, D_i)$ no longer increases the current data D_i is independent of the initial state D. Based on the plots in Figure 6.1 we say that in case of the data sets SUSHI and MLENS the chain has converged after five million swaps. The plot for DUBLIN indicates that the distance δ stabilizes already after about 500 000 swaps. With MSNBC this happens roughly after three million swaps. We note that especially in case of the SUSHI data δ might still increase if we would continue swapping.

Testing clustering validity

Figure 6.2 shows the histograms of the clustering error for the different algorithms and data sets based on a sample of 100 random elements of $\mathcal{C}_D$. The error observed in real data is indicated by a red vertical line in each histogram. The

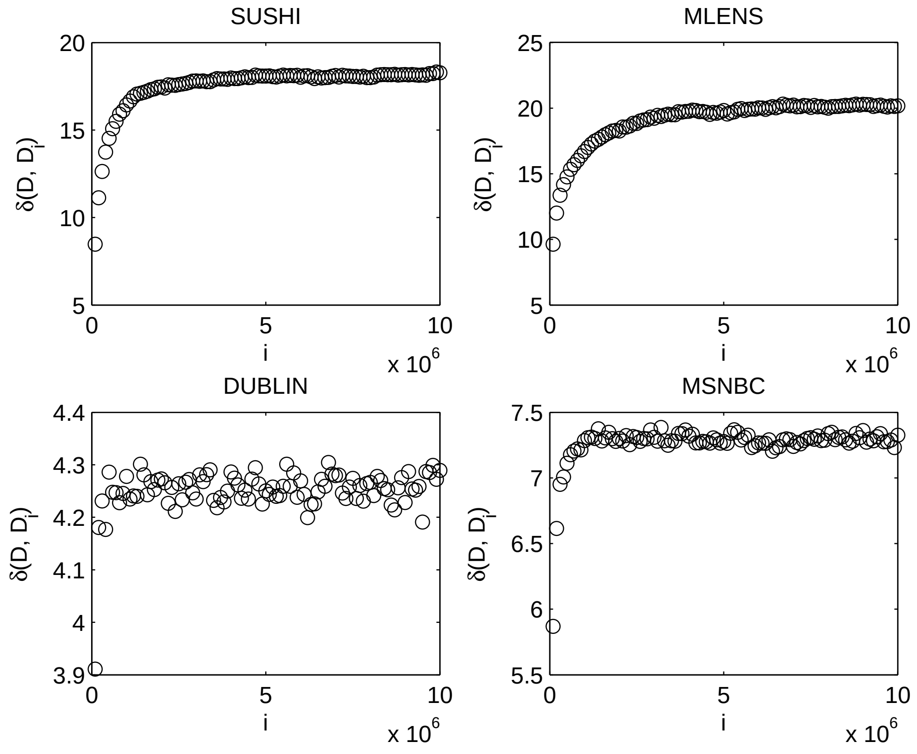


Figure 6.1: The distance $\delta(D, D_i)$ as a function of the number of swaps i.

results are mostly as expected; the clustering error in the real data is considerably smaller than the expected error in random data. There are a couple of interesting exceptions to this, however. With the DUBLIN data set expected clustering error in random data is in fact *smaller* than the true error of the real data set when the clustering is computed using the graph representation only. The same happens also with the MSNBC data with both the graph and hypersphere representations. Especially with MSNBC and graph representation the histogram suggests a bimodal distribution of the error. That is, the equivalence class of MSNBC contains two kinds of sets of chains: for some sets the graph representation results in clusterings that are "better" than the one we obtain from the true MSNBC, while for some sets the clusterings are notably worse. With both DUBLIN and MSNBC this phenomenon no longer occurs when we use Lloyd's algorithm to improve upon the initial clustering obtained based on the vector representation.

In general the results suggest that the clusterings we obtain from the actual

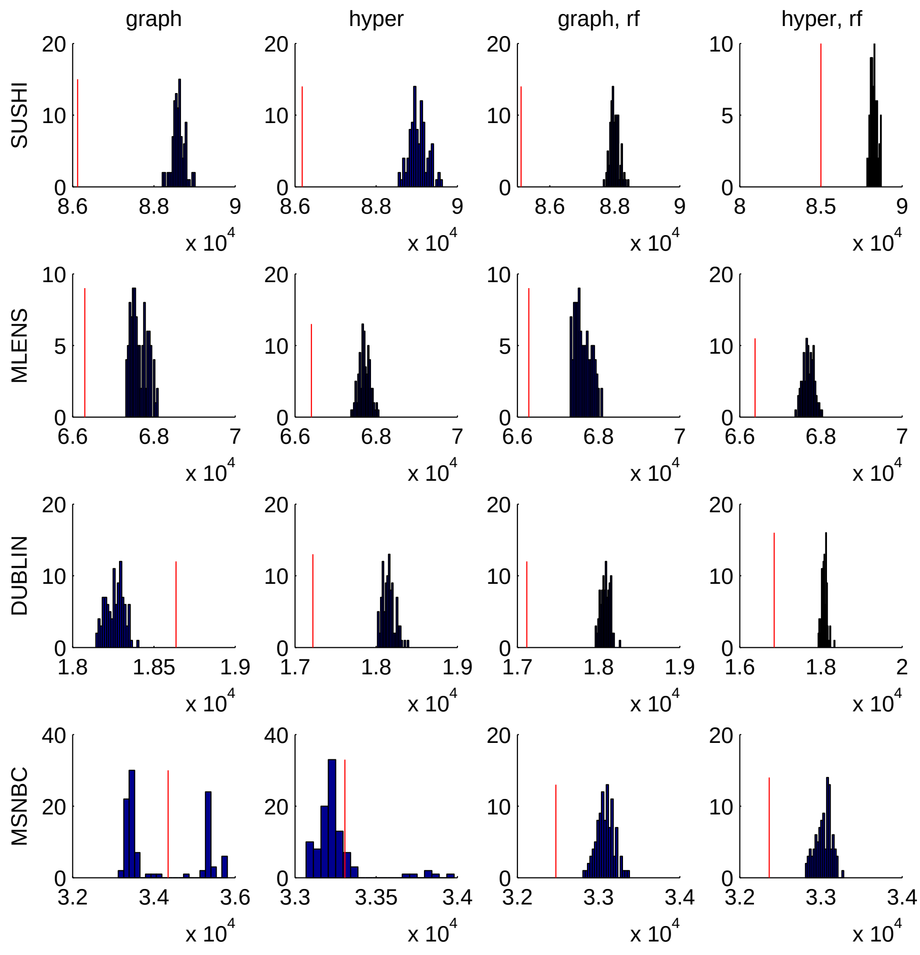


Figure 6.2: Histograms for the clustering error with different algorithms on 100 different samples from the equivalence class of the respective data set. The clustering error on the original ("real") data is indicated with a vertical line. See also Table 6.2.

	graph	hyper	graph, rf	hyper, rf
SUSHI	0.01	0.01	0.01	0.01
MLENS	0.01	0.01	0.01	0.01
DUBLIN	1.00	0.01	0.01	0.01
MSNBC	0.64	0.85	0.01	0.01

Table 6.2: The empirical p values as defined in Equation 6.1 for the clusterings in different data sets obtained by the different algorithms. The score is based on a sample of 99 random data sets from the equivalence class of the respective data sets. See also Figure 6.2.

data sets have a smaller error than a clustering computed with the same algorithm from a random data that shares certain characteristics with the real data.

6.4 Conclusion

We developed an MCMC algorithm for sampling sets of chains that all belong to the same equivalence class as a given set of chains. The algorithm is based on Metropolis-Hastings, and will sample uniformly from the set of sets that are reachable from the initial data set with a local modification that we called a swap. It is not known whether the reachable sets constitute all of the equivalence class.

The purpose for sampling the random sets of chains is randomization testing. We can assess the significance of the results we have obtained in real data by computing some statistic of the real data, and then comparing this to the same statistic computed from random sets of chains. Under the null-hypothesis there is no variation in the statistic between the real and random data sets. We say the result is significant if the statistic in the real data is considerably different from the one observed in random data.

Chapter 7

Chains, partial orders, and 0–1 data

In the previous chapters we have discussed techniques for analyzing sets of chains. To demonstrate the algorithms we performed experiments with chains arising from various applications. Ranked data, however, is not as typical as relational data. A large proportion of data mining literature is concerned with the analysis of transactional databases that can very often be represented simply as a binary matrix. Most notably this is the case with work on frequent itemset mining, see e.g. [AMS$^+$96, MT97, CG02, GZ03, HPYM04, ZH02].

In this chapter we study the relationship between orders and 0–1 data. More precisely, we will address two tasks. First, we discuss how to find chains that order the transactions based on the content of the transactions alone. A method for finding chain-like patterns from 0–1 data was already discussed in [GKM03]. We propose another technique that is not based on the levelwise algorithm. Our second task is to find a global order that covers all transactions. This order can be total, but again we argue that partial orders are a better model. We propose an approach where this order is constructed given a set of chains on the transactions.

The problem of finding an order for 0–1 data is mostly motivated by biostratigraphy, where the task is to determine the age of sediments using the fossils they contain. We use the proposed method for finding a temporal order for a number of sites where fossils have been located. The data is incomplete and contains noise, mostly in the form of false zeros, i.e., zeros that should in fact be ones. We argue that because of this the temporal order should be a partial order instead of a strict ranking of the fossil discovery sites.

We propose an algorithm for computing a set of chains from a given 0–1 data. These chains can be aggregated to a bucket order, or an arbitrary partial order on the transactions. We also discuss a method for assessing the orderability of 0–1 data in general. To this end we propose a simple randomized heuristic for testing the property that the transactions can be ordered based on their contents.

7.1 Ordering a set of 0–1 vectors

We start the discussion by considering the general problem of finding an order for 0–1 vectors. Let M be a set of n vectors $\mathbf{x} \in \{0,1\}^m$, and let π be a partial order on M. We will look at the special case where π is a total order later. We associate a *cost* with the pair (M, π), denoted $f(M, \pi)$. The task of finding an order for the vectors in M is now simply stated as follows:

Problem Given the set M of binary vectors, find the partial order π that minimizes $f(M, \pi)$.

This definition is incomplete unless we specify f. Of course the "meaning" of the found order is defined by f, and therefore it's precise definition is application dependant.

Our choice for f was originally motivated by the application of finding a temporal order for fossil discovery sites. The data is a set of 0–1 vectors that each correspond to a geographical location (or sediment at a certain location). Every dimension of the vectors corresponds to a species or genus of some organism, depending on the level of detail. In the vector representing a site all those positions have a 1 that correspond to a species of which fossil remains have been found at the site. The age difference between the youngest and oldest site is several millions of years. In this temporal scale the following assumption is (mostly) true: A species first appears, then exists for a certain time, and finally becomes extinct. Important is that a species does *not* become extinct and then re-appear later.

In practice this will happen to some extent as the fossil record is incomplete due to various reasons. Nonetheless, we assume that an ordering where there are many of such events is probably not correct in the evolutional sense. This should be captured by the cost function by assigning a higher cost for orders where there are many non-occurrences of a species (zeros) between occurrences of the same species (ones).

Species that do re-appear after their extinction are called *Lazarus species*[1] in the paleontological community. We call a zero that appears between two ones in an order a *lazarus event*. The problem of finding the order is thus to minimize the number of lazarus events.

[1]This refers to a character in the New Testament, see Gospel of John 11:1–44.

More precisely, we define the *lazarus count* as

$$f(M,\pi) = \sum_{\mathbf{y}\in M} \sum_{i=1}^{m} I\{\mathbf{y_i} = 0 \wedge \exists \mathbf{x} \in M : (\mathbf{x} \prec_\pi \mathbf{y}) \wedge \mathbf{x_i} = 1 \wedge \exists \mathbf{z} \in M : (\mathbf{y} \prec_\pi \mathbf{z}) \wedge \mathbf{z_i} = 1\}, \tag{7.1}$$

where $\mathbf{x} \prec_\pi \mathbf{y}$ denotes that the vector $\mathbf{x}$ precedes the vector $\mathbf{y}$ according to π, and $\mathbf{x_i}$ is the ith element of $\mathbf{x}$. $I\{\cdot\}$ is the indicator function that takes the value 1 when its argument is true, and the value 0 otherwise. Intuitively $f(M,\pi)$ simply counts all zeros that are both preceded and followed by a one in the order specified by π. Note that this is trivially zero when we let $\pi = \emptyset$. In practice we must somehow make sure that $\emptyset$ is not chosen as the solution unless the data really does not have any underlying order (see Section 7.2).

An interesting special case occurs if π happens to be a total order. In this case we can say the vectors in M are the rows (or columns) of a matrix ordered according to π, and the function f counts all occurrences of zeros that appear between the first and last one on every column. A matrix A (here A denotes both the matrix and the set of its rows) is said to have the *consecutive ones property* if there exists a permutation π so that $f(A,\pi) = 0$. Note that there may exist several of such permutations. If A does have the consecutive ones property, then π can be found in linear time [BL76, Hsu02]. From our point of view, however, the case when the vectors in M contain noise and a zero cost solution can not be found is more relevant. This is due to the incompleteness of the fossil record; a species may have existed at a site but it is possible that there are no fossils remaining or they simply have not been found.

Partial orders from 0–1 data

The input will thus inevitably contain noise in form of false zeros and false ones. This is the reason we think is not necessarily a good idea to model the temporal order by a total order, and consider partial orders to be a better model. To motivate this, consider the following simple example. Let

$$M = \{(1,0,0), (1,1,0), (1,0,1), (0,1,1), (0,0,1)\},$$

and define the permutation π so that we obtain the matrix

$$\begin{array}{llll}
1: & 1 & 0 & 0 \\
2: & 1 & 1 & 0 \\
3: & 1 & \mathbf{0} & 1 \\
4: & 0 & 1 & 1 \\
5: & 0 & 0 & 1.
\end{array}$$

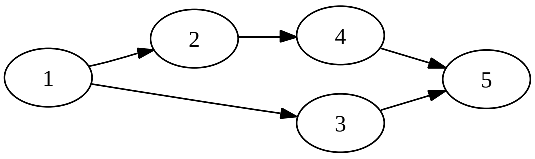


Figure 7.1: A partial order represented as a directed acyclic graph.

In this case the value of $f(M, \pi)$ is 1, the only zero that has a one appearing both before and after is in the second column and is indicated in bold above. It is easy to check that $f(M, \pi)$ is minimized for this selection of π. However, we can also see that two other permutations result in exactly the same value of f, namely:

$$\begin{array}{lccc} 1: & 1 & 0 & 0 \\ 2: & 1 & 1 & 0 \\ 4: & \mathbf{0} & 1 & 1 \\ 3: & 1 & 0 & 1 \\ 5: & 0 & 0 & 1 \end{array} \qquad \text{and} \qquad \begin{array}{lccc} 1: & 1 & 0 & 0 \\ 3: & 1 & 0 & 1 \\ 2: & 1 & 1 & \mathbf{0} \\ 4: & 0 & 1 & 1 \\ 5: & 0 & 0 & 1 \end{array}$$

All these three different permutations are all equally good in terms of the lazarus count. Choosing one of them as the "correct" solution is not necessarily optimal, as we introduce an order between some vectors (rows of the matrix) that can be more or less arbitrary. This order is neither supported nor contradicted by the data, but we can not distinguish it from a case where the data strongly suggests some vectors being ordered in a certain way. This problem is overcome when π is allowed to be an arbitrary partial order.

Next we discuss how this partial order should be chosen. In the example above, three permutations that all result in a cost of one for the given M are (1, 2, 3, 4, 5), (1, 2, 4, 3, 5) and (1, 3, 2, 4, 5). In every case 1 and 5 occupy the first and last positions, respectively. Furthermore, we see that 2 precedes 4 in all three permutations. The partial order that adheres to these properties is given in Figure 7.1. In fact, the set of linear extensions of this partial order contains exactly the three permutations listed above. The cost of this partial order is zero in terms of f. Note that this is not the case in general, usually the solution will have a nonzero cost despite being a partial order.

Unfortunately the above partial order is not unique either. We can create another partial order that has the same lazarus count as π in terms simply by reversing the direction of each arrow in Figure 7.1. Indeed, for every total order that we listed above, also the reversal has a cost of 1. It is easy to see that due to this property the following simple but computationally infeasible algorithm for finding the partial order does not work. Suppose that we enumerate all possible permutations of the transactions and compute the value $f(M, \pi)$ for each π. Then we discard every permutation except the ones with the smallest lazarus count and compute their intersection. The partial order constructed this way will only contain pairs that were contained in all of the permutations with minimal cost. Clearly this will result in the trivial (empty) partial order as the set of permutations we are intersecting contains also their reversals, and the intersection of a permutation with its reversal results in the empty set.

The direction of time

Even if enumerating all permutations of any realistic set of vectors was not unrealistic, the above suggestion would not work, because we still need to remove the reversals from the set of permutations that are used to construct the partial order. This can not be done by simply removing the reversal of each permutation at random, because we do not know which one of the permutations is the correct one. Returning to the above example, the set of permutations we obtain after checking $f(M, \pi)$ for every possible π is:

(1, 2, 3, 4, 5)	(5, 4, 3, 2, 1)
(1, 2, 4, 3, 5)	(5, 3, 4, 2, 1)
(1, 3, 2, 4, 5)	(5, 4, 2, 3, 1)

If we use permutations in either the left or right column, the result is as expected, but a priori we do not know this. We could have ended up picking (1, 2, 3, 4, 5), (5, 3, 4, 2, 1) and (1, 3, 2, 4, 5), the intersection of which does not correspond to the correct solution.

Before computing the intersection, we must partition the set of permutations to two sets so that permutations belonging to the same set are linear extensions of the same partial order. An approach for doing this based on graph partitioning is suggested in [UFM05]. This algorithm works nicely when the number of chains is not too big, but does not scale to large sets as the space requirement is quadratic in terms of the number of chains. Clearly an alternative approach is to use the clustering algorithms discussed in Chapter 4. We can say that the partial order depicted in Figure 7.1 is the first component that "generates" half of the permutations, while its reversal is the second component that generates the other half. (See Section 2.1 on a simple model that generates a set of chains given k partial orders.)

Chains from 0–1 data

The clustering algorithms provide a nice solution for deciding the direction of time, but the algorithm outlined above is still infeasible. Enumerating all permutations can not be done unless the number of transactions is very small, which typically is not the case. However, consider the following property: If π^* is a permutation that minimizes f given M, and $\pi(M')$ is the *projection* of π onto $M' \subseteq M$, then $\pi^*(M')$ minimizes f given M'. In less formal terms this means that vectors belonging to any subset M' of M are optimally ordered in the globally optimal solution for M. Note that the converse may not be true if the optimal solution (in terms of a total order) for M' is not unique. In this case it can happen that only one of them will be equal to $\pi^*(M')$.

A projection of a total order of M onto some of it's subsets is by definition a chain on M. This gives the idea for the following algorithm, which we call sample-chains. Pick a subset M' of M at random, where $|M'| = l$ is small. Compute $f(M', \pi)$ for all $l!$ chains and let X be the set of chains that minimize $f(M', \pi)$. If X contains only two chains (some π and its reversal), output both and repeat this procedure until enough chains, say n, have been collected. The reason for discarding X if it contains more than two chains is fairly obvious: if there are many permutations that have a small lazarus count on M', the algorithm has no way of telling which one of them is more likely to be compatible with the true permutation π^*.

In practice we do not want chains for which $f(M', \pi)$ is too large either. The data has only weak evidence for an order π if π has a high lazarus count, even if there are no other orders with the same lazarus count. Hence, we use a threshold σ to prune such chains from the final output. This threshold is crucial to the performance of the algorithm. If σ is too high, we get many chains that have only low evidence in the data. If σ is too low, sampling will be very slow, as only a small fraction of random subsets of M are likely to result in a chain being created.

See Algorithm 4 for an outline of sample-chains. If X is output by the algorithm, it contains only two chains, π and its reversal π^R. We know π is the optimal order of M', and hence in any of the globally optimal solutions vectors in M' are ordered in this way. The chains we obtain are all thus compatible with the globally optimal solutions. The algorithm can also be seen as an implementation of the following simple heuristic: π is a good order for some $M' \subset M$ if the lazarus count of M' given π is low, and there are no other π' with equally low lazarus counts for M'. We say that a subset M' and a permutation π form a *valid chain* given l and σ when the lazarus count $f(M', \pi) \leq \sigma$ and there are no other permutations π' so that $f(M', \pi') \leq \sigma$ as well. The algorithm sample-chains computes a sample from the set of all valid chains in the given data set M.

Finally, we can use the techniques discussed in Chapter 3 for constructing a bucket order on M. In [UFM05] the aim was to construct an arbitrary partial order, but as we stated in Chapter 3, they can be difficult to interpret especially if the number of ranked items is large.

Algorithm 4: The SAMPLE-CHAINS algorithm.

```
1: SAMPLE-CHAINS( $M$, $l$, $n$, $\sigma$ )
2: $i \leftarrow 0$
3: while $i < n$ do
4:    $M' \leftarrow$ a random $l$-sized subset of $M$
5:    $X \leftarrow \{\pi^* \mid \pi^* = \arg\min_\pi\{f(M',\pi)\} \wedge f(M',\pi^*) \leq \sigma\}$
6:    if $|X| = 2$ then
7:       output $X$
8:       $i \leftarrow i + 1$
9:    end if
10: end while
```

7.2 Assessing the orderability of a set of 0–1 vectors

So far we have assumed that looking for an order for the vectors in M by using the lazarus count given in Equation 7.1 is feasible. But as already stated in the introduction, a meaningful order can be expected to be found only for certain data sets. The approach described above will always find a set of chains that we can use to construct a partial order on the 0–1 vectors, provided the threshold σ is set appropriately. Given a new set of binary vectors, how can we assess its "orderability"? We pose the question of orderability in terms of matrices. Let M denote also the matrix that we obtain when we use the vectors in the set M as its rows. We say a set of vectors M is orderable if the matrix associated with it is orderable.

We say a 0–1 matrix M is orderable in terms of a cost function f, if f can be used to distinguish between "good" and "bad" orders for its rows. Obviously if all permutations have more or less the same cost in terms of f, it is impossible to say which one of them is correct, and hence we can not use f to find an order for M.

Let A be an orderable matrix. We define the matrix B as follows. Let $B = A + F_p^- + F_q^+$, where

$$F_p^-(u,v) = \begin{cases} -1 & \text{with probability } p \text{ iff } A(u,v) = 1, \\ 0 & \text{otherwise,} \end{cases}$$

and

$$F_q^+(u,v) = \begin{cases} 1 & \text{with probability } q \text{ iff } A(u,v) = 0, \\ 0 & \text{otherwise.} \end{cases}$$

That is, the matrix F_p^- introduces a false zero to A with probability p and the matrix F_q^+ introduces a false one to A with probability q. This model is motivated by the paleontology application. We assume that under perfect conditions all fossils would have been formed and were also all found. This ideal input has the consecutive ones property. But due to various sources of errors, we end up observing something that has a number of ones missing and a few false ones added.

When p and q are both 0.5, it is easy to see that the matrix A has no effect on the resulting matrix B. The matrix B will have a 1 or 0 at any given element with probability 0.5. Such completely random matrices are obviously not orderable. However, if p and q are not "too large", we can expect the orderability property having remained in B. In general we do not know either p or q.

Instead of trying to estimate these we address the following question: can we somehow distinguish the B we observe from a matrix that for sure is *not* orderable but has the same column frequencies? If we can, then this supports the assumption that B indeed was generated by the model described above and the order we find is relevant.

We will discuss a simple randomized method for analyzing this. Let τ be a random permutation on M, and let M' be a random subset of M of size l, where the distribution over the respective sample spaces is assumed to be uniform. Let Z be a random variable with the value $f(M', \tau(M'))$. The test that we propose is based on comparing the empirical distribution of Z that is obtained by sampling M, with the distribution that Z has under the zero hypothesis of M not being orderable.

If the vectors in M are not orderable using the cost function f defined in Equation 7.1, then every permutation on M' should have almost the same cost in terms of f. This is the case when the vectors in M are generated by the following simple stochastic process. We let

$$\mathbf{x_i} = \begin{cases} 1 & \text{with probability } p_i, \\ 0 & \text{with probability } 1 - p_i. \end{cases} \tag{7.2}$$

Suppose $l = 3$ and consider the ith coordinate of the vectors in M'. A zero can occur between two ones obviously only if there are two ones and one zero. This happens with probability $p_i^2(1 - p_i)$. Three items can be ordered in 6 different ways, of which two will lead to the forbidden configuration, hence the probability of coordinate i contributing a cost of 1 to Z is $q = \frac{1}{3}p_i^2(1 - p_i)$. If we denote this event with X_i we have $Z = \sum_i X_i$. When $p_i = p_j$ for all i and j this is clearly the Binomial distribution. In the more realistic case where the p_is are different, Z is the sum of independent but non-identically distributed Bernoulli variables, also called Poisson trials.

To demonstrate the idea, we generate the sets A_1 and A_2 that both contain 200 200-dimensional vectors. The set A_1 is constructed so that there exists a permutation π so that $f(A_1, \pi) = 0$, i.e., A_1 satisfies the consecutive ones

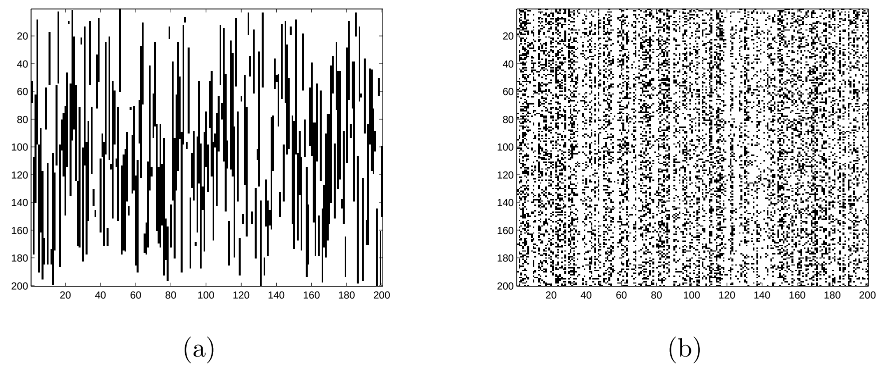


Figure 7.2: Two matrices having the same column frequencies but with different structure: matrix A_1 in (a) has the consecutive ones property while matrix A_2 in (b) contains noise.

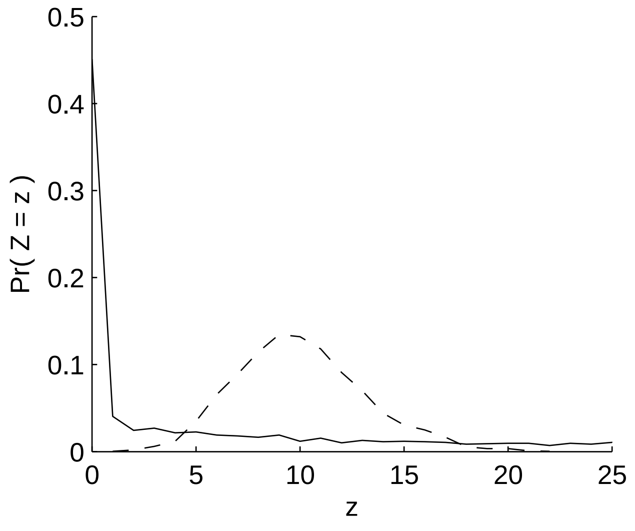


Figure 7.3: Examples of empirical probability distributions of Z when the input vectors either have the consecutive ones property as in subfigure (a) of Fig. 7.2 (solid line) or contain noise as in subfigure (b) of Fig. 7.2 (dashed line).

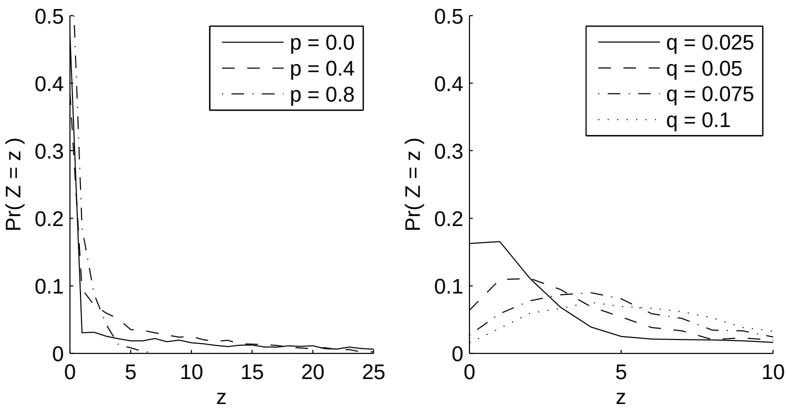


Figure 7.4: Left: Empirical distribution of Z for different values of p when $q = 0.0$. Right: Empirical distribution of Z for different values of q when $p = 0.0$.

property. Set A_2 is generated by the model of Equation 7.2. Both matrices have the same column frequencies, i.e., the number of ones on the ith column of A_1 is the same as the number of ones on the ith column of A_2. These column frequencies were chosen uniformly at random from $[0, 1]$. See Figure 7.2 for examples. Subfigure (a) shows an instance of matrix A_1 while in subfigure (b) the matrix A_2 is shown. In Figure 7.3 we have plotted the empirical distribution of Z for both A_1 (solid line) and A_2 (dashed line). Obviously the distributions are of a completely different shape. The one corresponding to A_2 is symmetric and has short tails, whereas the one based on sampling from A_1 has a lot of weight on $Z = 0$ and a long tail.

This shows the extreme cases of any input. Recall, that in general we observe the matrix B that is of the form $A + F_p^- + F_q^+$, where the matrices F_p^- and F_q^+ introduce false zeros and false ones, respectively. How sensitive is the distribution of Z to p and q? In Figure 7.4 we plot the empirical distribution of Z (Again using the same 200×200 matrix A_1 as the underlying component.) for different values of p and q while the other one is kept fixed at zero. We observe that the influence of p is considerably weaker than that of q. The difference between $p = 0$ and $p = 0.4$ is smaller than the difference between $q = 0.025$ and $q = 0.1$. In other words, false zeros have less effect on the final shape of the distribution of Z, while already a small fraction of false ones changes it by a noticeable margin.

Finally, we note that the column frequencies affect the shape of the distribution as well. If the input matrix is very sparse, we can expect the probability of $Z = 0$ to be higher also for matrices generated using Equation 7.2. It is thus

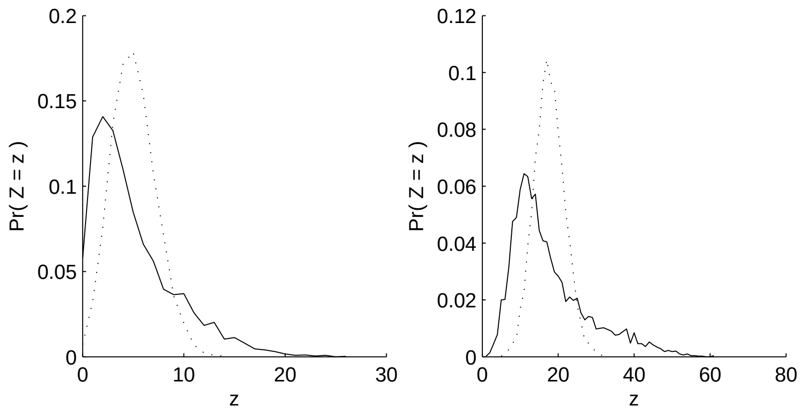


Figure 7.5: Left: Empirical distribution of Z (solid line) when $B = A_1 + F^-_{0.5} + F^+_{0.05}$ and the reference distribution (dotted line). Right: Empirical distribution of Z (solid line) when $B = A_1 + F^-_{0.1} + F^+_{0.2}$ and the reference distribution (dotted line).

important to always compare the distribution obtained from the real input to a matrix with the same column frequencies.

Distinguishing orderable matrices from ones that are not orderable is a matter of comparing the empirical distribution of Z obtained from the real data with the reference distribution obtained from a matrix based on Equation 7.2. If the distribution based on the real data has a higher probability for $Z = 0$ and a longer tail than the reference distribution, it *may* be possible that the input vectors indeed are orderable. However, the order can be ambiguous. For example, if the input matrix looks like a chessboard (1s for white squares and 0s for black squares), in a zero cost solution the rows with even indices must be grouped together, and placed either before or after the rows with odd indices. The order within the even (and odd, respectively) indices is not determined. There are thus a very large number of equally good permutations, but still the empirical distribution of Z will have $Pr(Z = 0) = 1$. Therefore, to be precise, this heuristic can only be used to rule out inputs that are essentially noise and hence not orderable.

7.3 Experiments

In this section we describe experiments performed on a paleontological data set. We make use of the analysis techniques discussed in the earlier chapters, and this way the section also serves the purpose of demonstrating the techniques in a practical setting. The data set we use, called G10S10, is a 124×139 0–1 matrix, where the rows correspond to sites and columns to different genera. The 1s indicate that a fossil belonging to a certain genus was found at a certain site. The task is to find a temporal order for the sites that is evolutionally feasible, i.e., extinct genera do not re-appear later, as argued in Section 7.1.

Especially we are interested in the number of temporal classes (or buckets) that the sites should be divided to. The current system used by the paleontologists divides the sites in G10S10 to 14 different classes. The results we obtained in Chapter 3 when combining the MCMC algorithm of [PFM06] with the BUCKET-PIVOT algorithm indicate on the average about 17 classes in G10S10. This number is not fully reliable, however. The algorithm of [PFM06] is also based on the lazarus count principle, and is hence also unable to distinguish between the two directions of time: the one where the sites become older and the one where they become younger. In [PFM06] this problem is solved by restricting the Markov Chain to those permutations that correspond to only one of the directions. This is done by fixing the order of a subset of 11 sites using so called *hard ages* that are determined for some sites based on other data. There is a small possibility that the number of buckets observed in the pair order matrix of G10S10 is an artefact of this. The approach discussed in this chapter allows us to construct the pair order matrix without resorting to any additional data about the ages of the sites.

Sampling chains from G10S10

We start by evaluating the orderability of the G10S10 data. To this end we compute the empirical distribution of the Z variable and compare it with its distribution under the zero hypothesis as described in the previous section. The data set itself along with the results is shown in Figure 7.6. The solid line indicates the empirical distribution of Z computed from the matrix shown on the left in Figure 7.6. The dotted line is the reference distribution. They are obviously not identical, and hence we can not rule out the possibility of G10S10 being orderable.

The next task is to pick a number of chains from the G10S10 matrix using the algorithm of Section 7.1. As the algorithm has to check every possible permutation of the chosen subset of rows, looking for long chains is not very efficient. We want to have a large number of chains, so we can not spend too much time per candidate. The length of the chain is a compromise between efficiency and quality; if the chains are too short we encounter problems when separating the different directions from each other. In Chapter 4 the experiments indicate that obtaining a correct clustering with chains shorter than 5 items tends to be difficult. We let $l = 5$, as this leads only to $5! = 120$ permutations to check,

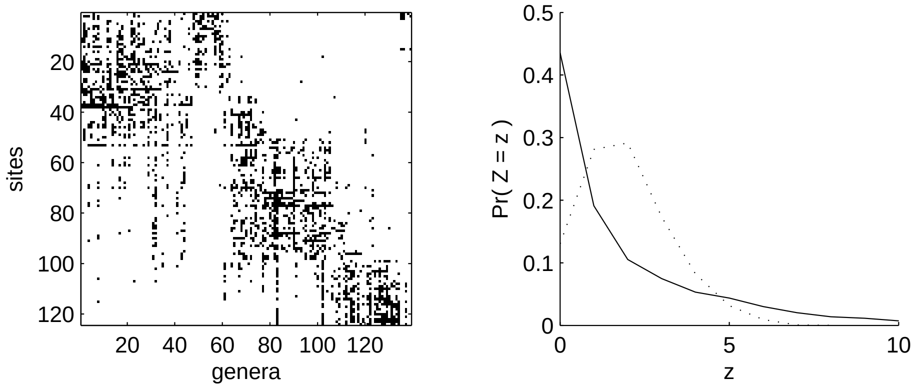


Figure 7.6: Left: The G10S10 data set with rows ordered according to the estimated age of the sites. Right: Distributions of Z for the real data (solid line) and the random data (dotted line).

and chains of length 5 can usually be clustered successfully with the algorithms proposed in Chapter 4.

Setting σ is another interesting question, for which we do not have a final answer. We can experimentally investigate how SAMPLE-CHAINS behaves for different values of σ, i.e., see how long it takes to sample n chains. If we run the algorithm with $\sigma = 0$, meaning that a chain $\pi(M')$ is accepted only if $f(M', \pi) = 0$ we obtain 1000 chains in approximately 15 minutes[2]. In this time the algorithm evaluates roughly 1.1 million different subsets of M. When σ is increased to 10, we obtain 1000 chains in less than a minute. Sampling 50000 chains from G10S10 with $l = 5$ and $\sigma = 10$ takes 40 min and the algorithm evaluates about 3.1 million subsets of the rows.

Note that there are about 225 million subsets of size five in total when $m = 124$. Thus, given that the current implementation can process roughly 1 million subsets (when $l = 5$) in 15 minutes, it would not be infeasible to evaluate them all and this way obtain all chains that are valid. This would take about 56 hours. However, our main motivation for sampling the chains is to use them for computing the normalized pair order matrix $\mathcal{C}_D$ for the sites. The set of chains we get with SAMPLE-CHAINS is a uniform sample of all possible chains that are valid given l and σ. It is easy to see that the relative frequency of chains where u precedes v is the same in the entire set of valid chains and in a uniform sample of

[2]Using a fairly unoptimized Java implementation of the algorithm running on a 1.8GHz CPU.

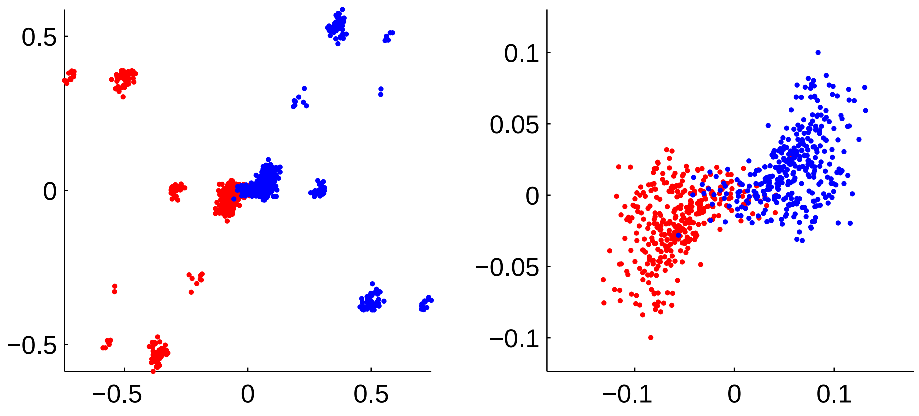


Figure 7.7: Left: A scatterplot of 1000 chains sampled from the G10S10 data set with $\sigma = 0$. Right: Detail of the central part of the left figure. In both figures the colors indicate a 2-way clustering of the chains. Even though the clusters overlap in the plot on the right they correspond as expected to the two sets of chains that point to different directions but are otherwise identical.

this set. Hence, the benefits of having all valid chains is negligible in the current application when compared to the additional cost of computing them.

Finding the direction of time

Recall, that for every chain π in the sample we obtain using SAMPLE-CHAINS, also its reversal π^R belongs to the sample. This means that one half of the chains correspond to an order where the sites "become younger", while in the other half they "become older". To resolve the direction of time, we have to partition the set of chains so that chains in the same group point to the same direction. As suggested above in Section 7.1, this can be done using the clustering algorithms of Chapter 4. We can also use the visualization techniques of Chapter 5 to see how the set of chains we have sampled is structured, and if the clustering really seems to correspond to the correct partition.

In Figure 7.7 we have plotted a set of 1000 chains that are valid given $\sigma = 0$. The plot is created using the hypersphere representation and PCA. On the left side of the figure is a plot showing all chains, while the plot on the right shows the group of points in the very center of the left plot. The clustering is indicated by the colors. Maybe somewhat surprisingly there are not only two groups of points, but several small ones with a bigger cluster of points in the center. However, it is relatively easy to see that the the figure is symmetric with respect to the origin. For every red point there is a corresponding blue point on the opposite side of

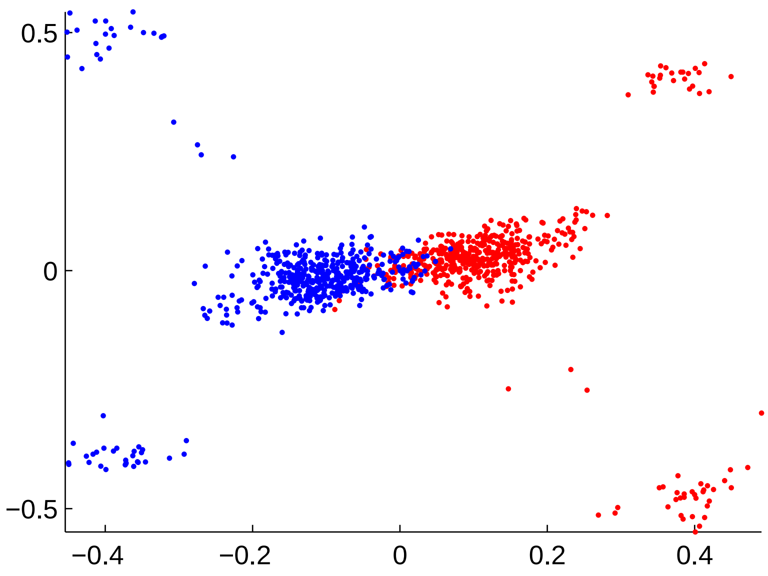


Figure 7.8: A scatterplot of 1000 chains sampled from the G10S10 data set with $\sigma = 10$. The colors indicate a 2-way clustering of the chains.

the origin. This indicates that the clustering has separated the two directions as expected.

When we cluster the set of 50000 chains that we sampled with $\sigma = 10$ and pick a random subset of 1000 chains for visualization, we obtain the scatterplot in Figure 7.8. As above, the chains are distributed symmetrically around the origin. Note that unlike in Figure 7.7 the figure is not strictly symmetric with respect to the origin. This is because we plot only a random subset of the 50000 chains. Also in this case the clustering algorithm has found the two directions of time.

Bucket orders for sites

We can construct the normalized pair order matrix $\hat{C}_D$ using chains from one of the clusters found above in the set of 50000 chains. This matrix is shown on the left in Figure 7.9, note that the rows and columns of $\hat{C}_D$ have been sorted in order of the row-wise sums. When looking at the left matrix, we observe some gray elements in the upper right and lower left corners, meaning that for some sites u and v $\hat{C}_D(u, v)$ is very close to 0.5.

These values correspond to pairs (u, v) where site u is a old and site v young. It is unlikely that old and young sites occur together in a chain, because they

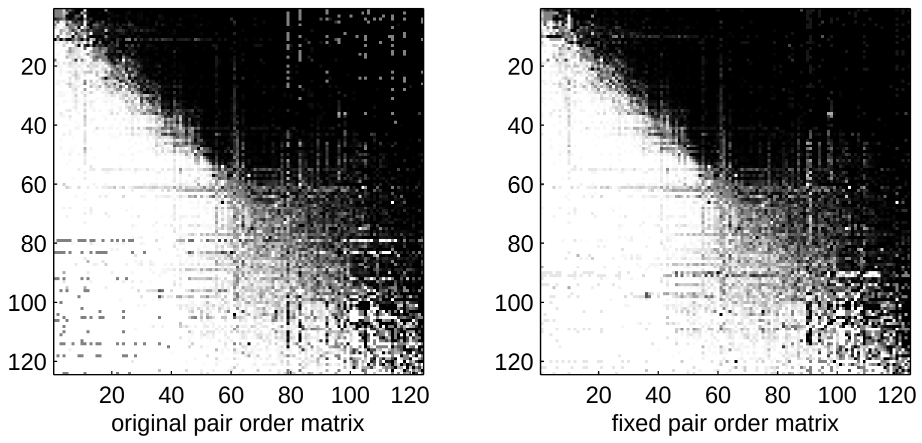


Figure 7.9: Left: The normalized pair order matrix for G10S10 computed from 25000 chains. Right: The normalized pair order matrix for G10S10 with missing values inferred by transitivity. Black corresponds to value 1.0, white to 0.0.

tend to have no common species, and can hence be ordered arbitrarily without affecting $f(M', \pi)$. This increases the number of permutations that have a lazarus count less than σ, which prevents the candidate subset M' from being a valid chain. If the input D does not contain any instances where either u precedes v or vice versa, we have $\hat{C}_D(u, v) = \hat{C}_D(v, u) = 0.5$ by definition. Note that this does not occur when the pair order matrix is built based on total orders on the sites, as is the case in Figure 3.5 on page 34.

This phenomenon can cause problems if we want to apply the BUCKET-PIVOT algorithm discussed in Chapter 3 to construct a bucket order on the sites. If the pivot site is chosen so that it belongs to many of such pairs, the resulting bucket order will have many errors.

Even though we may have no explicit information about the order of some u and v, we can still infer their order by transitivity from the other pairs. Suppose the set of chains we sampled has many occurrences of (u, w) and (w, v) for different w but none of (u, v), we can say that u should precede v. We implement this idea in the following simple algorithm for "fixing" a normalized pair order matrix that is obtained from a set of chains:

1. Let U_D denote the set of unordered pairs $\{u, v\}$, st. u and v never occur together in any chain of the input D.
2. For all $\{u, v\} \in U_D$, let $T(u, v)$ denote the number of different w, st. $\hat{C}_D(u, w) > \gamma$ and $\hat{C}_D(w, v) > \gamma$.

	Bp	Bp-pr
avg. c_C	850.35	847.42
min. c_C	682.97	674.97
max. c_C	1415.29	1426.45
std. c_C	129.96	132.63
avg. c_{MN}	1198.43	1189.77
min. c_{MN}	788.50	780.50
max. c_{MN}	1988.00	1979.50
std. c_{MN}	185.48	187.01
avg. N_b	32.72	29.29
min. N_b	20.00	15.00
max. N_b	42.00	40.00
std. N_b	3.41	4.54

Table 7.1: Results of the BUCKET-PIVOT algorithm with (right column) and without pruning. The numbers are based on 1000 independent runs.

3. For all $\{u, v\} \in U_D$ with $T(u, v) > \delta$, let $\hat{C}_D(u, v) = 1$ and $\hat{C}_D(v, u) = 0$.

Above $\gamma \in (0.5, 1]$ and $\delta \in \mathbb{N}$ are free parameters.

When the above algorithm is applied to the left matrix in Figure 7.9, we obtain the matrix on the right. This no longer contains any missing pairs, and is hence better suited to be used as input for BUCKET-PIVOT. When BUCKET-PIVOT is run 1000 times with the fixed pair order matrix of Figure 7.9 as the input, we obtain the results shown in Table 7.1. Here BP denotes the regular algorithm, and BP-PR the variant that uses pruning to reduce the number of redundant buckets (see Section 3.2).

When Table 7.1 is compared with Table 3.1 on page 3.1, we notice that the average number of buckets found is considerably higher (32.7 and 29.3 with BP and BP-PR, respectively) when the pair order matrix is built from chains instead of permutations (here average N_b equals 17.5 and 16.9). This result indicates that using the chains a more fine grained order of the sites can be obtained. Whether this means that the true number of classes in G10S10 is higher than 14, as indicated by the current MN-system, would require additional experiments.

7.4 Conclusion

We have proposed a simple method for sampling chains from a set of 0–1 vectors. Each chain defines a total order on some small subset of the vectors. Moreover, the chains can be combined to produce a global order on the vectors. As a practical example we used the method to produce a bucket order for fossil discovery sites given occurrence information of fossils of different species found at the sites.

The method is based on a measure we call the lazarus count which is related to the consecutive ones property of 0–1 matrices. A small drawback of this approach is that we obtain the chains in "both directions". The problem of separating these two directions from each other is a clustering problem, and is easily solved with one of the algorithms presented in Chapter 4 on page 53.

Another issue with the approach is that it will always yield some chains even if the input is inherently not orderable using the lazarus count measure. To tackle this we also discussed a heuristic for assessing the orderability of a set of 0–1 vectors.

Chapter 8

Discussion

In this thesis we have discussed algorithms for analyzing sets of chains. More specifically, we have considered the following tasks: finding bucket orders, clustering chains, visualizing sets of chains, randomization testing for sets of chains and sampling chains from 0–1 data. The work has been exploratory in the sense that no specific real world problem was targeted by the research. However, we have demonstrated the proposed methods on real data sets that arise from different application domains. Probably the most important application we have considered is that of biostratigraphy, where the task is to determine the age of rock sediments based on the fossil record.

For the problem of finding bucket orders we presented a constant factor approximation algorithm in Chapter 3. The problem is closely related to that of rank aggregation, where the task is to find a total that is as compatible as possible with a set of total orders. A key proposition that we make in this thesis is that in some cases bucket orders are a better model than total orders for representing sets of orders (permutations or chains).

In Chapter 4 we considered the problem of clustering a set of chains. By clustering we mean the usual problem of partitioning a set of items so that items similar to each other are placed in the same cluster, while items dissimilar to each other end up in different clusters. We gave a simple variant of Lloyd's iteration for solving the k-means problem in case of chains. We also compared our algorithm to that of [KA06]. The results indicate that the methods tend to perform equally well with our algorithm outperforming [KA06] by a small margin in a number of cases. We also gave two approaches for representing chains in a vector space.

Visualization can be a powerful tool in an early stage of data analysis. In Chapter 5 we discussed how existing dimension reduction techniques can be combined with the vector space representations of Chapter 4 to form scatterplots of sets of chains. We use measures from information retrieval and machine learning to evaluate the visualizations.

The topic of Chapter 6 is related to recent work on randomization testing of data mining results. Here the problem is to construct random data sets that share some well specified characteristics with the original data. Results computed from the random data sets are then compared with the results from real data. If these do not differ considerably (according to some statistic) we can question the significance of the found results. In Chapter 6 we gave an MCMC algorithm for sampling sets of chains. We used the algorithm to assess the significance of the clusterings found in Chapter 4.

Chapter 7 deals with a slightly different problem than the rest of this work. There we do not discuss a method for analyzing chains, but instead propose a simple algorithm for sampling chains from 0–1 data. The assumption is that an order for 0–1 vectors can be deduced from the contents of the vectors. This approach is largely motivated by the paleontology application. As a first task, we are interested in finding total orders for small subsets of the transactions. Only some subsets of the transactions can be ordered according to the chosen cost function, these are the valid chains. Instead of attempting to find all valid chains, of which there can be a number that is exponential in the number of vectors in the input, we are satisfied with a sample of this set. Our second task is to construct a global order for the 0–1 vectors. We discuss why it is better to use partial orders than total orders and give a simple heuristic for assessing whether such an order can be found given a set of 0–1 vectors.

The problems discussed in this thesis are by no means fully understood. Our approach to finding a bucket order is based on pairwise probabilities of the items to precede one another. By using some other definition for an optimal bucket order, it might be possible to develop algorithms that make use of the given set of chains directly instead of first estimating these probabilities. Also, it is not obvious that the relatively simple clustering approach as taken in Chapter 4 and also in [KA06] is the best way of finding multiple orders to describe a set of chains. When constructing the visualizations in Chapter 5, we have in many ways also only scratched the surface of the problem. Dimension reduction algorithms that would take the structure of the space in which the chains reside properly into account are yet another topic for further research.

Chapter 6 discusses the SWAP-PAIRS algorithm for sampling sets of chains. There are two important theoretical questions related to this. First, is the equivalence class as defined in Section 6.2 connected with respect the swap operation defined in Section 6.2? If not, then SWAP-PAIRS will sample only from a restricted subset of the equivalence class. How restricted is this subset? Moreover, can something precise be said about the convergence of SWAP-PAIRS, i.e., is it rapidly mixing?

Finally, the question of finding orders from 0–1 data certainly warrants further research. We see that there are two ways of extending the work. On one hand it would be interesting to develop more efficient algorithms that find chains based on the lazarus count principle. On the other hand one can study the problem of ordering 0–1 data in a more general setting. What kinds of other cost functions can be used to find orders? What common properties should they satisfy, and are there any efficient algorithms that could make use of these properties?

Appendix A

Proof of Theorem 4.3.1

The proof is a simple matter of upper bounding the equation

$$p = \binom{m}{l}^{-1} \sum_{i=2}^{l} \binom{l}{i}\binom{m-l}{l-i}.$$

First we note that using Equation 5.22 of [GKP94] (Vandermonde's convolution) the sum above can be rewritten as

$$\binom{m}{l} - \underbrace{\Big(\binom{l}{1}\binom{m-l}{l-1} + \binom{m-l}{l}\Big)}_{A}.$$

Essentially Vandermonde's convolution states that $\sum_{i=0}^{l} \binom{l}{i}\binom{m-l}{l-i} = \binom{m}{l}$, and we simply subtract the first two terms indicated by A, because above the sum starts from $i = 2$. Using simple manipulations we obtain

$$A = \binom{m-l}{l}\Big(\frac{l^2}{m-2l+1} + 1\Big),$$

which gives the following:

$$p = \binom{m}{l}^{-1}\Big(\binom{m}{l} - \binom{m-l}{l}\Big(\frac{l^2}{m-2l+1} + 1\Big)\Big).$$

Since the part $\frac{l^2}{m-2l+1}+1$ is always larger than 1, we have

$$\begin{aligned}
p &< \binom{m}{l}^{-1}\left(\binom{m}{l}-\binom{m-l}{l}\right)\\
&= 1-\binom{m}{l}^{-1}\binom{m-l}{l}\\
&= 1-\frac{(m-l)!}{l!(m-2l)!}\cdot\frac{l!(m-l)!}{m!}\\
&= 1-\frac{(m-l)(m-l-1)\cdots(m-2l+1)}{m(m-1)\cdots(m-l+1)}\\
&< 1-\frac{(m-l)(m-l-1)\cdots(m-2l+1)}{m^l}\\
&< 1-\frac{(m-2l+1)^l}{m^l}\\
&< \frac{m^l-(m-2l)^l}{m^l}.
\end{aligned}$$

We can factor $m^l-(m-2l)^l$ as follows:

$$\begin{aligned}
m^l-(m-2l)^l &= (m-(m-2l))\Big(m^{l-1}(m-2l)^0+m^{l-2}(m-2l)^1+\dots\\
&\quad \dots+m^1(m-2l)^{l-2}+m^0(m-2l)^{l-1}\Big)\\
&= 2l\sum_{i=0}^{l-1}m^{l-1-i}(m-2l)^i.
\end{aligned}$$

Using this we write

$$\frac{m^l-(m-2l)^l}{m^l}=2l\sum_{i=0}^{l-1}(\frac{1}{m})^l m^{l-1-i}(m-2l)^i.$$

Letting $a=l-1$ and taking one $\frac{1}{m}$ out of the sum we get

$$\begin{aligned}
&\frac{1}{m}2(a+1)\sum_{i=0}^{a}(\frac{1}{m})^a m^{a-i}(m-2(a+1))^i\\
=\;&\frac{1}{m}2(a+1)\sum_{i=0}^{a}(\frac{1}{m})^i(m-2(a+1))^i\\
=\;&\frac{1}{m}2(a+1)\sum_{i=0}^{a}(1-\frac{2(a+1)}{m})^i.
\end{aligned}$$

We assume $l = a + 1$ is considerably smaller than m, and hence $(1 - \frac{2(a+1)}{m})^i$ is at most 1. There are $a + 1$ terms in the sum, so the above is upper bounded by

$$\frac{1}{m}2(a+1)(a+1) = \frac{1}{m}2l^2,$$

which concludes the proof of the theorem. □

Bibliography

[Ach03] D. Achlioptas. Database-friendly random projections: Johnson-Lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66:671–687, 2003.

[ACN05] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: ranking and clustering. In *Proceedings of the 37th ACM Symposium on Theory of Computing*, pages 684–693, 2005.

[AIS93] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, 1993.

[Alo06] N. Alon. Ranking tournaments. *SIAM Journal on Discrete Mathematics*, 20:137–142, 2006.

[AMS$^+$96] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.

[BC89] J. Besag and P. Clifford. Generalized Monte Carlo significance tests. *Biometrika*, 76(4):633–642, 1989.

[BC91] J. Besag and P. Clifford. Sequential Monte Carlo p-values. *Biometrika*, 78(2):301–304, 1991.

[BG97] I. Borg and P. Groenen. *Modern Multidimensional Scaling.* Springer, 1997.

[BH67] G. H. Ball and D. J. Hall. A clustering technique for summarizing multivariate data. *Behavioral Science*, 12:153–155, 1967.

[BL76] S. Booth and G. S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *Journal of Computer and System Sciences*, 13:335–379, 1976.

[BN02] M. Belkin and P. Niyogi. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *Advances in Neural Information Processing Systems 14*, pages 585–591, 2002.

[CC03] G. W. Cobb and Y. P. Chen. An application of Markov Chain Monte Carlo to community ecology. *The American Mathematical Monthly*, 110(4):265–288, 2003.

[CFR06] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 776–782, 2006.

[CG02] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In *Principles of Data Mining and Knowledge Discovery, 6th European Conference, PKDD 2002*, pages 74–85, 2002.

[CK01] A. Condon and R. M. Karp. Algorithms for graph partitioning on the planted partition model. *Random Structures and Algorithms*, 18(2):116–140, 2001.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* MIT Press, 2nd edition edition, 2001.

[Cri85] D. Critchlow. *Metric Methods for Analyzing Partially Ranked Data*, volume 34 of *Lecture Notes in Statistics.* Springer-Verlag, 1985.

[DG77] P. Diaconis and R. L. Graham. Spearmans footrule as a measure of disarray. *Journal of the Royal Statistical Society. Series B*, 39(2):262–268, 1977.

[DH73] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[DH97] P. Demartines and J. Herault. Curvilinear component analysis: A self-organizing neural network for nonlinear mapping of data sets. *IEEE Transactions on Neural Networks*, 8(1):148–154, 1997.

[DKNS] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation revisited. (unpublished manuscript).

[DKNS01] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th International World Wide Web Conference (WWW 10)*, 2001.

[GCSR04] A. Gelman, J. B. Carlin, H. S. Stern, and D. B. Rubin. *Bayesian Data Analysis*. Texts in Statistical Science. Chapman & Hall, CRC, 2004.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-completeness*. Freeman, 1979.

[GKM03] A. Gionis, T. Kujala, and H. Mannila. Fragments of order. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 129–136, 2003.

[GKP94] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 2nd edition, 1994.

[GMMT07] A. Gionis, H. Mannila, T. Mielikäinen, and P. Tsaparas. Assessing data mining results via swap randomization. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 1(3), 2007.

[GMPU06] A. Gionis, H. Mannila, K. Puolamaki, and A. Ukkonen. Algorithms for discovering bucket orders from data. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 561–566, 2006.

[GZ03] B. Goethals and M. J. Zaki, editors. *Proceedings of the Workshop on Frequent Itemset Mining Implementations (FIMI–03)*, volume 90 of *CEUR-WS*, 2003.

[HKM$^+$01] J. Himberg, K. Korpiaho, H. Mannila, J. Tikanmäki, and H. Toivonen. Time series segmentation for context recognition in mobile devices. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 203–210, 2001.

[HMT07] N. Haiminen, H. Mannila, and E. Terzi. Comparing segmentations by applying randomization techniques. *BMC Bioinformatics*, 8(171), 2007.

[HPYM04] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[HR02] G. Hinton and S. Roweis. Stochastic neighbor embedding. In *Advances in Neural Information Processing Systems 15*, pages 833–840, 2002.

[Hsu02] W. L. Hsu. A simple test for the consecutive ones property. *Journal of Algorithms*, 43(1):1–16, 2002.

[ISO05] ISO/IEC 19501:2005. Unified modeling language (UML) version 1.4.2, 2005.

[KA06] T. Kamishima and S. Akaho. Efficient clustering for orders. In *Workshops Proceedings of the 6th IEEE International Conference on Data Mining (ICDM 2006)*, pages 274–278, 2006.

[KF03] T. Kamishima and J. Fujiki. Clustering orders. In *Proceedings of the 6th International Conference on Discovery Science*, pages 194–207, 2003.

[Koh82] T. Kohonen. Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43:59–69, 1982.

[KT06] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison Wesley, 2006.

[Llo82] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[MM00] H. Mannila and C. Meek. Global partial orders from sequential data. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 161–168, 2000.

[Mou91] H. Moulin. *Axioms of cooperative decision making*. Cambride Universiy Press, 1991.

[MT97] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Data Mining and Knowledge Discovery*, 1(3):241–258, 1997.

[MTV94] H. Mannila, H. Toivonen, and A. I. Verkamo. Improved methods for finding association rules. In *Proceedings of the Conference on Artificial Intelligence Research in Finland (STeP'94)*, pages 127–136, Turku, Finland, 1994.

[OVK$^+$08] M. Ojala, N. Vuokko, A. Kallio, N. Haiminen, and H. Mannila. Randomization of real-valued matrices for assessing the significance of data mining results. In *Proceedings of the Eighth SIAM International Conference on Data Mining*, pages 494–505, 2008.

[PFM06] K. Puolamäki, M. Fortelius, and H. Mannila. Seriation in paleontological data using Markov Chain Monte Carlo methods. *PLoS Computational Biology*, 2(2), 2006.

[QY04] M. Quist and G. Yona. Distributional scaling: An algorithm for structure preserving embedding for metric and nonmetric spaces. *Journal of Machine Learning Research*, 5:399–420, 2004.

[RS00] S. Roweis and L. Saul. Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290:2323–2326, 2000.

[RY06] D. Rajan and P. S. Yu. Discovering partial orders in binary data. In *Proceedings of the Sixth IEEE International Conference on Data Mining*, pages 510–521, 2006.

[Sha95] S. Sharma. *Applied Multivariate Techniques.* Wiley, 1995.

[SSW05] J. Schwartz, A. Steger, and A. Weissl. Fast algorithms for weighted bipartite matching. In *Proceedings of 4th International Workshop on Efficient and Experimental Algorithms*, pages 476–487, 2005.

[ST02] R. Shamir and D. Tsur. Improved algorithms for the random cluster graph model. In *Proceedings of Scandanavian Workshop on Algorithms Theory (SWAT) 2002*, pages 230– 239, 2002.

[TdL00] J. B. Tenenbaum, V. de Silva, and J. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, 2000.

[TK03] S. Theodoridis and K. Koutroumbas. *Pattern Recognition.* Elsevier Academic Press, 2nd edition, 2003.

[TT06] E. Terzi and P. Tsaparas. Efficient algorithms for sequence segmentation. In *Proceedings of the 2006 SIAM International Conference on Data Mining*, pages 316–327, 2006.

[Tuf01] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press USA, 2nd edition, 2001.

[UFM05] A. Ukkonen, M. Fortelius, and H. Mannila. Finding partial orders from unordered 0-1 data. In *Proceedings of the 11th ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 285–293, 2005.

[Ukk07] A. Ukkonen. Visualizing sets of partial rankings. In *Advances in Intelligent Data Analysis VII*, pages 240–251, 2007.

[UM07] A. Ukkonen and H. Mannila. Finding outlying items in sets of partial rankings. In *Knowledge Discovery in Databases: PKDD 2007*, pages 265–276, 2007.

[vBHL03] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard ai problems for security. In *Advances in Cryptology - EUROCRPYT 2003: International Conference on the Theory and Applications of Cryptographic Techniques*, pages 294–311, 2003.

[Ven07] J. Venna. *Dimensionality Reduction for Visual Exploration of Similarity Structures.* PhD thesis, Helsinki University of Technology, 2007.

[VK06] J. Venna and S. Kaski. Local multidimensional scaling. *Neural Networks*, 19(6–7):889–899, 2006.

[VK07] J. Venna and S. Kaski. Nonlinear dimensionality reduction as information retrieval. In *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, pages 568–575, 2007.

[WSZS07] K. Weinberger, F. Sha, Q. Zhu, and L. Saul. Graph regularization for maximum variance unfolding with an application to sensor localization. In *Advances in Neural Information Processing Systems 19*, 2007.

[ZH02] M. J. Zaki and C. J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining*, pages 457–473, 2002.

[Zha04] J. Zhang. Binary choice, subset choice, random utility, and ranking: A unified perspective using the permutahedron. *Journal of Mathematical Psychology*, 48(2):107–134, 2004.