Efficient Knowledge Extraction from Structured Data

Bianca Wackersreuther

München 2011

Efficient Knowledge Extraction from Structured Data

Bianca Wackersreuther

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Bianca Wackersreuther
aus Füssen

München, den 24.02.2011

Erstgutachter: Prof. Dr. Christian Böhm

Zweitgutachter: Prof. Dr. Thomas Seidl

Tag der mündlichen Prüfung: 15.12.2011

For my children Julius and Simon.

Contents

Acknowledgments

I would like to express my warmest gratitude to all the people who supported me during the past four years while I have been working on this thesis. I avail myself of the opportunity to thank them, even if I cannot mention all of their names here.

First of all, I would like to express my warmest and sincerest thanks to my supervisor, Professor Dr. Christian Böhm. The joy he has for his research was contagious and motivational for me, even during tough times. I warmly acknowledge Professor Dr. Thomas Seidl for his immediate willingness to act as a second referee for my thesis. I would also like to thank the other two members of my oral defense committee, Professor Dr. Claudia Linnhoff-Popien and Professor Dr. Rolf Hennicker, for their time and insightful questions.

This work could not have grown and matured without the discussions with my colleagues in our research group at LMU. In particular, I would like to give my thanks to Can Altinigneli, Jing Feng, Frank Fiedler, Katrin Haegler, Dr. Tong He, Xiao He, Bettina Konte, Annahita Oswald, Son Mai Thai, Nikola Müller, Dr. Claudia Plant, Michael Plavinski, Junming Shao, Peter Wackersreuther, Qinli Yang and Andrew Zherdin for their help, support, interesting hints, constructive and productive teamwork. My time at LMU was made enjoyable in large part due to my colleague Annahita Oswald that became a part of my life. Last, but not least, I had the pleasure to supervise and to work with several students who supported my work and who have been beneficial for this thesis. In particular, I would like to mention here Nadja Benaissa, Michael Dorn, Sebastian Goebl, Johannes Huber, Robert Noll, Michael Plavinski, Christian Richter and Dariya Sharonova.

Dr. Karsten Borgwardt was an awesome teacher of mine for the field of graph mining. It has been a unique chance for me to learn from his scientific experience and his never-ending enthusiasm for his research.

I have appreciated the database research group of the LMU. I am grateful to our chair secretary, Susanne Grienberger and our group's administrative assistant Franz Krojer who kept us organized and were always ready to help.

I was honored to be a mentee of the LMU Mentoring program. Professor Dr. Francesca Biagini always supported my work and has been a dedicated mentor to me.

Lastly, I would like to thank my family and my friends for all their love and encouragement, and most of all my loving, encouraging, and patient husband Peter whose faithful support during the final stages of this thesis is so appreciated. Thank you!

Bianca Wackersreuther
Munich, February 2011.

Abstract

Knowledge extraction from structured data aims for identifying valid, novel, potentially useful, and ultimately understandable patterns in the data. The core step of this process is the application of a data mining algorithm in order to produce an enumeration of particular patterns and relationships in large databases. Clustering is one of the major data mining tasks and aims at grouping the data objects into meaningful classes (clusters) such that the similarity of objects within clusters is maximized, and the similarity of objects from different clusters is minimized.

In this thesis, we advance the state-of-the-art data mining algorithms for analyzing structured data types. We describe the development of innovative solutions for hierarchical data mining. The EM-based hierarchical clustering method ITCH (Information-Theoretic Cluster Hierarchies) is designed to propose solid solutions for four different challenges. (1) to guide the hierarchical clustering algorithm to identify only meaningful and valid clusters. (2) to represent each cluster content in the hierarchy by an intuitive description with e.g. a probability density function. (3) to consistently handle outliers. (4) to avoid difficult parameter settings. ITCH is built on a hierarchical variant of the information-theoretic principle of Minimum Description Length (MDL). Interpreting the hierarchical cluster structure as a statistical model of the dataset, it can be used for effective data compression by Huffman coding. Thus, the achievable compression rate induces a natural objective function for clustering, which automatically satisfies all four above mentioned goals. The genetic-based hierarchical clustering algorithm GACH (Genetic Algorithm for finding Cluster Hierarchies) overcomes the problem of getting stuck in a local optimum by a beneficial combination of genetic algorithms, information theory and model-based clustering. Besides hierarchical data mining, we also made contributions to more complex data structures, namely objects that consist of mixed type attributes and skyline objects. The algorithm INTEGRATE performs integrative mining of heterogeneous data, which is one of the major challenges in the next decade, by a unified view on numerical and categorical information in clustering. Once more, supported by the MDL principle, INTEGRATE guarantees the usability on real world data. For skyline objects we developed SkyDist, a similarity measure for comparing different skyline objects, which is therefore a first step towards performing data mining on this kind of data structure. Applied in a recommender system, for example SkyDist can be used for pointing the user to alternative car types, exhibiting a similar price/mileage behavior like in his original query. For mining graph-structured data, we developed different approaches that have the ability to detect patterns in static as well as in dynamic networks. We confirmed the practical feasibility of our novel approaches on large real-world case studies ranging from medical brain data to biological yeast networks.

In the second part of this thesis, we focused on boosting the knowledge extraction process. We achieved this objective by an intelligent adoption of Graphics Processing Units (GPUs). The GPUs have evolved from simple devices for the display signal preparation into powerful coprocessors that do not only support typical computer graphics tasks but can also be used for general numeric and symbolic computations. As major advantage, GPUs provide extreme parallelism combined with a high bandwidth in memory transfer at low cost. In this thesis, we propose algorithms for computationally expensive data mining tasks like similarity search and different clustering paradigms which are designed for the highly parallel environment of a GPU, called CUDA-DClust and CUDA-k-means. We define a multi-dimensional index structure which is particularly suited to support similarity queries under the restricted programming model of a GPU. We demonstrate the superiority of our algorithms running on GPU over their conventional counterparts on CPU in terms of efficiency.

Zusammenfassung

Das Ziel der Wissensgewinnung aus strukturierten Daten ist es, gültige, bislang unbekannte, potentiell nützliche und letztendlich verständliche Muster in den Daten aufzudecken. Zentraler Schritt dieses Prozesses ist die Anwendung eines sog. *Data Mining* Algorithmus, der eine Aufzählung bestimmter Muster und Beziehungen in großen Datenbanken erzeugt. Eine wichtige Aufgabe des Data Minings stellt das sog. *Clustering* dar. Dabei sollen die Datenobjekte so in Gruppen (*Cluster*) zusammengefasst werden, dass die Ähnlichkeit aller Objekte innerhalb eines Clusters maximiert und die Ähnlichkeit der Objekte unterschiedlicher Cluster minimiert wird.

In dieser Arbeit erweitern wir aktuelle Data Mining Algorithmen dahingehend, dass damit strukturierte Datentypen untersucht werden können. Wir stellen innovative Lösungen vor, die eine Beschreibung der Daten durch hierarchische Strukturen erlauben. Die EM-basierte Clustering-Methode ITCH (Information-Theoretic Cluster Hierarchies) ist darauf ausgelegt, vier wichtige Zielsetzungen zu erfüllen. Erstens stellt sie sicher, dass der hierarchische Clustering-Algorithmus nur aussagekräftige und gültige Cluster erkennt. Zweitens stellt sie den Inhalt jedes Clusters der Hierarchie intuitiv dar, z.B. in Form einer Wahrscheinlichkeitsdichtefunktion. Drittens werden Ausreißer-Objekte konsistent verarbeitet, und viertens vermeidet sie das oft komplizierte Setzen von Parametern. ITCH baut auf eine hierarchische Variante des informationstheoretischen Prinzips *Minimum Description Length* (MDL) auf. Indem die hierarchische Clusterstruktur als statistisches Modell des Datensatzes interpretiert wird, kann es wirksam zur Datenkomprimierung im Sinne der *Huffman Codierung* eingesetzt werden. Daher stellt die theoretisch erreichbare Komprimierungsrate eine natürliche Zielfunktion für das Clustering-Problem dar, die automatisch alle vier zuvor genannten Zielsetzungen erfüllt. Der genetische hierarchische Clustering-Algorithmus GACH (Genetic Algorithm for finding Cluster Hierarchies) beseitigt das Problem, ein lediglich lokales Optimum als endgültige Lösung zu finden, durch eine geschickte Kombination von genetischen Algorithmen, Informationstheorie und Modell-basiertem Clustering. Neben dem hierarchischen Data Mining, widmeten wir uns auch komplexeren Datenstrukturen, nämlich Objekten, die durch Attribute unterschiedlichen Typs repräsentiert sind sowie Skyline-Objekten. Der Algorithmus INTEGRATE setzt die integrative Analyse heterogener Daten um, was sich als eine der Hauptaufgaben für die nächsten Jahrzehnte herausgestellt hat, indem er sowohl die numerische als auch die kategorische Information für den Clustering-Prozess vereint berücksichtigt. Auch in diesem Fall garantiert INTEGRATE, unterstützt durch das MDL Prinzip, den praktischen Einsatz für reelle Daten. Für Skyline-Objekte haben wir das Ähnlichkeitsmaß SkyDist entwickelt, welches den Vergleich unterschiedlicher Skyline-Objekte erlaubt und damit einen ersten Schritt in Richtung Data Mining auf dieser Art von strukturierten Daten darstellt. Eingesetzt in einer priorisierten Suchmaschine, kann SkyDist beispielsweise dazu verwendet werden, dem Nutzer unterschiedliche Autotypen vorzuschlagen, die entsprechend dem in der Anfrage spezifizierten Preis/Kilometerstand Profil ähnliche Eigenschaften aufweisen. Zur Analyse Graph-strukturierter Daten haben wir unterschiedliche Ansätze entwickelt, die sowohl in statischen als auch in dynamischen Netzwerken zum Auffinden von Mustern herangezogen werden können. Wir belegten die praktische Anwendbarkeit unserer neuen Ansätze an großen realen Fallstudien, die von medizinischen Gehirndaten bis hin zu biologischen Hefe-Netzwerken reichen.

Im zweiten Teil dieser Arbeit haben wir uns darauf konzentriert, den Prozess der Wissensgewinnung zu beschleunigen. Dies haben wir dadurch erreicht, dass wir uns den Einsatz von Grafikkarten oder auch *Graphics Processing Units* (GPUs) zunutze gemacht haben. GPUs haben sich von einfachen Bausteinen für die

Grafikverarbeitung in leistungsstarke Co-Prozessoren entwickelt, die sich nicht mehr nur für typische Grafikaufgaben eignen, sondern mittlerweile auch für allgemeine numerische und formale Berechnungen verwendet werden können. Der Hauptvorteil der GPUs liegt darin, dass sie extrem starke Parallelität und gleichzeitig hohe Bandbreite im Speichertransfer zu niedrigen Kosten bieten. In dieser Arbeit schlagen wir unterschiedliche Algorithmen für rechenintensive Data Mining Aufgaben, wie z.B. der Ähnlichkeitssuche und unterschiedliche Clustering-Paradigmen vor, die speziell auf die hoch parallele Arbeitsweise einer GPU ausgelegt sind, bezeichnet mit CUDA-DClust und CUDA-k-means. Wir definieren eine multidimensionale Indexstruktur, die so konzipiert wurde, dass sie Ähnlichkeitsanfragen auch unter dem eingeschränkten Programmierungsmodell der GPU unterstützt und zeigen die Überlegenheit unserer Algorithmen auf der GPU gegenüber ihrem Pendant in der CPU in Bezug auf Effizienz.

Chapter 1

Preliminaries

The amount of data being collected by high-throughput experimental technologies or high-speed internet connections far exceeds the ability to reduce and analyze data without the use of automated analysis techniques. Knowledge extraction, often also referred to as *Knowledge Discovery in Databases* (KDD), is a young sub-discipline of computer science aiming at the automatic interpretation of large datasets. The core step of this process is the application of a data mining algorithm in order to produce an enumeration of particular patterns and relationships in large databases. Over the last decade, a wealth of research articles on new data mining techniques has been published, and the field remains growing. Section 1.1 introduces first the main concepts of KDD. Afterwards *data mining*, the core step of the KDD process, is presented in Section 1.2 and the most prominent data mining methods are reviewed. Section 1.3 describes the data mining task *clustering* in more detail. In the following two sections, we specify how to support the clustering process by information theory and how to accelerate the data mining process by the idea of parallelization. Section 1.6 presents a detailed outline of this thesis.

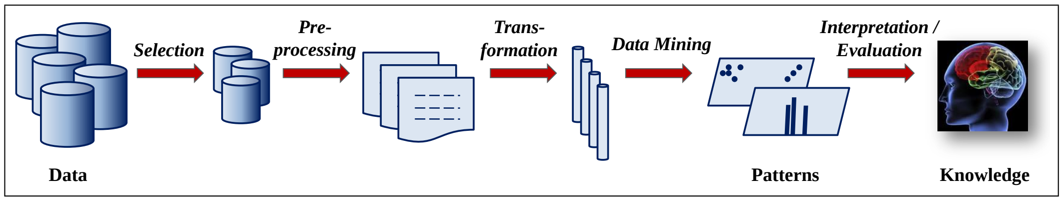


Figure 1.1: The KDD process.

1.1 The Classic Definition of KDD

The classic definition of KDD by Fayyad *et al.* from 1996 describes it as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [32]. The KDD process, as illustrated in Figure 1.1, consist of an iterative sequence of the following steps:

1. ***Selection***: Creating a target dataset by selecting a dataset or focusing on a subset of variables or data samples on which discovery is to be performed.

2. ***Preprocessing***: Performing data cleaning operations, such as removing noise or outliers if appropriate, handling missing data fields, accounting for time-sequence information, as well as deciding DBMS issues, such as data types, schema and mapping of missing and unknown values.

3. ***Transformation***: Finding useful features to represent the data depending on the goal of the task, e.g., using dimensionality reduction or transformation methods to reduce the number of attributes or to find invariant representations for the data.

4. ***Data Mining***: Searching for interesting patterns in a particular representational form or a set of such representations, including classification rules or trees, regression, clustering, sequence modeling, dependency and line analysis. Choosing the right data mining algorithm includes selecting the method(s) to be used to search for patterns in the data and matching a particular data mining method with the overall criteria of the KDD process.

5. ***Interpretation and Evaluation***: Applying visualization and knowledge representation techniques to the extracted patterns, and removing redundant or irrelevant patterns and translating the useful ones into terms understandable by users. The user may return to previous steps of the KDD process if the results are unsatisfactory.

1.2 Data Mining: The Core Step of Knowledge Extraction

Since data mining is the core step of the KDD process, the terms *KDD* and *Data Mining* are often used as synonyms. In [32], data mining is defined as a step in the KDD process which consists of applying data analysis algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data.

In accordance with [44], existing data mining algorithms include the following data mining methods. Classification and prediction, clustering, outlier analysis, characterization and discrimination, frequent itemset and association rule mining, and evolution analysis. Classification and prediction are called *supervised* data mining tasks, because the goal is to learn a model for predicting a predefined variable. Supervised data mining requires that class labels are given for all data objects. The class label of an object assigns it to one of several predefined classes. The other techniques are called *unsupervised*, because the user does not previously identify any of the variables to be learned. Instead, the algorithms have to automatically identify any interesting regularities and patterns in the data. Clustering means grouping the data objects into classes by maximizing the similarity between objects of the same class and minimizing the similarity between objects of different classes. We explain this important data mining task in detail in the following section. Outlier analysis deals with the identification of data objects that cannot be grouped in a given class or cluster, since they do not correspond to the general model of the data. Characterization and discrimination provide the summarization and comparison of general features of objects. Frequent itemset and association rule mining focuses on discovering rules showing attribute value conditions that occur frequently together in a given dataset. And evolution analysis models trends in time related data that evolve.

1.3 Clustering: One of the Major Data Mining Tasks

In this thesis, we mainly focus on clustering techniques w.r.t. the special challenges posed by complex structured data. The goal of clustering is to find a natural grouping of a dataset such that data objects assigned to a common group called *cluster* are as similar as possible and objects assigned to different clusters differ as much as possible. Consider for example the set of objects visualized in Figure 1.2. A natural grouping would assign the objects to two different clusters $blue$ and red. Two outliers not fitting well to any of the clusters should be left unassigned.

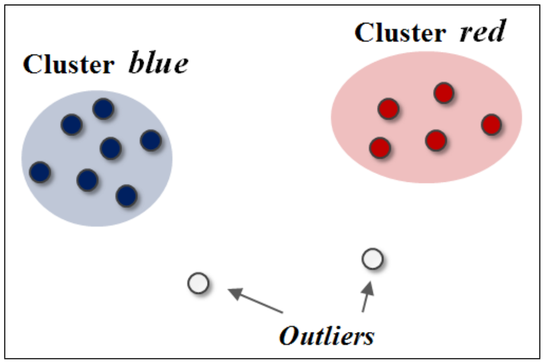


Figure 1.2: Example for clustering consisting of two clusters $blue$ and red and two outlier objects.

Cluster analysis has been used in a large variety of fields such as astronomy, physics, medicine, biology, archeology, geology, geography, psychology and marketing. Many different research areas contributed new approaches (namely pattern recognition, statistics, information retrieval, machine learning, bioinformatics and data mining). In some cases, the goal of cluster analysis is a better understanding

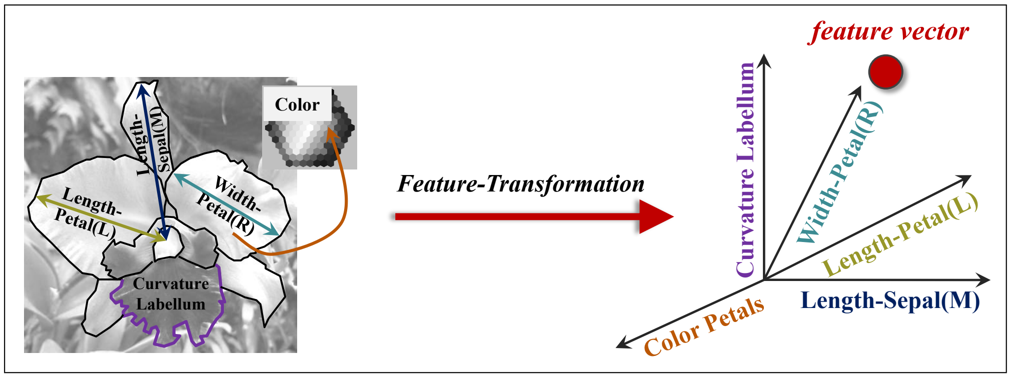


Figure 1.3: Example for feature transformation. Five features are measured and transformed into a 5-dimensional vector space.

of the data (e.g. learning the *natural* structure of data which should be reflected by a meaningful clustering). In other cases, cluster analysis is merely a first step for different purposes such as indexing or data compression. While clustering in general is a rather dignified problem, mainly in about the last decade new approaches have been proposed to cope with new challenges provided by modern capabilities of automatic data generation and acquisition in more and more applications producing a vast amount of high dimensional data. These data need to be analyzed by data mining methods in order to gain the full potentials from the gathered information. However, structured data pose different challenges for clustering algorithms that require specialized solutions. So, this area of research has been highly active in the recent years with an abundance of proposed algorithms.

Like most data mining algorithms, the definition of clustering requires specifying some notion of similarity among objects. In most cases, the similarity is expressed in a vector space, called the *feature space*. In Figure 1.2, we indicate the similarity among objects by representing each object by a vector in 2-dimensional space. Characterizing numerical properties (from a continuous space) are extracted from the objects, and taken together to a vector $x \in \mathbb{R}^d$ where d is the dimensionality of the space, and the number of properties which have been extracted, respectively. For instance, Figure 1.3 shows a feature transformation where the object is a certain kind of an orchid. The phenotype of orchids can be characterized using the lengths and widths of the two petal and the three sepal leaves, of the form (curvature) of the labellum, and of the colors of the different compartments. In this example, five features are measured, and each object is thus transformed into a 5-dimensional vector space. To detect the similarity among two data objects x and y represented as feature vectors, a metric distance function $dist$ is used. More specifically, let $dist$ be one of the L_p norms, which is defined as follows for an arbitrary $p \in \mathbb{N}$.

$$dist(\vec{x}, \vec{y}) = \sqrt[p]{\sum_{i=1}^{d} |x_i - y_i|^p}$$

Usually the Euclidean distance is used, i.e. $p = 2$. If we have more complex data types, we have to define a suited distance function.

For complex data objects, such as images, protein structures or text data, it is a non-trivial task to define domain specific distance functions. In Chapter 5, we introduce SkyDist, a similarity measure for skyline objects. Skyline objects represent a set of data objects that fit to a certain kind of query. The classic example is a user looking for cheap hotels which are close to the beach. Typically, it is unknown to the system how important the two conditions, $distance$ and $price$, are to the user. The skyline query returns all offers which may be of interest. This means, the skyline does not only contain the cheapest hotel and the hotel closest to the beach but also all hotels providing an outstanding combination of $price$ and $distance$, which makes them more attractive to the user than any other hotel in the database. The high expressiveness of skylines aims at performing data mining on such data structures.

1.4 Information Theory for Clustering

Most clustering approaches suffer from a common problem: To obtain a good result, the user needs expertise about the data to select suitable parameters, e.g.

the number of clusters, density thresholds or neighborhood sizes. In practice, the best way to cope with this problem often is to run the algorithm several times with different parameter settings. Thereby, suitable values for the parameters can be learned in a trial and error fashion. But it is obvious, that this is an extremely time consuming approach. And anyhow, it cannot guarantee that at least useful values for the parameters are obtained.

A beneficial solution for that problem is to relate clustering to data compression. Imagine you want to transfer a dataset consisting of feature vectors via an extremely expensive and small-banded communication channel from a sender to a receiver. Without clustering, each coordinate needs to be fully coded by transforming the numerical value into a bit string. But, if the data provides regularities, clustering can drastically reduce communication costs. For example a model-based clustering algorithm, like EM [27], can be applied to determine a statistical model for the dataset, which defines more or less likely areas of the data space. With this knowledge, the data can be compressed very effectively since only the deviations from the model need to be encoded which requires much less bits than the full coordinates. The model can be regarded as a codebook which allows the receiver to de-compress the data again. Following the idea of (optimal) Huffman coding, we assign few bits to points in areas of high probability and more bits to areas of low probability. The interesting observation is the following: the compression becomes the more effective, the better our statistical model fits to the data.

This basic idea, often referred as the *Minimum Description Length* Principle (MDL) [40] allows comparing different clustering results: Assume we have two different clusterings A and B of the same dataset. We can state that A is better than B if it allows compressing the dataset more effectively than B. Note that we have to consider the overall communication costs, meaning that we have to add the costs caused by the statistical model itself to the code length spent for the data. Thereby we achieve a natural balance between the complexity of the model and its fit to the data, and avoid the so-called *over-fitting effect*. Closely related ideas developed by different communities include the Bayesian Information Criterion (BIC), the Aikake Information Criterion (AIC) and the Information Bottleneck method (IB) [95].

In Chapter 3, we define a hierarchical variant of the MDL principle, called $hMDL$. This novel criterion is able to assess a complete cluster hierarchy. Chapter 4 proposes $iMDL$, a coding scheme for performing integrative data mining on data that consist of mixed type attributes.

1.5 Boosting the Data Mining Process

It has been shown that many data mining algorithms, including clustering, have another common problem besides the challenging parameterization. Most computations are extremely time consuming, particularly if they have to deal with large databases, which is a serious problem, as the amount of scientific data is approximately doubling every year [92]. Typical algorithms for learning complex statistical models from data are in quadratic or cubic complexity classes w.r.t. the number of objects and/or the dimensionality. To make these algorithms usable in a large and high dimensional database context is therefore an important goal.

One opportunity for boosting the data mining process is the intelligent adoption of graphics processing units (GPUs). GPUs have recently evolved from simple devices for the display signal preparation into powerful coprocessors supporting the CPU in various ways (cf. Figure 1.4). Graphics applications such as realistic 3D games are computationally demanding and require a large number of complex algebraic operations for each image update. Therefore, today's graphics hardware contains a large number of programmable processors, which are optimized to cope with this high workload in a highly parallel way. Vendors of graphics hardware have anticipated that trend and developed libraries, precompilers and application programming interfaces. Most prominently, NVIDIA's technology *Compute Unified Device Architecture* (CUDA) offers free development tools for the C programming language in which both the *host program* as well as the

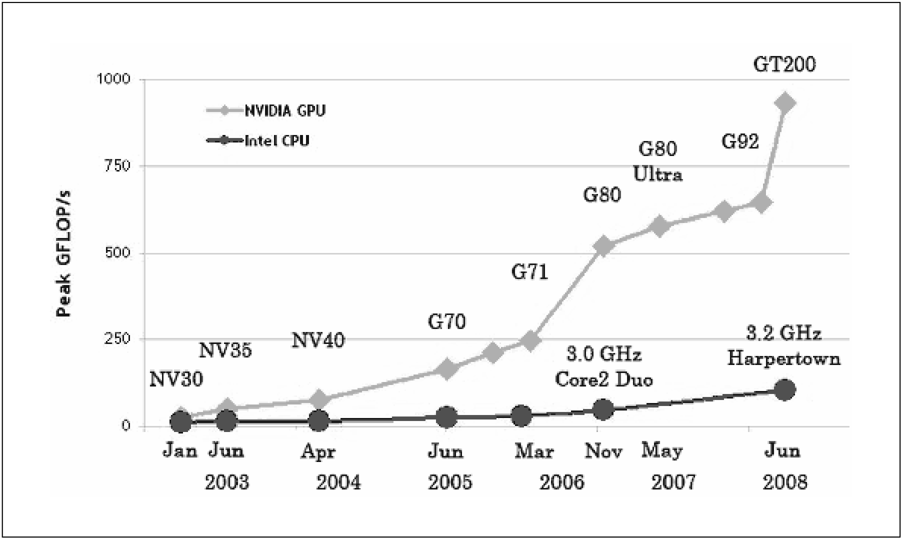


Figure 1.4: Growth trend of NVIDIA GPU vs. CPU (Source: NVIDIA).

kernel functions are assembled in a single program [1]. The host program or so-called *main program* is executed on the CPU. In contrast, the *kernel functions* are executed in a massively parallel fashion on hundreds of processors on the GPU. Analogous techniques are also offered by ATI using the brand names Close-to-Metal, Stream SDK, and Brook-GP.

In terms of peak performance, the GPU has outperformed state-of-the-art multi-core CPUs by a large margin. Hence, there is a great effort in many research communities such as life sciences [69, 73], mechanical simulation [94], cryptographic computing [8], machine learning [21], or even data mining [88] to use the computational capabilities of GPUs even for purposes which are not at all related to computer graphics. The corresponding research area is called General Processing-Graphics Processing Units (GP-GPU). We focus on exploiting the computational power of GPUs for data mining. Therefor specialized indexing methods are required because of the highly parallel but restricted programming environment. In this thesis, we propose a multi-dimensional indexing method that is well suited for the architecture provided by the GPU.

To demonstrate that highly complex data mining tasks can be efficiently implemented using novel parallel algorithms, we propose parallel versions of general similarity search and two widespread clustering algorithms in Chapter 7. These algorithms make use of the high computational power provided by the GPU. They are highly parallel and exploit thus the high number of simple SIMD (Single Instruction Multiple Data) processors of today's graphics hardware. The parallelization which is required for GPUs differs considerably from previous parallel algorithms which have focused on shared-nothing parallel architectures prevalent in server farms. For even further accelerations, we propose the indexed variants of these algorithms.

1.6 Outline of the Thesis

In this thesis, we aim at discovering novel data mining algorithms for mining different kinds of data structures. In addition, we focus on boosting opportunities for computationally intensive steps of the data mining process. The detailed outline is given in the following.

In Chapter 1, we give the reader a short introduction to the broader context of this thesis. In Chapter 2, we provide a brief overview of traditional data mining algorithms on structured data and present a rather general survey on advanced clustering methods. In addition, we introduce the broad related work on techniques for boosting the data mining process by an intelligent adoption of Graphics Processing Units (GPUs).

Chapter 3 to Chapter 6 deal with advanced data mining algorithms on structured data types. Here we focus on hierarchical data mining, the analysis of more complex data types like mixed type attribute data and skyline objects, and graph mining. In Chapter 3, we present the two hierarchical clustering algorithms ITCH (Information-Theoretic Cluster Hierarchies) and GACH (Genetic Algorithm for finding Cluster Hierarchies). ITCH, is an EM-based clustering approach to arrange only natural, valid, and meaningful clusters in a hierarchy guided by the idea of data compression. GACH overcomes the common problem of getting stuck in a local optimum by a beneficial combination of genetic algorithms, information theory and model-based clustering. The ideas on hierarchical data mining have already been published by the author [12]. In Chapter 4, we propose INTEGRATE [13], a new algorithm for performing integrative parameter-free clustering of data with mixed type attributes. Chapter 5 provides SkyDist [17], a novel similarity measure for the comparison of different skyline objects. Chapter 6 is dedicated to our novel approaches that have the ability to detect patterns in static as well as in dynamic networks. We published a large case-study on static brain network models [78, 99] and our work on finding frequent dynamic subgraphs [100].

Chapter 7 presents our work on data mining using GPUs. Section 7.1 explains the graphics hardware and the CUDA programming model underlying our procedures. In Section 7.2, we develop a multi-dimensional index structure for similarity queries on the GPU that can be used for further accelerations of the data mining process. Section 7.3 presents non-indexed as well as indexed algorithms to perform the similarity join on graphics hardware. Section 7.4 and Section 7.5 are dedicated to GPU-capable algorithms for density-based and partitioning clustering. The contributions on data mining using GPUs have already been published by the author in two publications [15, 14].

Chapter 8 summarizes and discusses the major contributions of this thesis. It includes some ideas for future directions.

Chapter 2

Related Work

To survey the large work on performing data mining of structured data, we mainly focus on well-known clustering techniques for hierarchical data in Section 2.1 as well as for mixed type attributes data in Section 2.2. In Section 2.3, we describe some ideas of how to enhance the effectiveness of many data mining concepts by information theory. In the field of graph mining, we present different algorithms for finding interesting patterns, i.e. frequent subgraphs in one large network or in a given set of networks (cf. Section 2.4). Finally, we conclude the related research on acceleration opportunities for the data mining process by the use of graphics processors.

2.1 Hierarchical Clustering

Hierarchical clustering algorithms compute a hierarchical decomposition of the dataset instead of a unique assignment of data objects to clusters in partitioning clustering. Given a dataset DS, the goal is to produce a hierarchy, which is visualized as a tree, called *dendrogram*. The nodes of the dendrogram represent subsets of DS simulating the structure found in DS with the following properties (cf. Figure 2.1):

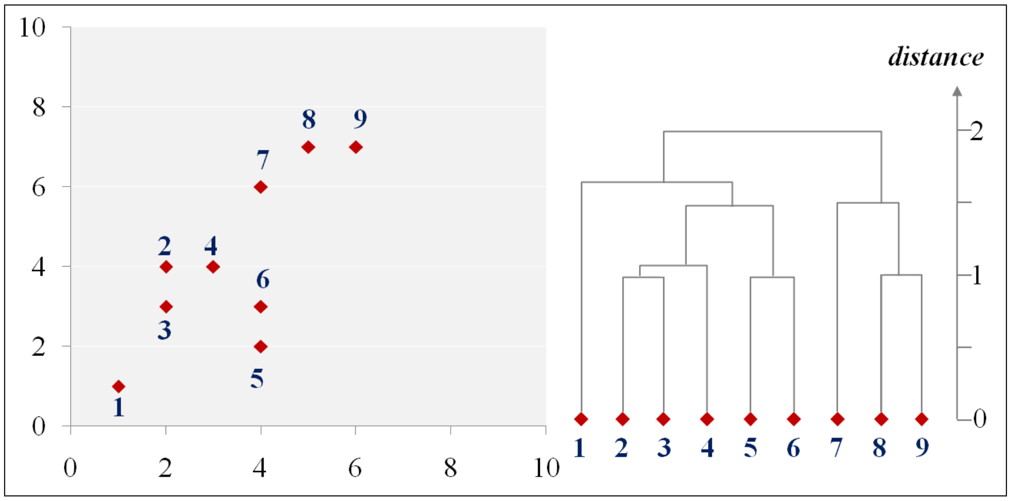


Figure 2.1: The hierarchical structure, present in the dataset on the left side, is visualized by the dendrogram on the right side.

- The root of the dendrogram represents the whole dataset DS.
- Each leaf of the dendrogram corresponds to one data object of DS.
- The internal nodes of the dendrogram are defined as the union of their children, i.e., each node represents a cluster containing all objects in the corresponding subtree.
- Each level of the dendrogram represents a partition of the dataset into distinct clusters.

Hierarchical clustering can be subdivided into *agglomerative* methods, which proceed by series of fusions of data objects into clusters, and *divisive* methods, which separate the data objects successively into finer clusters. Agglomerative hierarchical clustering algorithms follow a bottom-up strategy by merging the clusters iteratively. They start by placing each data object in its own single cluster. Then, the clusters are merged together into larger clusters by grouping similar data objects together until the entire dataset is encapsulated into one final root cluster.

Most hierarchical methods belong to this category. They differ only in their definition of between-cluster similarity.

Divisive hierarchical clustering works in the opposite way - it starts with all data objects in one root cluster and subdivides them into smaller clusters until each cluster consists of only one single data object. Divisive methods are not generally available, and have been rarely applied. The reason for this is mainly computational - divisive clustering is more computationally expensive when it comes to making decisions in dividing a cluster in two clusters given all possible choices. While in agglomerative procedures in one step two out of maximum n elements have to be chosen for merging, in divisive procedures fundamentally all subsets have to be analyzed so that divisive procedures have an algorithmic complexity of $O(2^n)$.

2.1.1 Agglomerative Hierarchical Clustering

Given a dataset DS of n objects, the basic process of agglomerative hierarchical clustering is defined as follows:

1. Place each data object $x_i \in DS$ $(i = 1, \dots n)$ in one single cluster C_i. Create the list of initial clusters $\mathbf{C} = C_1, \dots, C_n$, which will build the leaves of the resulting dendrogram.

2. Find the two clusters $C_i, C_j \in \mathbf{C}$ with the minimum distance to each other.

3. Merge the clusters C_i and C_j to create a new internal node C_{ij} which will be the parent of C_i and C_j in the resulting dendrogram. Remove C_i and C_j from $\mathbf{C}$.

4. Repeat step (2) and (3) until the total number of clusters in $\mathbf{C}$ becomes one.

In the first step of the algorithm, when each object represents its own cluster, the distances between the clusters are defined by the chosen distance function between the objects of the dataset. However, once several objects have been linked together, a linkage rule is needed to determine the actual distance between two clusters. There are numerous linkage rules that have been proposed. In the following, some of the most commonly used are presented.

2.1.2 Linkage methods

Let DS be a dataset, $dist$ denotes the distance function between the data objects of DS, and let C_i and C_j be two disjunct clusters consisting of objects of DS, i.e. $C_i, C_j \subseteq DS$ and $C_i \cap C_j = \emptyset$. In the following, some of the most commonly used linkage rules to determine the distance between two clusters are described.

Single Link

One of the most widespread approaches to agglomerative hierarchical clustering is Single Link [72]. Single Link defines the distance $dist_{SL}$ between any two clusters C_i and C_j as the minimum distance between them, i.e. as the distance between the two closest objects:

$$dist_{SL}(C_i, C_j) = \min_{x_i \in C_i, x_j \in C_j} \{dist(x_i, x_j)\}.$$

Using the Single Link method often causes the *chaining phenomenon*, also called *Single Link effect*, which is a direct consequence of the Single Link approach tending to build a chain of objects that connects two clusters.

Complete Link

The Complete Link method [25] is the opposite of Single Link. Complete Link defines the distance $dist_{CL}$ between any two clusters C_i and C_j as the maximum distance between them:

$$dist_{CL}(C_i, C_j) = \max_{x_i \in C_i, x_j \in C_j} \{dist(x_i, x_j)\}.$$

This method should not be used if there is a lot of noise expected to be present in the dataset. It also produces very compact clusters. This method is useful if one is expecting objects of the same cluster to be far apart in multi-dimensional space. In other words, outliers are given more weight in the cluster decision.

Average Link

Average Link [98] calculates the mean distance between all possible pairs of objects belonging to the two clusters C_i and C_j. Hence, it is computationally more expensive to compute the distance $dist_{AVG}$ than the aforementioned methods:

$$dist_{AVG}(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{x_i \in C_i, x_j \in C_j} \{dist(x_i, x_j)\}.$$

Average Link is sometimes also referred to as UPGMA (Unweighted Pair-Group Method using Arithmetic averages). There are several other variations of this method, e.g. Weighted Pair-Group Average, Unweighted Pair-Group Centroid, Weighted Pair-Group Centroid, but one should understand that it is a tradeoff between Single Link and Complete Link.

2.1.3 Density-based Hierarchical Clustering

The algorithm OPTICS (Ordering Points To Identify the Clustering Structure) [4] avoids the Single Link effect by requiring a minimum object density per cluster, i.e. $MinPts$ number of objects are within a hypersphere with radius ϵ. The main idea of OPTICS is to compute a complex hierarchical cluster structure, i.e. all possible clusterings with the parameter ϵ ranging from 0 to a given maximum value simultaneously during a single traversal of the dataset. The resulting cluster structure is visualized by a so-called *reachability plot* where the objects are plotted according to the sequence specified in the cluster ordering along the x-axis, and for each object, its corresponding distance along the y-axis. Figure 2.2 depicts the reachability plot based on the cluster ordering computed by OPTICS for a given

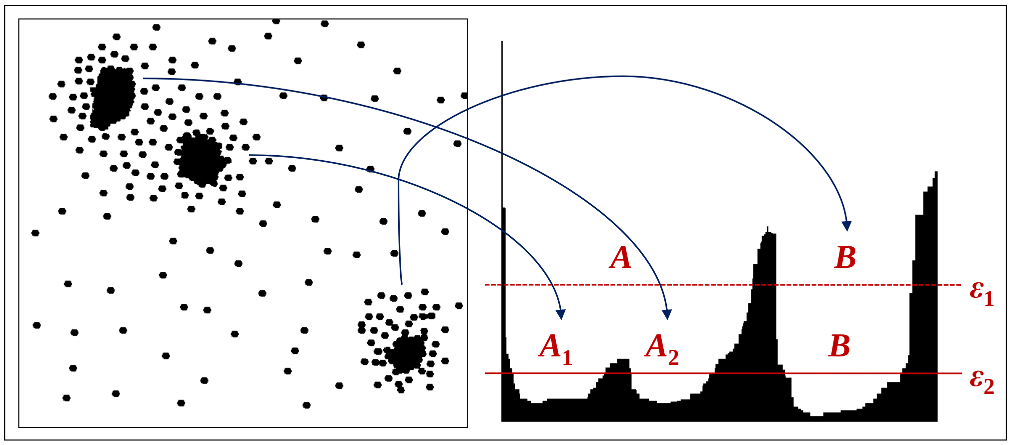


Figure 2.2: The reachability plot computed by OPTICS for a given sample 2-dimensional dataset.

sample 2-dimensional dataset. Valleys in this plot indicate clusters: objects having a small reachability value are closer and thus more similar to their predecessor objects than objects having a higher reachability value. The reachability plot generated by OPTICS can be cut at any level ϵ' parallel to the x-axis. It represents the partitioning according to the density threshold ϵ'. A consecutive subsequence of objects having a smaller reachability value than ϵ' belongs to the same cluster then. Referring to the given example in Figure 2.2, a cut at the level ϵ_1 results in two clusters A and B. Compared to this clustering, a cut at level ϵ_2 would yield three clusters. The cluster A is split into two smaller clusters denoted by A_1 and A_2 and cluster B decreased its size. This illustrates, how the hierarchical cluster structure of a dataset is revealed at a glance and can be easily explored by visual inspection.

Single Link is a special case of OPTICS (where $MinPts = 1$ and a maximum value of $\epsilon = \infty$). In this case, OPTICS has the same drawbacks like Single Link, i.e. missing stability w.r.t. noise objects and the Single Link effect. OPTICS overcomes this drawback if $MinPts$ is set to higher values.

2.1.4 Model-based Hierarchical Clustering

For many applications and further data mining steps, it is essential to have a model of the data. Hence, clustering with PDFs, which goes back to the popular EM algorithm [27], is a widespread approach for performing clustering. EM is a generalization of the k-means algorithm, a partitioning clustering algorithm for numerical data (cf. Section 2.2.1). After a suitable initialization, EM iteratively optimizes a mixture model of k Gaussian distributions until no further significant improvement of the log-likelihood of the data can be achieved. In detail, we generate a dataset that consists of first picking a centroid at random and then adding some noise. If the noise is normally distributed, this procedure will result in clusters of spherical shape. Model-based clustering assumes that the data were generated by a model and tries to recover the original model from the data. More precisely, the data has been generated from a finite mixture of k distributions, but that the cluster membership of each data point is not observed. In EM, the log-likelihood is used to calculate the model parameters θ:

$$L(\theta) = \log \prod_{i=1}^{n} P(x_i|\theta) = \sum_{i=1}^{n} \log P(x_i|\theta)$$

Model-based clustering aims for determine the parameter $\hat{\theta}$ that maximizes the log-likelihood:

$$L(\hat{\theta}) = \max_{\theta} L(\theta) = \max_{\theta} \sum_{i=1}^{n} \log P(x_i|\theta)$$

In the case of k clusters a vector $\vec{\theta} = \{\theta_1, \cdots, \theta_k\}$ has to be optimized:

$$L(\vec{\theta}) = \sum_{i=1}^{n} \log P(x_i|\vec{\theta}) = \sum_{i=1}^{n} \sum_{j=1}^{k} \log W_j + \log P(x_i|\theta_j)$$

,

where W_j is the weight of the cluster that represents the number of objects associated to that cluster.

EM can be applied to many different types of probabilistic modeling. In case the probability density function (PDF) which is associated to a cluster C is a multivariate Gaussian in a d-dimensional space which is defined by the parameters μ_C and Σ_C (where $\mu_C = (\mu_{C,1}, ..., \mu_{C,d})^\mathbf{T}$ is a vector from a d-dimensional space, called the location parameter, and Σ_C is a $d \times d$ covariance matrix), the definition of $P(x|\theta)$ is as follows:

$$P(x|\mu_C, \Sigma_C, x) = \frac{1}{\sqrt{(2\pi)^d \cdot |\Sigma_C|}} \cdot \mathbf{e}^{-\frac{1}{2}(x-\mu_C)^\mathbf{T} \cdot \Sigma_C^{-1} \cdot (x-\mu_C)}.$$

The EM algorithm is an iterative procedure where in each iteration step the mixture model of the k distributions are optimized until no further significant improvement of the log-likelihood of the data can be achieved. Usually, a very fast convergence of EM is observed. However, the algorithm may get stuck in local maximum of the log-likelihood. Moreover, the quality of the clustering result strongly depends on an appropriate choice of k, which is a non-trivial task in most applications.

A hierarchical extension of the EM algorithm was presented by Vasconcelos and Lippman in 1998 [97]. The efficiency of this approach is achieved by progressing the data in a bottom-up fashion, i.e. the mixture components of a given level are clustered to retrieve those components of the level above. This procedure requires only knowledge of the mixture parameters, the intermediate samples do not have to be resorted. With regards to practical applications, this algorithm leads to computationally efficient methods for estimating density hierarchies capable of describing data at different resolutions.

2.2 Mixed Type Attributes Data

A prominent characteristic of data mining is that it deals with very large and complex datasets. These data often contain millions of objects described by various types of attributes or variables, e.g. numerical, categorical, ratio or binary. Hence, integrative data mining algorithms have to be scalable and capable of dealing with different types of attributes. In terms of clustering, we are interested in algorithms which can efficiently cluster large datasets containing both numeric and categorical values because such datasets are frequently encountered in data mining applications. The traditional way to treat categorical attributes as numeric does not always produce meaningful results because many categorical domains are not ordered.

One simple idea for performing integrative clustering on heterogeneous data is to combine k-means based methods with the k-modes algorithm. The algorithm k-prototypes derives advantage from this combination, and is therefore one of the first approaches towards integrative clustering.

2.2.1 The k-means Algorithm for Clustering Numerical Data

The k-means algorithm [72] is one of the mostly used partitioning non-hierarchical clustering approach. The procedure follows a simple and easy way to group a given dataset DS consisting of n numeric objects into a certain number of k $(< n)$ clusters. First, k centroids are defined, one for each cluster. Then, the algorithm takes each point of DS and associates it to the nearest centroid, until no more point is pending. Afterwards, the k new centroids (the mean value μ over the coordinates of all data points belonging to the specific cluster) are recalculated, and thus, a new association has to be determined between the points of DS and the nearest new centroid. These two steps are performed until no more location changes of the centroids are observed. Finally, k-means aims at minimizing an objective function, in this case the within clusters sum of squared errors (WCSS).

$$WCSS = \sum_{j=1}^{k} \sum_{i=1}^{n} (dist(x_{i,j}, \mu_j))^2,$$

where $dist(x_{i,j}, \mu_j)$ is a chosen distance function between a data point $x_{i,j}$ and the centroid μ_j of cluster C_j, is an indicator of the distance of the n data points from their respective cluster centers.

The main advantage of k-means is its efficiency. It can be proven that convergence is reached in $O(n)$ iterations. However, this method has four major drawbacks: First, the number of clusters k has to be specified in advance, second, the cluster compactness measure WCSS and thus the clustering result is very sensitive to noise and outliers. In addition, k-means implicitly assumes a Gaussian data distribution, and is thus restricted to detect spherically compact clusters. The major drawback lies in its limited practicability to numeric data.

2.2.2 Conceptual Clustering Algorithms for Categorical Data

In principle the formulation of the WCSS in Section 2.2.1 is also valid for categorical and mixed type objects. The reason why the k-means algorithm cannot cluster categorical objects is that the calculation of the mean value is not defined for categorical data. These limitations can be removed by the following modifications:

- Use a simple matching distance function for categorical objects.
- Replace means as cluster representatives by modes.
- Use a frequency-based method to find the modes.

A Distance Function for Categorical Objects

Let x, y be two categorical objects described by m categorical attributes. The distance function between these two objects can be defined by the total mismatches of the corresponding attribute categories of the two objects. The smaller the number of mismatches is, the more similar x and y. This measure is often referred to as *simple matching* [56]. Formally this distance function is defined as follows:

$$dist_{Simple}(x,y) = \sum_{i=1}^{m} \delta(x_i, y_i), \qquad \text{where} \qquad \delta(x_i, y_i) = \begin{cases} 0, & \text{if } x_i = y_i, \\ 1, & \text{if } x_i \neq y_i. \end{cases}$$

Modes as Cluster Representatives

Consider a set of n categorical objects X described by categorical attributes, $A_1, A_2, \cdots, A_m$. The mode of this set is an array $q = [q_1, q_2, \cdots q_m]$ of length m that minimizes the following formula:

$$D(X, q) = \sum_{i=1}^{n} dist_{Simple}(X_i, q)$$

Calculation of the Modes

Let $n_{c_{k,i}}$ be the number of objects having the k-th category $c_{k,i}$ in attribute A_i, and let $p(A_i = c_{k,i}|X)$ be the relative frequency of category $c_{k,i}$ in X. Then the function $D(X, q)$ is minimized if and only if $p(A_i = q_i|X) \geq p(A_i = c_{k,i}|X)$ for $q_i \neq c_{k,i}$ for all $i \in \{1, \cdots, m\}$[48].

This theorem defines a way to find the mode q from a given set of categorical objects X, and therefore it is important because it allows the k-means paradigm to be used to cluster categorical data.

The Algorithm k-modes

Conceptual clustering algorithms, like k-modes [49] implement the idea of clustering categorical data by using $dist_{Simple}$ as distance function and by means of the aforementioned modifications of k-means clustering. The procedure of the algorithm k-modes can be summarized as follows:

1. Select k initial modes, one for each cluster.

2. Allocate an object to the cluster whose mode is the nearest to it according to the distance function $dist_{Simple}$; update the mode of the cluster after each allocation according to the theorem presented in Section 2.2.2.

3. After all objects have been allocated to clusters, retest the distance of objects against the current modes. If an object is found such that its nearest mode

belongs to another cluster rather than its current one, reallocate the object to that cluster and update the modes of both clusters.

4. Repeat step (3) until no object has changed clusters after a full cycle test of the whole dataset.

Like the k-means algorithm, the k-modes algorithm also produces locally optimal solutions that are dependent on the initial modes and the order of objects in the dataset. Its efficiency relies on good search strategies. For data mining problems, which often involve many concepts and very large object spaces, the concepts based search methods can become a potential handicap for these algorithms to deal with extremely large datasets.

2.2.3 The k-prototypes Algorithm for Integrative Clustering

One of the first algorithms for integrative clustering is k-prototypes [47]. The distance between two mixed type objects x and y, which are described by m attributes $A_1^n, A_2^n, \cdots, A_p^n, A_{p+1}^c, A_{p+2}^c, \cdots, A_m^c$ is measured by the following formula:

$$dist_{Mixed} = \sum_{i=1}^{p}(x_i - y_i)^2 + \gamma \sum_{i=p+1}^{m} \delta(x_i, y_i).$$

The first term is the squared Euclidean distance measure on the p numeric attributes A_i^n and the second term is the simple matching distance function on the $m-p$ categorical attributes A_i^c. The weight γ is used to avoid favoring either type of attribute. The influence of this parameter in the clustering process is discussed in the publication by Huang [47].

2.3 Information Theory in the Field of Clustering

Many clustering algorithms suffer from the problem that they require a difficult parameter setting. Hence, several advanced clustering approaches directly focus on parameter-free partitioning clustering and are based on the Akaike Information Criterion (AIC), the Bayesian Information Criterion (BIC) and Minimum Description Length (MDL) [40]. For these methods, the data itself is the only source of knowledge. Information theoretic arguments are applied for model selection during clustering, and these approaches involve a lossless compression of the data.

The work of Still and Bialek [91] provides important theoretical background by using information-theoretic arguments to relate the maximum number of clusters that can be detected by partitioning clustering with the size of the dataset. The algorithm X-Means [81] combines the k-means paradigm with BIC for parameter-free clustering. X-Means involves an efficient top-down splitting algorithm where intermediate results are obtained by bisecting k-means and are evaluated with BIC. However, only spherically Gaussian clusters can be detected. The algorithm G-means (Gaussian means) introduced in [43] has been designed for parameter-free correlation clustering. G-means follows a similar algorithmic paradigm as X-Means with top-down splitting and the application of bisecting k-means upon each split. However, the criterion to decide whether a cluster should be split up into two is based on a statistical test for Gaussianity. Splitting continues until the clusters are Gaussian, which implies of course, that non-Gaussian clusters can still not be detected. The algorithm PG-means (Projected Gaussian means) by Feng and Hamerly [33] is similar to G-means but learns models with increasing k with the EM algorithm. In each iteration, various 1-dimensional projections of the data and the model are tested for Gaussianity. Experiments demonstrate that PG-means is less prone to over-fitting than G-means.

It turns out that the underlying clustering algorithm and the choice of the similarity measure are already some kind of parameterization which implicitly comes with specific assumptions. The commonly used Euclidean distance e.g. assumes Gaussian data. In addition, the algorithms discussed so far are very sensitive w.r.t. noise objects or outliers. These problems are addressed by the recently proposed algorithm RIC (Robust Information-theoretic Clustering) by Böhm *et al.* [10]. This algorithm can be applied for post-processing an arbitrary imperfect initial

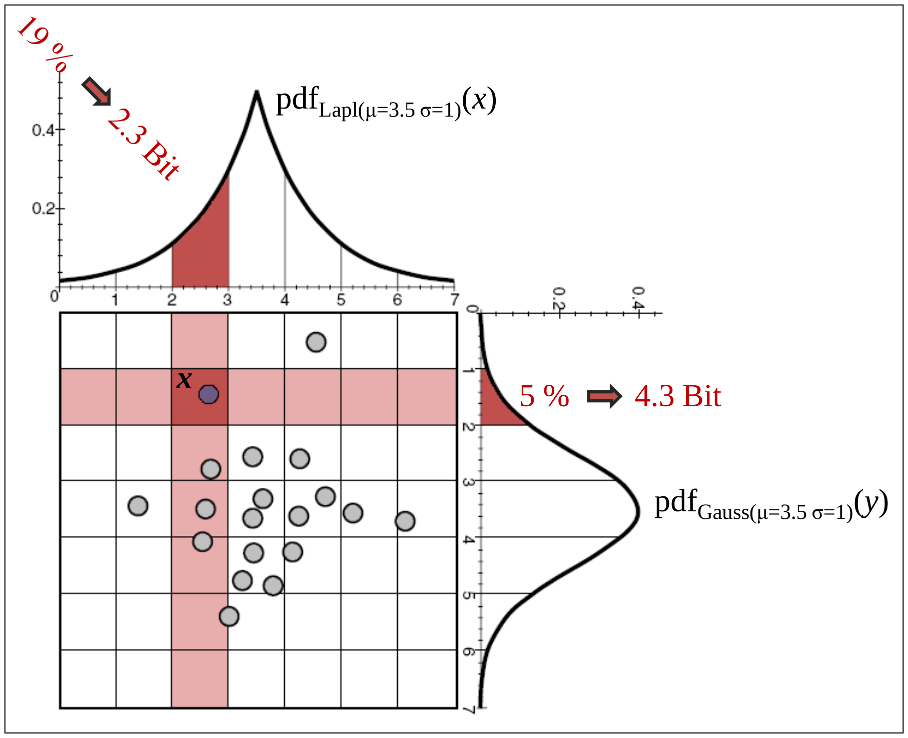

19 %
2.3 Bit
$\mathrm{pdf}_{\mathrm{Lapl}(\mu=3.5\ \sigma=1)}(x)$
5 %
4.3 Bit
$\mathrm{pdf}_{\mathrm{Gauss}(\mu=3.5\ \sigma=1)}(y)$
x

Figure 2.3: Coding scheme for cluster objects of the RIC algorithm. In addition to the data, type and parameters of the PDF need to be coded for each cluster.

clustering. It is based on MDL and introduces a coding scheme especially suitable for clustering together with algorithms for purifying the initial clusters from noise. In a first step, RIC removes noise objects from the initial clusters, and then merges clusters if this allows for more effective data compression. The algorithm operates with arbitrary data distributions, taken from a fixed set of PDFs.

The coding scheme for a data point p of a cluster C is illustrated in Figure 2.3. Here, we have a Laplacian distribution for the x-coordinate and a Gaussian distribution for the y-coordinate. Both distributions are assumed with $\mu = 3.5$ and $\sigma = 1$. We need to assign code patterns to the coordinate values such that coordinate values with a high probability (such as $3 < x < 4$) are assigned short patterns, and coordinate values with a low probability (such as $y = 12$ to give a more extreme example) are assigned longer patterns. Provided that a coordinate is really distributed according to the assumed distribution function, Huffman codes optimize the overall compression of the dataset. Huffman codes associate a bit string of length $l = log_2(1/p(p_i))$ to each coordinate p_i, where $p(p_i)$ is the probability of the (discretized) coordinate value.

The algorithm OCI (Outlier-robust Clustering using Independent Components) [11] provides parameter-free clustering of noisy data and allows detecting non-Gaussian clusters with non-orthogonal major directions. Technically this is achieved by defining a very general cluster notion based on the Exponential Power Distribution (EPD) and by integrating Independent Component Analysis (ICA) into clustering. The EPD includes a wide range of symmetric distribution functions, e.g. Gaussian, Laplacian and uniform distributions and an infinite number of hybrid types in between by an additional shape parameter. Beyond correlations detected by Principal Component Analysis (PCA), which correspond to correlation clusters with orthogonal major directions, ICA allows to detect general statistical dependencies in the data.

2.4 Graph-structured Data

In this section, we review the previous work on finding patterns in graph data and focus on three problems. First, frequent subgraphs across a dataset of graphs. Second, frequent subgraphs within one single large graph. Third, frequent subgraphs in dynamic graph data.

2.4.1 Graph Dataset Mining

The graph dataset mining approaches can be broadly divided into two classes, apriori-based and pattern-growth based. AGM (Apriori-based Graph Mining) [52] determines subgraphs s in a dataset DS of labeled graphs that occur in at least t percentage of all graphs (also called *transactions*) in DS. AGM uses a canonical

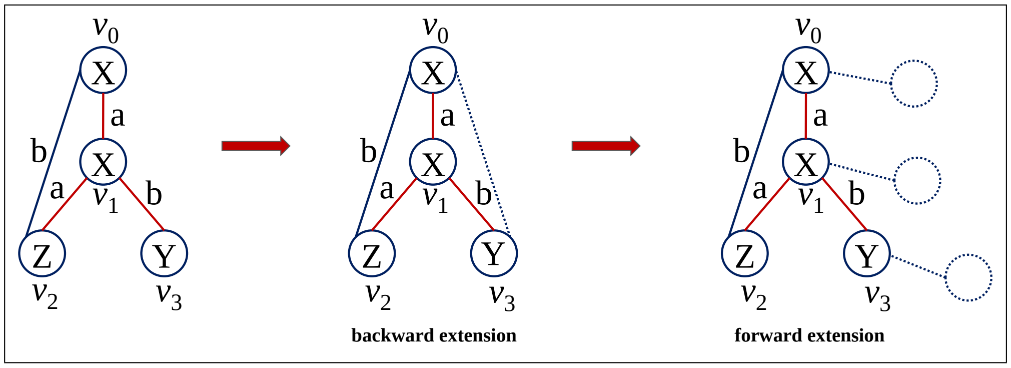


Figure 2.4: Illustration of the rightmost extension used in gSpan.

representation of subgraphs in order to reduce runtime costs for subgraph isomorphism checking.

Similar to AGM, FSG (Frequent SubGraph Discovery) [61] uses a canonical labeling based on the adjacency matrix. Canonical labeling, candidate generation and evaluation are sped up in FSG using graph invariants and the Transaction ID principle, which stores the ID of transactions a subgraph appeared in. This speed-up is paid for by reducing the class of subgraphs discovered to connected subgraphs, i.e. subgraphs where a path exists between all pairs of nodes.

The most well-known member of the class of pattern-growth algorithms, gSpan (graph-based Substructure pattern mining) [107], discovers frequent substructures efficiently without candidate generation. It is built on depth first search (DFS) tree. Given a graph S with nodes N and edges E, and a DFS tree TR, we call the starting node in TR, v_0, the root, and the last visited node, v_n, the *rightmost node*. The straight path from v_0 to v_n is called the *rightmost path*. Figure 2.4 shows an example. The red edges form a DFS tree. The nodes are discovered in the order v_0, v_1, v_2, v_3. The node v_3 is the rightmost node. The rightmost path is $v_0 \to v_1 \to v_3$.

The new method, called *rightmost extension*, restricts the extension of new edges in a graph as follows: Given a graph S and a DFS tree TR, a new edge e can be added between the rightmost node and other nodes on the rightmost path (*backward extension*) or it can introduce a new node and connect it to nodes on the rightmost path (*forward extension*). If we want to extend the graph in Figure 2.4, the backward extension candidate can be (v_3, v_0). The forward extension candidates can be edges extending from v_3, v_1 or v_0 with a new node inserted. Since there could be multiple DFS trees for one graph, gSpan establishes a set of rules to select one of them as representative so that the backward and forward extensions will only take place on one DFS tree.

Overall, new edges are only added to the nodes along the rightmost path. With this restricted extension, gSpan reduces the generation of the same graphs, and defines a canonical form of S. However, it still guarantees the completeness of enumerating all frequent subgraphs.

2.4.2 Large Graph Mining

Unlike graph dataset mining, large graph mining intends to find subgraphs that have at least t embeddings in *one* large graph. Note that any large graph mining algorithm can be applied to the graph dataset mining problem as well, simply by concatenation of all single graphs from a dataset into one large graph. Of course, subgraphs that include nodes from distinct single graphs must be pruned during pattern search. The reverse, applying graph dataset mining algorithms to large graphs, is not directly possible. GREW [63] and SUBDUE [23] are greedy heuristic approaches for frequent graph mining that deal speed for completeness of the solution. GREW iteratively joins frequent pairs of nodes (i.e. edges) into one supernode and determines disjoint embeddings of connected subgraphs by a maximal independent set algorithm. GREW maintains the location of the embeddings of the previously identified frequent subgraphs by rewriting the input graph, and therefore it is also applicable on very large graph datasets. This concept of edge contraction and graph rewriting is basically similar to that used by other heuristic algorithms, e.g. SUBDUE. However, GREW substantially extends this idea in multiple ways: First, it allows the concurrent contraction of different edge types, and second, GREW employs a set of heuristics in order to find longer and denser frequent patterns. Hence, this algorithm is enabled to simultaneously discover multiple patterns and to find longer patterns in fewer iterations. Finally, GREW uses a representation of the rewritten graph that retains all the information present in the original graph, and thus it precisely identifies whether or not there is an embedding of a particular subgraph. This result guarantees a lower bound on the frequency of each discovered pattern. SUBDUE tries to minimize the minimum description length (MDL) of a graph by compressing frequent subgraphs. Frequent subgraphs are replaced by one single node and the MDL of the remaining graph is then determined. Those subgraphs whose compression minimizes the MDL are considered frequent patterns in the input graph. The candidate graphs are generated starting from single nodes to subgraphs with several nodes, using a computationally constrained beam search.

Similarly, vSIGRAM and hSIGRAM [62] find subgraphs that are frequently embedded within a large sparse graph, using "**h**orizontal" breadth-first search and "**v**ertical" depth-first search, respectively. They employ efficient algorithms for candidate generation and candidate evaluation that exploit the sparseness of the graph.

While most of the techniques mentioned before often use some sort of heuristic search strategy that repeatedly compresses the graph to find frequent subgraphs, methods based on sampling subgraphs to estimate their frequency are predominant in application domains, like bioinformatics [55, 102]. Another strategy is to exhaustively enumerate all subgraphs. This has the advantage that one can then compute exact rather than approximate frequencies, but for large graphs, it is only feasible for subgraphs with a limited number k of nodes, typically $k \in \{3, 4\}$.

2.4.3 Dynamic Graph Mining

While the evolution of graphs over time has been addressed before, the corresponding studies predominantly dealt with topics such as densification and shrinking diameters of real-world graphs over time [65]. Only a few papers [18, 29, 64] define terminology for mining dynamic networks, but to the best of our knowledge no paper presents an efficient algorithm for detecting frequent subgraphs within dynamic graphs so far.

2.5 Boosting Data Mining

In this section, we survey the related research in general purpose computations using Graphics Processing Units (GPUs) with particular focus on database management and data mining.

Figure 2.5: The NVIDIA GeForce 8800 GTX graphics processor (Source: NVIDIA).

2.5.1 General Processing-Graphics Processing Units

Theoretically, GPUs are capable of performing any computation that can be transformed to the model of parallelism and that allow for the specific architecture of the GPU. This model has been exploited for multiple research areas. One scientific publication on computations from the field of life sciences has been published by Manavski and Valle [73]. The authors propose an extremely fast solution of the Smith-Waterman algorithm, a procedure for searching for similarities in protein and DNA databases, running on GPU and implemented in the CUDA programming environment. This algorithm launches a great number of parallel threads simultaneously to fully exploit the huge computational power of the GPU. It pursues the strategy that each GPU thread computes the whole alignment of the query sequence with one database sequence. All threads are grouped in a grid of blocks when running on the graphics card. In order to make the most efficient use of the GPU resources the computing time of all threads in the same grid must be as near as possible. For this reason, the sequences of the database are pre-ordered w.r.t. their length. Hence, all adjacent threads align the query sequence with two database queries having the nearest possible sizes.

Another widespread application area that uses the processing power of the GPU is mechanical simulation. One example is the work by Tascora *et al.* [94], that presents a novel method for solving large cone complementarity problems by means of a fixed-point iteration algorithm, in the context of simulating the frictional contact dynamics of large systems of rigid bodies. The proposed algorithm is well suited for running on parallel platforms that support the single instruction multiple data (SIMD) computational paradigm, which is ideally suited for handling problems with contacts in excess of hundreds of thousand.

To demonstrate the nearly boundless possibilities of performing computations on the GPU, we introduce one more example, namely cryptographic computing [8]. In this paper, the authors present a record-breaking performance for the Elliptic Curve Method (ECM) of integer factorization. This approach uses the parallelism of the GPU to handle several curves for a given auxiliary integer, which can thus be stored in the shared memory of a multiprocessor. All processors in one multiprocessor follow the same series of instructions which is a scalar multiplication on the respective curve modulo the same parameter m and with the same scalar s. The authors achieve an extreme speedup that results of extra hardware using the novel ECM implementation.

2.5.2 Database Management Using GPUs

Some research papers propose techniques to speed up relational database operations on GPU. Two recent papers [67, 16] address the topic of similarity join in feature space which determines all pairs of objects from two different sets R and S fulfilling a certain join predicate. The most common join predicate is the ϵ_{SJ}-join which determines all pairs of objects having a distance of less than a predefined threshold ϵ_{SJ}. The authors of [67] propose an algorithm based on the concept of space filling curves, e.g. the z-order, for pruning the search space, running on an NVIDIA GeForce 8800 GTX using the CUDA toolkit. The z-order of a set of objects can be determined very efficiently on GPU by highly parallelized sorting. Their algorithm operates on a set of z-lists of different granularity for efficient pruning. However, since all dimensions are treated equally, performance degrades in higher dimensions. An approach that overcomes that kind of problem is presented in [16]. Here, the authors parallelize the baseline technique underlying any join operation with an arbitrary join predicate, namely the nested loop join (NLJ), a powerful database primitive that can be used to support many applications including data mining.

Govindaraju *et al.* [38, 39] demonstrate that important building blocks for query processing in databases, e.g. sorting, conjunctive selections, aggregations, and semi-linear queries can be significantly speed up by the use of GPUs. Their algorithm GPUTeraSort uses the GPU as a co-processor to sort databases with billions of records. GPUTeraSort is general and can handle long records with wide keys. This hybrid sorting architecture offloads compute- and memory-intensive tasks to the GPU to achieve higher I/O performance and better main memory performance. A bitonic sorting network is mapped to GPU rasterization operations and uses the GPUs programmable hardware and high bandwidth memory interface. This novel data representation improves GPU cache efficiency and minimizes data transfers between the CPU and the GPU.

2.5.3 Data Mining Using GPUs

Finally, we survey some recent approaches for data mining using the GPU. In [20] a parallelized clustering approach for graphics hardware is presented. This algorithm extends the basic idea of k-means clustering by calculating the distances from a single input centroid to all objects at one time that can be done simultaneously in hardware. Thus the authors are able to exploit the high computational power and pipeline of GPUs, especially for core operations, like distance computations and comparisons. An additional efficient method that is designed to execute clustering on data streams confirms a wide practical field of clustering on GPU.

The algorithm k-means was also parallelized by Shalom *et al.* [87]. In their implementation multi-pass rendering and multi shader program constants are used. This is done by minimizing the use of GPU shader constants, which leads to a significant improvement of the performance as well as to a reduction of the data transactions between the CPU and the GPU. Handling data transfers between the necessary textures within the GPU is much more efficient than using shader constants. This is mainly due to the high memory bandwidth available in the GPU pipeline. Since all the steps of the k-means algorithm are able to be implemented in the GPU, the transferring of data back to the CPU during the iterations is avoided. The programmable capabilities of the GPU have been thus exploited to efficiently implement k-means clustering in the GPU.

Another approach implemented hierarchical agglomerative clustering based on complete linkage and the single linkage methods for the use in GPU [88]. The computations of distances and centroids, as well as the update operation of the similarity matrix is performed in parallel on the Graphics processor. To process the complete similarity matrix, $n \times n/2$ threads are needed, as this refers to the size of the matrix for a dataset consisting of n data objects. Compared to the conventional CPU implementation, this approach achieves a speedup factor of about 15 to 90.

Chapter 3

Hierarchical Data Mining

Since dendrograms and similar hierarchical representations provide extremely useful insights into the structure of a dataset, hierarchical clustering has become very popular in various scientific disciplines, such as molecular biology, medicine, or economy. However, well-known hierarchical clustering algorithms often either fail to detect the true clusters that are present in a dataset, or they identify invalid clusters, which are not existing in the dataset. These problems are particularly dominant in the presence of noise and outliers.

In Section 3.1, we propose ITCH, a novel EM-like clustering method that is built on a hierarchical variant of the information-theoretic principle of Minimum Description Length (MDL). The genetic-based clustering algorithm GACH, presented in Section 3.2 solves the complex hierarchical clustering problem by a beneficial combination of genetic algorithms, information theory and model-based clustering.

3.1 ITCH: An EM-based Hierarchical Clustering Algorithm

Many hierarchical clustering problems result in the questions "How can we decide if a given representation is really natural, valid, and therefore meaningful?" and "How can we enforce a hierarchical clustering algorithm to identify only the meaningful cluster structure?"

Information Theory for Clustering. We give the answer to these questions by relating the hierarchical clustering problem to that of information theory and data compression. Imagine you want to transfer the dataset via an extremely expensive and small-banded communication channel. Then you can interpret the hierarchy as a statistical model of the dataset, which defines more or less likely areas of the data space. This knowledge can be used for an efficient compression of the dataset. Following the idea of (optimal) Huffman coding, we assign few bits to points in areas of high probability and more bits to areas of low probability. The interesting observation is the following: the compression becomes the more effective, the better our statistical model fits to the data. This so-called *Minimum Description Length* (MDL) principle has recently received increasing attention in the context of partitioning (i.e. non-hierarchical) clustering methods. Note that it can not only be used to assess and compare the quality of the clusters found by different algorithms and/or varying parameter settings. Rather, we use this concept as an objective function to implement clustering algorithms directly using simple but efficient optimization heuristics.

We extend the idea of MDL to the hierarchical case and develop $hMDL$ for *hierarchical* cluster structures. Whereas previous MDL approaches can only evaluate the result of partitioning clustering methods, $hMDL$ is able to assess a complete cluster hierarchy. Moreover, it can be used in combination with an optimization heuristic to determine exactly that cluster hierarchy, which optimizes the data compression according to our $hMDL$ criterion.

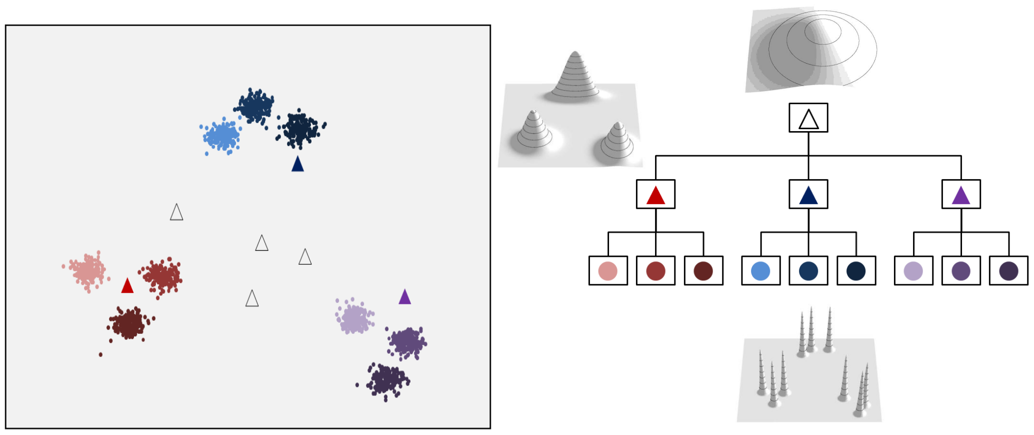


Figure 3.1: ITCH gives the hierarchical representation of a complex dataset consisting of three clusters, where each cluster contains three subclusters. Outliers are placed on different hierarchy levels and represented by the symbol Δ in the corresponding color. Besides the hierarchical structure, ITCH provides a model of each cluster content in form of a Gaussian PDF.

Challenges and Contributions. With an assessment condition for cluster hierarchies, we develop a complete hierarchical clustering approach on top of the idea of $hMDL$. This proposed algorithm ITCH (**I**nformation-**T**heoretic **C**luster **H**ierarchies) yields numerous advantages, out of which we demonstrate the following four:

1. All single clusters as well as their hierarchical arrangement are guaranteed to be **meaningful**. Nodes only are placed in the cluster hierarchy if they improve the data compression. This is achieved by optimizing the $hMDL$ criterion. Moreover, a maximal consistency with partitioning clustering methods is obtained.

2. Each cluster is represented by an **intuitive description** of its content in form of a Gaussian probability density function (PDF). Figure 3.1 presents an example of a 3-stage hierarchy. The output of conventional methods is often just the (hierarchical) cluster structure and the assignment of points.

3. ITCH has a consistent **outlier-handling**. Outliers are assigned to the root of the cluster hierarchy or to an appropriate inner node, depending on the degree of outlierness. For example, in Figures 3.1 the outlier w.r.t. the three red clusters at the bottom level is assigned to the parent cluster in the hierarchy, marked by a red triangle.

4. ITCH is **fully automatic** as no difficult parameter settings are necessary.

3.1.1 Information-theoretic Hierarchical Clustering

The clustering problem is highly related to that of data compression: The detected cluster structure can be interpreted as a PDF $f_\Theta(x)$ where $\Theta = \{\theta_1, \theta_2, ...\}$ is a set of parameters, and the PDF can be used for an efficient compression of the dataset. It is well-known that the compression by *Huffman coding* is optimal if the data distribution really corresponds to $f_\Theta(x)$. Huffman coding represents every point x by a number of bits which is equal to the negative binary logarithm of the PDF:

$$C_{\text{data}}(x) = -\log_2(f_\Theta(x)).$$

The better the point set corresponds to $f_\Theta(x)$, the smaller the coding costs $C_{\text{data}}(x)$ are. Hence, $C_{\text{data}}(x)$ can be used as the objective function in an optimization algorithm. However, in data compression, Θ serves as a *code book* which is required to decode the compressed dataset again. Therefore, we need to complement the compressed dataset with a coding of this code book, the parameter set Θ. When, for instance, a Gaussian Mixture Model (GMM) is applied, Θ corresponds to the weights, the mean vectors and the variances of the single Gaussian functions in the GMM. Considering Θ in the coding costs is also important for the clustering problem, because neglecting it leads to overfitting. For partitioning (non-hierarchical) clustering structures, several approaches have been proposed for the coding of Θ [81, 90]. These approaches differ from each other because there is no unambiguous and natural choice for a distribution function, which can be used for the Huffman coding of Θ, and different assumptions lead to different objective functions. In case of the hierarchical cluster structure in ITCH, a very natural distribution function for Θ exists: With the only exception of the root node, every node in the hierarchy has a parent node. This parent node is associated with a PDF which can naturally be used as a code book for the mean vector (and indirectly also for the variances) of the child node. The coding costs of the root node, however, are not important, because every valid hierarchy has exactly one root node with a constant number of parameters, and therefore, the coding costs of the root node are always constant.

Hierarchical Cluster Structure

In this Section, we formally introduce the notion of a hierarchical cluster structure (HCS). A HCS contains clusters $\{A, B, ...\}$ each of which is represented by a Gaussian distribution function. These clusters are arranged in a tree:

Definition 1 (Hierarchical Cluster Structure)

(1) A ***H****ierarchical* ***C****luster* ***S****tructure* ***HCS*** *is a tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ consisting of a set of nodes $\mathcal{N} = \{A, B, ...\}$ and a set of directed edges $\mathcal{E} = \{e_1, e_2, ...\}$ where A is a parent of B (B is a child of A) iff $(A, B) \in \mathcal{E}$. Every node $C \in \mathcal{N}$ is associated with a weight W_C and a Gaussian PDF defined by the parameters μ_C and Σ_C such that the sum of the weights equals one:*

$$\sum_{C \in \mathcal{N}} W_C = 1.$$

(2) If a path from A to B exists in $\mathcal{T}$ (or $A = B$) we call A an ancestor of B (B a descendant of A) and write $B \sqsubseteq A$.

(3) The level l_C of a node C is the height of the descendant subtree. If C is a leaf, then C has level $l_C = 0$. The root has the highest level (length of the longest path to a leaf).

The PDF which is associated with a cluster C is a multivariate Gaussian in a d-dimensional data space which is defined by the parameters μ_C and Σ_C (where $\mu_C = (\mu_{C,1}, ..., \mu_{C,d})^{\mathbf{T}}$ is a vector from a d-dimensional space, called the location parameter, and Σ_C is a $d \times d$ covariance matrix) by the following formula:

$$N(\mu_C, \Sigma_C, x) = \frac{1}{\sqrt{(2\pi)^d \cdot |\Sigma_C|}} \cdot \mathbf{e}^{-\frac{1}{2}(x-\mu_C)^{\mathbf{T}} \cdot \Sigma_C^{-1} \cdot (x-\mu_C)}.$$

For simplicity we restrict $\Sigma_C = \mathrm{diag}(\sigma^2_{C,1}, ..., \sigma^2_{C,d})$ to be diagonal such that the multivariate Gaussian can also be expressed by the following product:

$$\begin{aligned} N(\mu_C, \Sigma_C, x) &= \prod_{1 \le i \le d} N(\mu_{C,i}, \sigma^2_{C,i}, x_i) \\ &= \prod_{1 \le i \le d} \frac{1}{\sqrt{2\pi\sigma^2_{C,i}}} \cdot \mathbf{e}^{-\frac{(x_i-\mu_i)^2}{2\sigma^2_{C,i}}}. \end{aligned}$$

Since we require the sum of all weights in a hierarchical cluster structure HCS to be 1, a HCS always defines a function whose integral is ≤ 1. Therefore, the HCS can be interpreted as a complex, multimodal, and multivariate PDF, defined by the mixture of the Gaussians of the HCS $\mathcal{T} = (\mathcal{N}, \mathcal{E})$:

$$f_{\mathcal{T}}(x) = \max_{C \in \mathcal{N}} \{W_C \, N(\mu_C, \Sigma_C, x)\} \text{ with} \int_{\mathbb{R}^d} f_{\mathcal{T}}(x) \mathbf{d}x \le 1.$$

If the Gaussians of the HCS do not overlap much, then the integral becomes close to 1.

The operations, described in Section 3.1.1, assign each point $x \in DB$ to a cluster of the HCS $\mathcal{T} = (\mathcal{N}, \mathcal{E})$. We distinguish between the *direct* and the *indirect* association. A point is directly associated with that cluster $C \in \mathcal{N}$ the probability density of which is maximal at the position of x, and we write $C = Cl(x)$ and also $x \in C$, with:

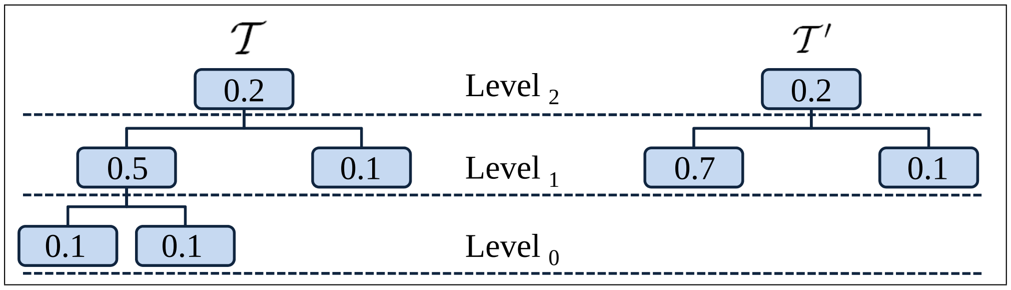


Figure 3.2: Hierarchical cut $\mathcal{T}'$ of the HCS $\mathcal{T}$ at level 1. All weights are placed inside the corresponding nodes.

$$Cl(x) = \arg\max_{C \in \mathcal{N}} \{W_C \cdot N(\mu_C, \Sigma_C, x)\} .$$

As we have already stated in the introduction, one of the main motivations of our hierarchical, information-theoretic clustering method ITCH is to represent a sequence of clustering structures which range from a very coarse (unimodal) view to the data distribution to a very detailed (multimodal) one, and that all these views are meaningful and represent an individual complex PDF. The ability to cut a cluster hierarchy at a given level L is obtained by the following definition:

Definition 2 (Hierarchical Cut) *A HCS $\mathcal{T}' = (\mathcal{N}', \mathcal{E}')$ is a* ***hierarchical cut*** *of a HCS $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ at level L, if the following properties hold:*
(1) $\mathcal{N}' = \{A \in \mathcal{N} | l_A \geq L\}$,
(2) $\mathcal{E}' = \{(A, B) \in \mathcal{E} | l_A > l_B \geq L\}$,
(3) For each $A \in \mathcal{N}'$ the following properties hold:

$$W'_A = \begin{cases} W_A & \textit{if } l_A > L \\ \sum_{B \in \mathcal{N}, B \sqsubseteq A} W_B & \textit{otherwise,} \end{cases}$$

where W_C and W'_C is the weight of node C in $\mathcal{T}$ and $\mathcal{T}'$, respectively.

(4) Analogously, for the direct association of points to clusters the following property holds: Let x be associated with Cluster B in $\mathcal{T}$, i.e. $Cl(x) = B$. Then in $\mathcal{T}'$, x is associated with:

$$Cl'(x) = \begin{cases} B & \textit{if } l_B \geq L \\ A | B \sqsubseteq A \wedge l_A = L & \textit{otherwise.} \end{cases}$$

Here, the weights of the pruned nodes are automatically added to the leaf nodes of the new cluster structure, which used to be the ancestors of the pruned nodes. Therefore, the sum of all weights is maintained (and still equals 1), and the obtained tree is again a HCS according to Definition 1. The same holds for the point-to-cluster assignments. Figure 3.2 presents an illustrative example of the hierarchical cut.

Generalization of the MDL Principle

Now we explain how the MDL principle can be generalized for hierarchical clustering and develop the new objective function $hMDL$. Following the traditional MDL principle, we compress the data points according to their negative log likelihood corresponding to the PDF which is given by the cluster hierarchy. In addition, we penalize the model complexity by adding the code length of the HCS parameters to the negative log likelihood of the data. The better the PDFs of child nodes fit into the PDFs of the parent, the less the coding costs will be. Therefore, the overall coding costs corresponds to the natural, intuitive notion of a good hierarchical representation of data by distribution functions. The discrete assignment of points to clusters allows us to determine the coding costs of the points clusterwise and dimensionwise, as explained in the following: The coding costs of the points associated with the clusters $C \in \mathcal{N}$ of the HCS $\mathcal{T} = (\mathcal{N}, \mathcal{E})$ corresponds to:

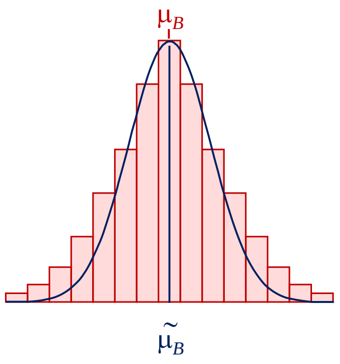
$$C_{\text{data}} = -\log_2 \prod_{x \in DB} \max_{C \in \mathcal{N}} \left\{ W_C \prod_{1 \leq j \leq d} N(\mu_{C,j}, \sigma^2_{C,j}, x_j) \right\}.$$

Since every point x is associated with that cluster C in the hierarchy which has maximum probability density, we can rearrange the terms of the above formula and combine the costs of all points that are in the same cluster:

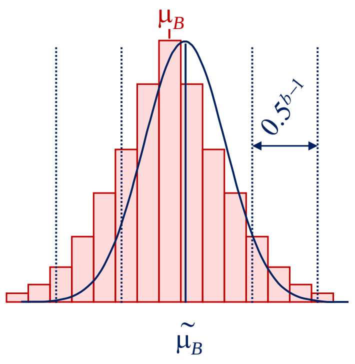
$$\begin{aligned}
&= -\sum_{x \in DB} \log_2 \Bigg(W_{Cl(x)} \prod_{1 \le j \le d} N(\mu_{Cl(x),j}, \sigma^2_{Cl(x),j}, x_j) \Bigg) \\
&= -\Bigg(\Bigg(\sum_{C \in \mathcal{N}} nW_C \log_2 W_C \Bigg) + \Bigg(\sum_{x \in DB;} \sum_{1 \le j \le d} \log_2 N(\mu_{Cl(x),j}, \sigma^2_{Cl(x),j}, x_j) \Bigg) \Bigg) \\
&= -\sum_{C \in \mathcal{N}} \Bigg(nW_C \log_2 W_C + \sum_{x \in C;} \sum_{1 \le j \le d} \log_2 N(\mu_{C,j}, \sigma^2_{C,j}, x_j) \Bigg),
\end{aligned}$$

where $W_{Cl(x)}$ is the weight of the cluster in the hierarchy which has maximum probability density for the point x. The ability to model the coding costs of each cluster separately allows us now, to focus on a single cluster, and even on a single dimension of a single cluster. A common interpretation of the term $-nW_C \log_2 W_C$, which actually comes from the weight a single Gaussian contributes to the GMM, is a Huffman coding of the cluster ID. We assume that every point carries the information which cluster it belongs to, and a cluster with many points gets a shortly coded cluster ID. These costs are referred to the *ID cost* of a cluster C.

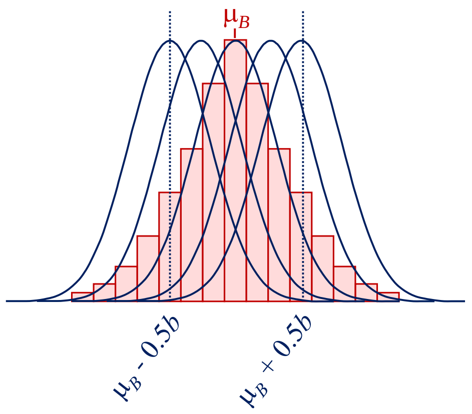
Consider two clusters, A and B, where $B \sqsubseteq A$. We now want to derive the coding scheme for the cluster B and its associated points. Several points are associated with B, where the overall weight of assignment sums up to W_B. When coding the parameters of the associated PDF of B, i.e. μ_B, and σ_B, we have to consider two aspects: (1) The *precision* both parameters should be coded to minimize the overall description length depends on W_B, as well as on σ_B. For instance, if only few points are associated with cluster B and/or the variance σ_B is very large, then it is not necessary to know the position of μ_B very precisely and vice versa. (2) The knowledge of the PDF of cluster A can can be exploited for the coding of μ_B, because for likely positions (according to the PDF of A) we



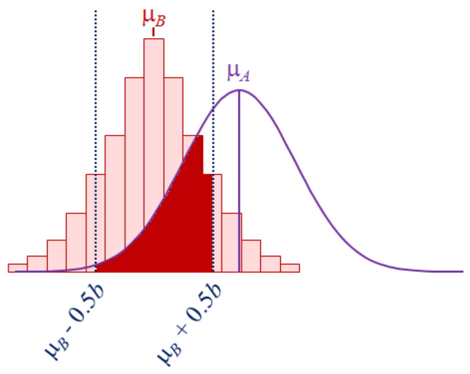
(a) Exact coding of the location parameter μ_B. $\widetilde{\mu}_B$ of the coding Gaussian function fits exactly to μ_B of the data distribution.



(b) Inexact coding of the location parameter μ_B by some regular quantization grid.



(c) Different Gaussians w.r.t. different grid positions.



(d) Complete interval of all possible values for the recovered location parameter μ_B w.r.t. to the PDF of the predecessor cluster A.

Figure 3.3: Optimization of coding scheme for the location parameter.

can assign fewer bits. Basically, model selection criteria, such as the Bayesian Information Criterion (BIC) or the Aikake Information Criterion (AIC) already address the first aspect, but not the hierarchical aspect. To make this thesis self contained, we consider both aspects. In contrast to BIC, which uses the natural logarithm, we use the binary logarithm to represent the code length in *bits*. For simplicity, we assume that our PDF is univariate and the only parameter is its mean value μ_B. We neglect σ_B by assuming e.g. some fixed value for all clusters. We drop these assumptions at the end of this section. When the true PDF of cluster B is coded inexactly by some parameter $\widetilde{\mu}_B$, the coding costs for each point x (which truly belongs to the distribution $N(\mu_B, \sigma_B^2, x)$) in B is increased compared to the *exact* coding of μ_B, which would result in c_{ex} bits per point:

$$c_{\text{ex}} = \int_{-\infty}^{+\infty} -\log_2(N(\mu_B, \sigma_B^2, x)) \cdot N(\mu_B, \sigma_B^2, x) \; \mathbf{d}x = \log_2(\sigma_B \sqrt{2\pi \cdot \mathbf{e}}).$$

If $\widetilde{\mu}_B$ instead of μ_B is applied for compression, we obtain:

$$c(\widetilde{\mu}_B, \mu_B) = \int_{-\infty}^{+\infty} -\log_2(N(\widetilde{\mu}_B, \sigma_B^2, x)) \cdot N(\mu_B, \sigma_B^2, x)\, \mathbf{d}x.$$

The difference is visualized in Figure 3.3(a) and 3.3(b) respectively: In 3.3(a) $\widetilde{\mu}_B$ of the coding PDF, depicted by the Gaussian function, fits exactly to μ_B of the data distribution, represented by the histogram. This causes minimum code lengths for the compressed points but also a considerable effort for the coding of μ_B. In Figure 3.3(b) μ_B is coded by some regular quantization grid. Thereby, the costs for the cluster points slightly increase, but the costs for the location parameter decreases. The difference between $\widetilde{\mu}_B$ and μ_B depends on the bit resolution and on the position of the quantization grid. One example is depicted in Figure 3.3(b) by five vertical lines, the Gaussian curve is centered by the vertical line closest to μ_B. We derive lower and upper limits of $\widetilde{\mu}_B \in [\mu_B - 1/2^b ... \mu_B + 1/2^b]$ from the number of bits b, spent for coding $\widetilde{\mu}_B$. The real difference between μ_B and $\widetilde{\mu}_B$ depends again on the grid position. Not to prefer clusters that are incidentally aligned with the grid cells, we average over *all* possible positions of the discretization grid. Figure 3.3(c) presents five different examples of the infinitely many Gaussians that could be recovered w.r.t. different grid positions. Note that all positions inside the given interval have equal probability. Hence, the average coding costs for every possible position of $\widetilde{\mu}_B$ can be expressed by the following integral:

$$\begin{aligned} c_{\text{appx}}(b) &= 2^{b-1} \int_{\mu_B - 1/2^b}^{\mu_B + 1/2^b} c(\widetilde{\mu}_B, \mu_B)\, \mathbf{d}\widetilde{\mu}_B \\ &= \frac{1}{2}\log_2(\pi \cdot \mathbf{e} \cdot \sigma_B^2) + \frac{1}{2} + \frac{\log_2 \mathbf{e}}{6 \cdot \sigma_B^2} \cdot 4^{-b}. \end{aligned}$$

Coding all $n \cdot W_B$ coordinates of the cluster points as well as the parameter μ_B (neglecting the ID cost) requires then the following number of bits:

$$C_{\text{appx}}(B) = c_{\text{appx}}(b) \cdot n \cdot W_B + b.$$

The optimal number b_{opt} of bits is determined by setting the derivation of the above term to zero.

$$\frac{\mathbf{d}}{\mathbf{d}b} C_{\text{appx}}(B) = 1 - \frac{4^{-b} \cdot n \cdot W_B}{3 \cdot \sigma_B^2} = 0 \implies b_{\text{opt}} = \frac{1}{2}\log_2\left(\frac{n \cdot W_B}{3 \cdot \sigma_B^2}\right).$$

The unique solution to this equation corresponds to a *minimum*, as inserting b_{opt} into the second derivative

$$\frac{\mathbf{d}^2}{\mathbf{d}b^2} C_{\text{appx}}(B) = \frac{4^{-b} \cdot n \cdot W_B \cdot log_2(4)}{3 \cdot \sigma_B^2}$$

leads to a positive value.

Utilization of the Hierarchical Relationship. We do not want to code the (inexact) position of μ_B without the prior knowledge of the PDF associated with cluster A. Without this knowledge, we would have to select a suitable range of values and code μ_B at the determined precision b assuming e.g. a uniform distribution inside this range. In contrast, μ_B is a value taken from the distribution function of cluster A. Hence, the number of bits used for coding of μ_B corresponds to the overall density around the imprecise interval defined by μ_B, i.e.

$$c_{\text{hMDL}}(\mu_B) = -\log_2 \int_{\mu_B - 1/2^b}^{\mu_B + 1/2^b} N(\mu_A, \sigma_A^2, x)\, \mathbf{d}x.$$

Figure 3.3(d) visualizes the complete interval of all possible values for the recovered mean value (marked in red) and illustrates the PDF of the cluster$A$, which is the predecessor of cluster B. $\widetilde{\mu}_B$ can be coded by determining the whole area under the PDF of A where $\widetilde{\mu}_B$ could be. The area actually corresponds to a probability value. The negative logarithm of this probability represents the required code length for μ_B. The costs for coding all points of cluster B and μ_B then corresponds to

$$c_{\text{appx}}(b) \cdot n \cdot W_B + c_{\text{hMDL}}(\mu_B).$$

Note, that it is also possible to optimize b directly by setting the derivative of this formula to zero. However, this is impossible in an analytic way, and the difference to the optimum which is obtained by minimizing $C_{\text{appx}}(B)$ is negligible. In addition, if the parent A of B is not the root of the hierarchy, μ_B causes some own ID cost. In this case, μ_B is a sample from the complex distribution function of the hierarchical cut (cf. Definition 2), which prunes the complete level of B and all levels below. Hence, the weight of these levels is added to the new leaf nodes (after cutting), and the ID costs of μ_B correspond to:

$$-\log_2 \big(\sum_{X \sqsubseteq A} W_X\big).$$

A similar analysis can be done for the second parameter of the distribution function, σ_B. Since it is not straightforward to select a suitable distribution function for the Huffman coding of variances, one can apply a simple trick: Instead of coding σ_B, we code $y_B = \mu_B \pm v \cdot \sigma_B$, where v is a constant close to zero. Then, y_B is also a sample from the distribution function $N(\mu_A, \sigma_A^2, x)$ and can be coded similar to μ_B. Therefore, $c_{\text{hMDL}}(\sigma_B) = c_{\text{hMDL}}(\mu_B)$, and we write $c_{\text{hMDL}}(\text{param})$ for the coding costs per parameter instead. In general, if the PDF, which is associated with a cluster has r parameters, then the optimal number of bits can be obtained by the formula:

$$b_{\text{opt}} = \frac{1}{2} \log_2 \left(\frac{n \cdot W_B}{3 \cdot r \cdot \sigma_B^2}\right).$$

And the overall coding costs are:

$$C_{\text{hMDL}}(B) = c_{\text{appx}}(b) \cdot n \cdot W_B + r \cdot c_{\text{hMDL}}(\text{param})$$

Until now, only the tradeoff between coding costs of points and the parameters of the assigned cluster are taken into account. If we go above the lowest level of the cluster hierarchy, we have to trade between coding costs of parameters at a lower level and coding costs of the parameters at the next higher level. This can be done in a similar way as before: Let b_B be the precision, which has already been determined for the representation of μ_B and σ_B, the parameters for cluster B, which is a subcluster of A. However, this is the minimum coding costs assuming that μ_A and σ_A have been stored at maximum precision, and that μ_B and σ_B are also given. Now, we assume that μ_B is an arbitrary point selected from the distribution function $N(\mu_A, \sigma_A^2, x)$ and determine an expectation for the cost:

$$\int_{-\infty}^{+\infty} - \log_2 \int_{\mu_B - 1/2^{b_B}}^{\mu_B + 1/2^{b_B}} N(\mu_A, \sigma_A^2, x) \mathbf{d}x \; N(\mu_A, \sigma_A^2, \mu_B) \mathbf{d}\mu_B.$$

Finally, we assume that μ_A is also coded inexactly by its own grid with resolution b_A. Then the expected costs are:

$$2^{b_A-1}\int_{\mu_A-1/2^{b_A}}^{\mu_A+1/2^{b_A}}\int_{-\infty}^{+\infty}\left(-\log_2\int_{\mu_B-1/2^{b_B}}^{\mu_B+1/2^{b_B}} N(y,\sigma_A^2,x)\,\mathbf{d}x\right)\cdot$$

$$\cdot\, N(\mu_A,\sigma_A^2,\mu_B)\,\mathbf{d}\mu_B\,\mathbf{d}y.$$

Since it is analytically impossible to determine the optimal value of b_A, we can easily get an approximation of the optimum by simply treating μ_B and σ_B like the points which are directly associated with the cluster A. The only difference is the following. While the above integral considers that the PDF varies inside the interval $[\mu_B - 1/2^{b_B}, \mu_B + 1/2^{b_B}]$ and determines the average costs in this interval, treating the parameters as points only considers the PDF value at one fixed position. This difference is negligible provided that $\sigma_B < \sigma_A$, which makes sense as child clusters should usually be much smaller (in terms of σ) than their parent cluster.

Coding Costs for a Cluster. Summarizing, the coding costs for a cluster can be obtained as follows: (1) Determine the optimal resolution parameter for each dimension according to the formula:

$$b_{\text{opt}} = \frac{1}{2}\log_2\left(\frac{n\cdot W_B + r\cdot \text{\#ChildNodes}(B)}{3\cdot r\cdot \sigma_B^2}\right).$$

(2) Determine the coding costs for the data points and the parameters according to:

$$C_{\text{hMDL}}(B) = c_{\text{appx}}(b)\cdot n\cdot W_B + r\cdot c_{\text{hMDL}}(\text{param})$$

(3) Add the costs obtained in step (2) to the ID costs of the points $(-nW_B\log_2(W_B))$ and of the parameters $(-\log_2(\sum_{X\sqsubseteq A} W_X))$. Whereas the costs determined in (2) are individual in each dimension the costs in (3) occur only once per stored point or parameter set of a cluster.

Coding Costs for the HCS. The coding costs for all clusters sum up to the overall coding costs of the hierarchy where we define constant ID costs for the parameters of the root:

$$hMDL = \sum_{C \in \mathcal{N}} \left(C_{\text{hMDL}}(C) - nW_C \log_2(W_C) - \log_2(\sum_{X \sqsubseteq \text{parent of } C} W_X) \right).$$

Obtaining and Optimizing the HCS

We optimize our objective function in an EM-like clustering algorithm ITCH. Reassignment of objects and re-estimation of the parameters of the cluster hierarchy are done interchangeably until convergence. Starting from a suitable initialization, ITCH periodically modifies the hierarchy.

Initialization of the HCS. Clustering algorithms that follow the EM-scheme have to be suitable initialized before starting with the actual iterations of E- and M-step. An established method is to initialize with the result of a k-means [72] clustering. This is typically repeated several times with different seeds and the result with best mean squared overall deviation from the cluster centers is taken. Following this idea, ITCH uses an initialization hierarchy determined by a bisecting k-means algorithm taking the $hMDL$ value of the HCS as a stopping criterion for partitioning. First, a root node that contains all points is created. Then this root node is partitioned into two subclusters by applying k-means with $k = 2$. This is done recursively until the $hMDL$ of the binary hierarchy does not improve anymore within three steps. This ensures not to get stuck in a local minimum. Finally, after the best hierarchy is selected, μ_C and Σ_C are determined for each node C according to Section 3.1.1, and equal weights are assigned to the nodes, to ensure that clusters compete likewise for the data points.

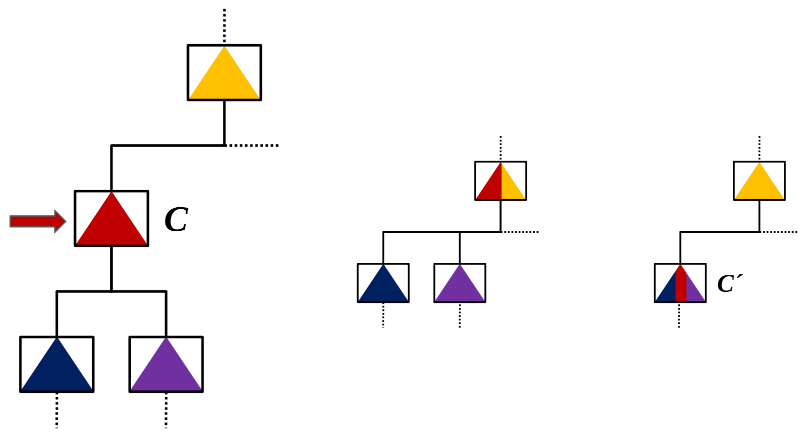
E-step and M-step. Whenever an object is associated directly to a cluster C then it is also indirectly associated with every ancestor of C. Nevertheless, points can also be directly associated not only to leaf nodes but also to inner nodes of the cluster structure. For instance, if a point P_i is an outlier w.r.t. any of the clusters at the bottom level of the hierarchy, then P_i has to be associated with an inner node or even the root. As established in Section 3.1.1, the clusters at all levels of the hierarchy compete for the data points. A point x is directly associated with that Cluster $C \in \mathcal{N}$ the probability density function of which is maximal:

$$Cl(x) = \arg\max_{C \in \mathcal{N}} \left\{ W_C \cdot N(\mu_C, \Sigma_C, x) \right\}.$$

In the E-step of our hierarchical clustering algorithm, the direct association $Cl(x)$ for every object x is updated. Whereas, in the E-step only the direct association is used in the M-step which updates the location and scale parameters of all clusters we use both the direct and indirect association. The motivation is the following: The distribution function of every node in the cluster hierarchy should always represent the whole dataset in this branch of the tree, and the root node should even represent the complete dataset. Therefore, for the location and scale parameters, all directly and indirectly associated objects are considered, as in the following formulas:

$$\begin{aligned} \mu_C &= \frac{\sum_{B \in \mathcal{N}, B \sqsubseteq C} \left(\sum_{x \in B} x \right)}{\sum_{B \in \mathcal{N}, B \sqsubseteq C} |B|}, \sigma^2_{C,j} = \frac{\sum_{B \in \mathcal{N}, B \sqsubseteq C} \left(\sum_{x \in B} (x_j - \mu_{C,j})^2 \right)}{\sum_{B \in \mathcal{N}, B \sqsubseteq C} |B|} \\ \Sigma_C &= \text{diag}(\sigma^2_{C,1}, ..., \sigma^2_{C,d}). \end{aligned}$$

In contrast, the weight W_C of each cluster should reflect the strength of the individual Gaussian in the overall mixture of the HCS and sum up to 1 in order to define a valid PDF with an integral over the complete data space of 1. Therefore, we use the direct associations for calculating the cluster weight with $W_C = |C|$.



(a) Detail of the initial state of a given hierarchy.

(b) Hierarchy after deleting cluster C.

(c) Hierarchy after collapsing cluster C with its child nodes.

Figure 3.4: Restructure operations of ITCH.

Rearrangement of the HCS. The binary hierarchy that results from the initialization, does not limit our generality. ITCH is flexible enough to convert the initial cluster structure into a general one. Given a binary hierarchy, which is deeper than any n-ary hierarchy with $n > 2$, ITCH aims in flattening the hierarchy as far as the rearrangement improves the $hMDL$ criterion. Therefore we trade off the two operations *delete* or *collapse* a node to eliminate clusters that do not pay off any more. Figure 3.4 visualizes the operations for an extract of a given HCS. By deleting a cluster C, the child nodes of C become child nodes of the parent of C (cf. Figure 3.4(b)). By collapsing C, all of its child nodes are merged into a new cluster C' (including C), and therefore all of their child nodes become child nodes of C' (cf. Figure 3.4(c)). Afterwards all points are redistributed, and E- and M-step are performed alternately until convergence. ITCH rearranges the structure in an iterative way. In each iteration we tentatively delete/collapse each node in the hierarchy and perform E- and M-steps. Then first, the node and the operation that improves the $hMDL$ criterion best is selected and second, the corresponding local neighborhood (parent, child and sibling nodes) is processed.

3.1.2 Experimental Evaluation

Since ITCH is a hybrid approach combining the benefits of hierarchical *and* model-based clustering, we compare to algorithms of both classes to demonstrate the effectiveness of ITCH. We selected Single Link (SL) [53] which probably is the most common approach to hierarchical clustering. As especially on noisy data, SL suffers from the so-called Single Link effect, we additionally compare to OPTICS [4], a more outlier-robust hierarchical clustering algorithm, with a standard parameterization of $MinPts = 6$ and $epsilon = 0.9$. Furthermore, we compare to RIC [10], an outlier-robust and parameter-free state-of-the-art algorithm to model-based clustering. ITCH is implemented in Java. SL was provided by Matlab and OPTICS was provided by WEKA [42]. For RIC we used the original implementations by the authors.

For each dataset, the clustering results were assessed against a ground-truth classification. To compare the clustering results produced by different approaches, we chose measures that comprise fix bounds. More precisely, we selected the following recently proposed information-theoretic methods [77]: the Normalized Mutual Information (NMI), the Adjusted Mutual Information (AMI), and the Expected Mutual Information (EMI). These three information-theoretic measures are based on the general mutual information (MI) of two given clusterings $C = \{C_1, \cdots, C_l\}$ and $C' = \{C_1, \cdots, C_k\}$ of a dataset DS and a contingency table $M = [m_{ij}]_{j=1,\cdots,k}^{i=1,\cdots,l}$, where each entry denotes the number of common objects of both clusterings. Hence, MI is a symmetric measure that quantifies the information that the two clusterings C and C' share. Formally it is defined as follows:

$$MI(C, C') = \sum_{i=1}^{l} \sum_{j=1}^{k} p(i,j) \log \frac{p(i,j)}{p(i)p(j)},$$

where $p(i,j)$ denotes the probability that a point belongs to cluster $C_i \in C$ and cluster $C'_j \in C'$. The NMI then is a normalized version of the MI, by a division by the maximum value of the index. It takes a value of 1 if two clusterings are

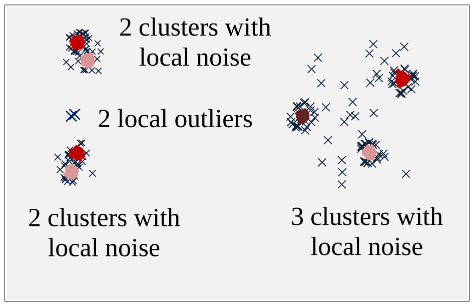
identical and 0 if the two clusterings are independent. The AMI is based on the EMI, and is corrected for randomness. It takes a value of 1 if the two clusterings are identical and 0 if the mutual information between the two clusterings equals its expected value. The EMI defines the average mutual information between all pairs of clusterings that have the same number of clusters and objects in each cluster as in the two given clusters C and C'. Also the EMI take the value of 1 if the resulting clustering corresponds exactly to the ground-truth classification, and 0 if both clusterings are totally independent.
Furthermore, we chose the well-known Precision (Prec) and Recall (Rec) from information retrieval. The Precision of a cluster can be seen as a measure of exactness and is defined as the fraction of the clustered datapoints that are labeled by the true class label (named true positives) devided by all data points of the specific cluster. The Recall is a measure of completeness. It is defined as the proportion of the true positives against all data points that are labeled by the class label of the specific cluster.

Finally, we use the measure proposed by DOM *et al.* [30], referred to as DOM in the following, that corresponds to the number of bits required to encode the class labels when the cluster labels are known. If the number of clusters is equal in both clusterings C and C', DOM is equal to the mutual information. In cases, where the number of clusters is different, it computes the reduction in number of bits that would be required to encode the class labels. Hence, small DOM values refer to good clusterings.

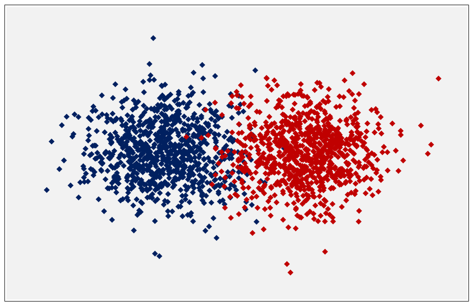
Synthetic Data

Experiments on DS_1 demonstrate the superiority of ITCH on hierarchical datasets. DS_1 comprises about 3,500 2-dimensional points that form a hierarchy of 12 clusters with outliers at different levels of the hierarchy. Seven Gaussian clusters are located at the bottom level (cf. Figure 3.5(a)). Experiments on DS_2 indicate the limitations of existing approaches to form meaningful clusters in extremely noisy non-hierarchical data. DS_2 is composed of two Gaussian clusters with

1,650 points each, overlapping in the marginal area without any global outliers (cf. Figure 3.5(b)).



(a) Plotted DS_1.

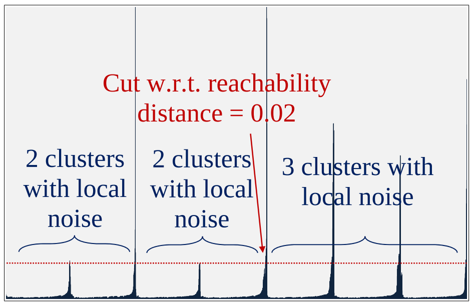


(b) Plotted DS_2.

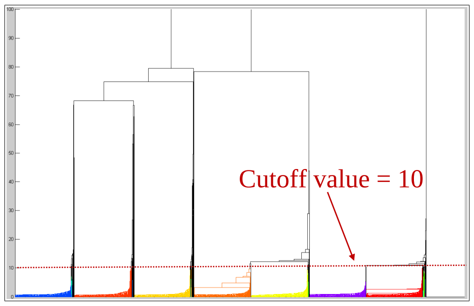
Figure 3.5: The synthetic dataset DS_1 forms a hierarchy including local noise with different degrees of outlierness. DS_2 consists of two highly overlapping clusters. It comprises no outliers.

Experimental Evaluation on $\mathbf{DS_1}$. As clusters can be recognized as valleys in the reachability plot illustrated in Figure 3.6(a), OPTICS yields a satisfactory solution resulting in good validity values (cf. Table 3.1) w.r.t. a cut through the hierarchy at a maximum reachability distance of 0.02. But one has to notice that it is hard to spot the outliers since high distance peaks can also be caused by the usual jumps. Also, the SL-hierarchy (cf. Figure 3.6(b)) represents the true hierarchy quite well. The reason for that is that DS_1 comprises a hierarchy that can be recognized by agglomerative clustering algorithms relatively easy. However, the DOM value of 0.2142 reflects that the clustering information of the outlier objects causes slightly more coding costs, as the red cluster on the right side suffers from a slight Single Link effect. All values shown in Table 3.1 refer to a cut through the dendrogram at a cutoff value of 10, resulting in seven disjoint clusters.

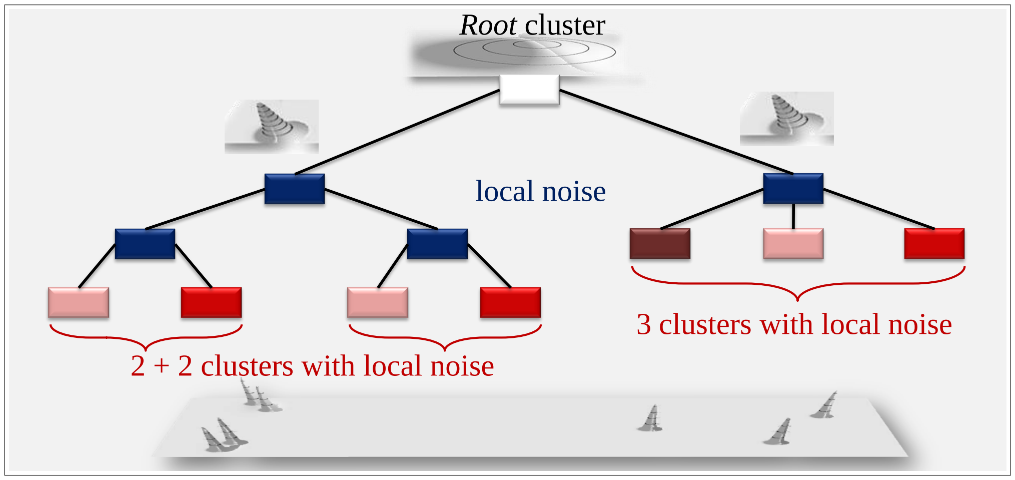
In order to apply RIC to the hierarchical dataset, we preprocessed DS_1 with SL and applied RIC ($k = 18$) as postprocessing step in each level of the hierarchy.



(a) OPTICS on DS_1.



(b) SL on DS_1.



(c) The result of ITCH on DS_1 comprises the hierarchical cluster structure and the PDF of each cluster content.

Figure 3.6: Competitive evaluation of ITCH on the synthetic dataset DS_1.

Table 3.1 shows the validation of the corresponding clustering result. Even RIC fails to successfully filter out all outliers. More precisely, it assigns points that obviously are local outliers misleadingly to clusters. Also a majority of the outliers are incorrectly identified as cluster points. Hence, RIC performs worst on the hierarchical dataset DS_1 concerning the quantitative analysis presented in Table 3.1. Only ITCH detects the true cluster hierarchy including outliers fully automatically, and provides meaningful models on the data for each level of the hierarchy (cf. Figure 3.6(c)).

Table 3.1: Performance of ITCH on DS_1.

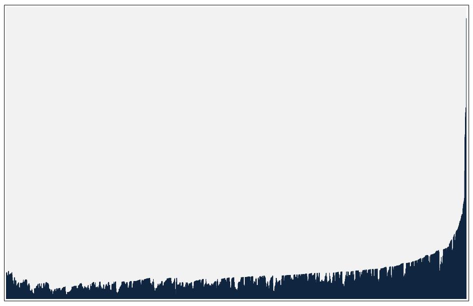
	ITCH	RIC	OPTICS	SL
NMI	0.9895	0.8665	0.9585	0.9555
AMI	0.9894	0.8453	0.9461	0.9632
EMI	0.0001	0.0001	0.0001	0.0001
PREC	0.9958	0.8589	0.9654	0.9903
REC	0.9956	0.8883	0.9719	0.9601
DOM	0.1334	0.4279	0.2167	0.2142

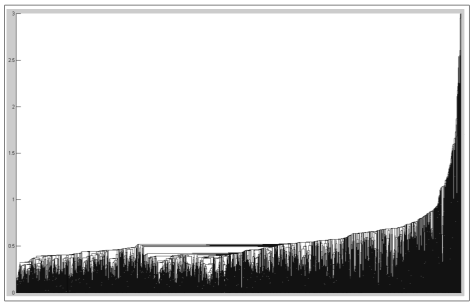
Experimental Evaluation on $\mathbf{DS_2}$. RIC merges the two Gaussian clusters into only one cluster which results in a bad cluster quality (cf. Table 3.2). Also with OPTICS and SL, it is impossible to detect the true structure of DS_2 (cf. Figures 3.7(a) and 3.7(b)). OPTICS assigns the points in an almost arbitrary order. Even when increasing the parameter for the minimum object density per cluster to a large value, OPTICS fails in detecting two clusters. SL miscarries due to the massive Single Link effect. The hierarchies generated by OPTICS and SL are overly complex but do not capture any cluster structure. Hence, it has to be assumed that all data points are assigned to one single cluster. Only ITCH discovers a meaningful result without requiring any input parameters. All clusters that do not pay off w.r.t. $hMDL$ criterion are pruned and therefore, only two Gaussian clusters remain in the resulting flat hierarchy which are described by the corresponding PDF (cf. Figure 3.7(c)). Since DS_2 does not contain any outliers, ITCH assigns all points to clusters that are located at the bottom level. ITCH performs best w.r.t. all validity measures shown in Table 3.2.

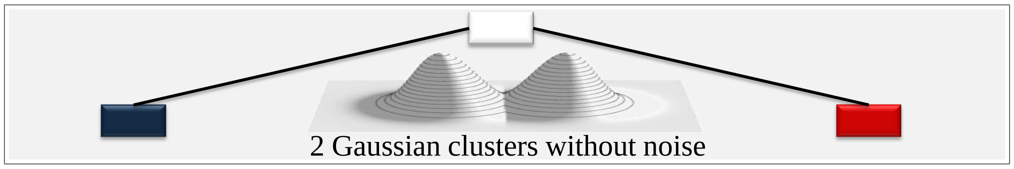
Real World Data

Finally, we show the practical application of ITCH on real datasets available at the UCI Machine Learning repository [1].

[1] `http://archive.ics.uci.edu/ml/datasets`


(a) OPTICS on DS_2.


(b) SL on DS_2.



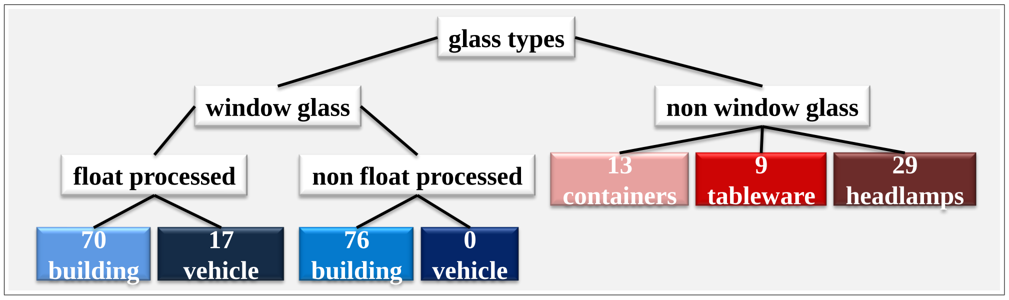
(c) Resulting cluster structure of ITCH on DS_2 including the PDF of each cluster content.

Figure 3.7: Competitive evaluation of ITCH on the synthetic dataset DS_2.

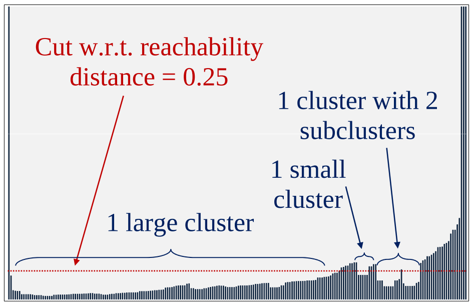
Glass Data. The *Glass Identification* dataset comprises nine numerical attributes representing different glass properties. 214 instances are labeled according to seven different glass types that form a hierarchy as presented in Figure 3.8(a). ITCH perfectly separates *window glass* from *non window glass*. The four sub-clusters of *window glass* are very similar. Hence, ITCH arranges them at the same level. Some outliers are directly assigned to *window glass*. In contrast to ITCH, neither SL nor OPTICS separates *window glass* from *non window glass*

Table 3.2: Performance of ITCH on DS_2.

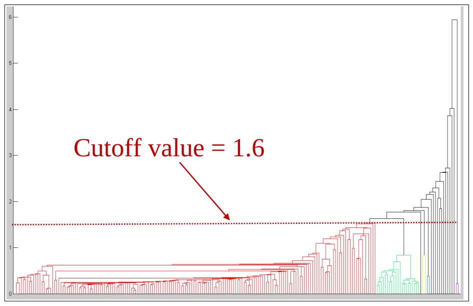
	ITCH	RIC	OPTICS	SL
NMI	0.9586	0.0000	0.0000	0.0000
AMI	0.9586	0.0000	0.0000	0.0000
EMI	0.3466	0.3466	0.3466	0.3466
PREC	0.9949	0.2521	0.2521	0.2521
REC	0.9948	0.5021	0.5021	0.5021
DOM	0.0332	0.6956	0.6956	0.6956



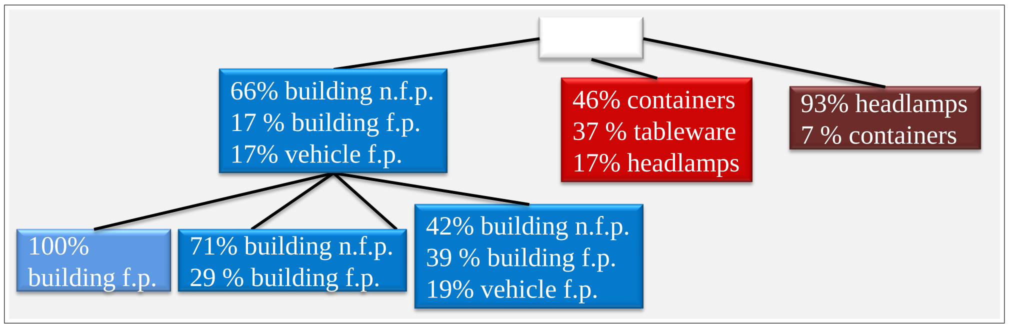
(a) Hierarchy, including ground-truth classification in the original dataset.



(b) OPTICS on *Glass*.



(c) SL on *Glass*.



(d) Resulting cluster structure of ITCH on *Glass*.

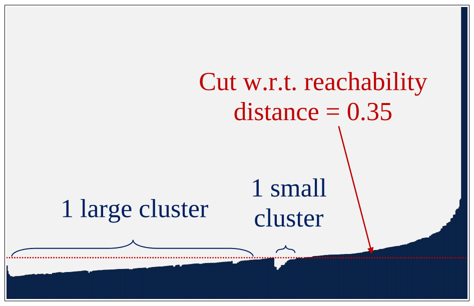
Figure 3.8: Competitive evaluation of ITCH on the real-world *Glass* dataset.

perfectly. In the dendrogram one large cluster and one medium size cluster is visible, whereas OPTICS determines two clusters of different size and one cluster with two nested subclusters. RIC, applied at the bottom level of the SL result comprises only two clusters without any separation between *window glass* or *non window glass*. Table 3.3 summarizes the quantitative analysis of this dataset. For OPTICS the results refer to the partitioning clustering with a maximum reachability distance of 0.25. The SL results indicate the results for a cutoff value of 1.6.

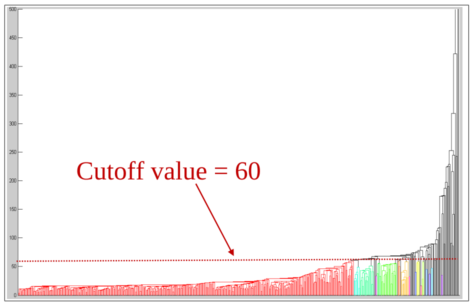
Table 3.3: Performance of ITCH on the real-world *Glass* dataset.

	ITCH	RIC	OPTICS	SL
NMI	0.3390	0.1221	0.4029	0.4110
AMI	0.3222	0.0804	0.3092	0.3085
EMI	0.0018	0.0010	0.0017	0.0012
PREC	0.5350	0.1710	0.4739	0.3812
REC	0.3551	0.1822	0.4626	0.4393
DOM	1.4805	1.5872	1.2980	1.5010

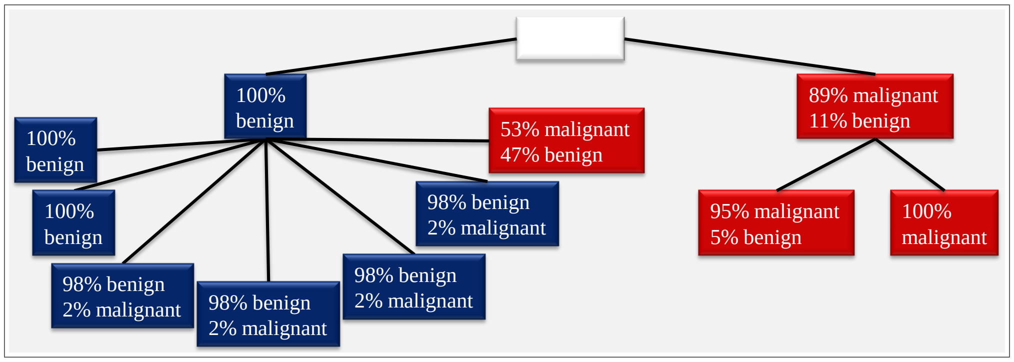
Cancer Data. The high-dimensional *Breast Cancer Wisconsin* dataset contains 569 instances each describing 30 different characteristics of the cell nuclei, where each instance is either labeled by *benign* (blue) or *malignant* (red). OPTICS and SL both fail to detect a clear cluster structure in this dataset. OPTICS finds one large cluster and one very small cluster that could also be interpreted as noise (cf. Figures 3.9(a)). In the dendrogram of SL, shown in Figure 3.9(b), one large and many small clusters are visible. The quantitative evaluation represented in Table 3.4 refer to a cutoff value of 60.
We applied RIC on top of a k-means clustering with $k = 15$. As stated by the authors we chose k large enough compared to the number of classes. However, RIC also fails and results in three mixed clusters. ITCH almost perfectly separates the *benign* from the *malignant* objects which are then split into different subclusters (cf. Figure 3.9(c)). This precise structure is penalized by the validity measures summarized in Table 3.4. However, this result is consistent with previous findings



(a) OPTICS on *Cancer*.



(b) SL on *Cancer*.



(c) Resulting cluster structure of ITCH on *Cancer*.

Figure 3.9: Competitive evaluation of ITCH on the real-world *Cancer* dataset. The coloring corresponds to the majority class.

as the two classes exhibit a degree of overlap with each other [80].

Stability of ITCH

As stated in Section 3.1.1, ITCH is initialized with the results of a k-means clustering with different seeds. Since we do not want to rely on single results we additionally tested the stability of ITCH over 20 runs for each dataset. Figure 3.10 shows the variance of the $hMDL$ value in percent depending on the mean value. The result of ITCH is highly stable within DS_1, DS_2 having only a variance of 0.03% and 0.12%, respectively. Also in the real world datasets the result of ITCH shows only little variance.

Table 3.4: Performance of ITCH on the real-world *Cancer* dataset.

	ITCH	RIC	OPTICS	SL
NMI	0.3821	0.4594	0.2835	0.3097
AMI	0.2044	0.3612	0.2518	0.2127
EMI	0.0004	0.0003	0.0003	0.0006
PREC	0.9574	0.9846	0.8033	0.8884
REC	0.2830	0.6608	0.7575	0.6626
DOM	0.2635	0.3022	0.4752	0.4771

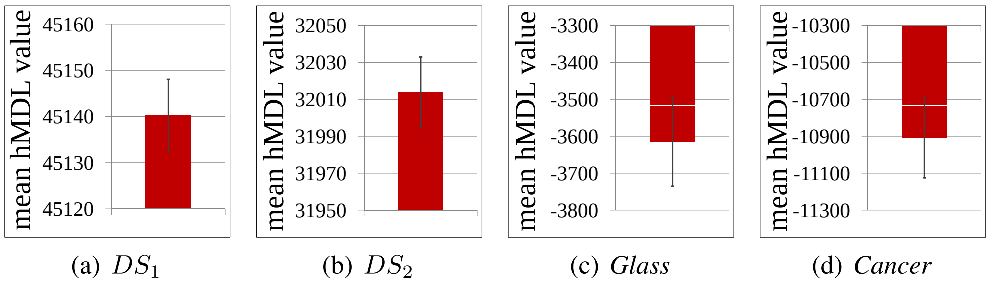


Figure 3.10: Stability of the ITCH results, determined over 20 runs for each dataset.

3.1.3 Conclusions

We have introduced a new hierarchical clustering method to arrange only natural, valid, and meaningful clusters in a hierarchical structure – ITCH. ITCH is based on an objective function for clustering that was guided by the information-theoretic idea of data compression. We have shown that without difficult parameter settings ITCH finds the *real* cluster hierarchy effectively, and that it provides accurate and intuitive interpretable information in a wide variety of domains, even in the presence of local and global outliers that correspond to inner nodes of the resulting hierarchy.

3.2 GACH: A Genetic Algorithm for Hierarchical Clustering

Many EM-based algorithms, including ITCH, suffer from the problem that they often get stuck in a local optimum. One solution that overcomes that problem is the application of genetic algorithms (GAs). A GA is a stochastic optimization technique based on the mechanism of natural selection and genetics, originally proposed by [45]. The general idea behind a GA is that the candidate solutions to an optimization problem (called *individuals*) are often encoded as binary strings (called *chromosomes*). A collection of these chromosomes forms a *population*. The evolution initially starts from a random population that represents different individuals in the search space. In each generation, the fitness of every individual is evaluated, and multiple individuals are then selected from the current population based on Darwin's principle "Surviving of the fittest". These individuals build the mating pool for the next generation. The new population is then formed by the application of recombination operations like *crossover* and *mutation*. A GA commonly terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. A survey of GAs along with the programming structure used can be found in [34].

GAs have been successfully applied to a variety of challenging optimization problems, like image processing, neural networks and machine learning, etc. [79, 104, 6]. Solving the NP-hard clustering problem makes GA therefore a natural choice such as in [60, 26, 74, 86, 70, 82]. Our motivation for using GAs for hierarchical clustering is visualized in Figure 3.11. The clustering research community focuses on clustering methods where the grouped objects are described by an intuitive model of the data [28] or clustering methods that are particularly insensitive to outliers [31]. Moreover, several approaches have also been addressing the question, how to avoid difficult parameter settings such as the number of clusters, e.g. [81, 43, 10, 11, 12]. Most of them meet this question by relating the clustering problem with the idea of data compression.

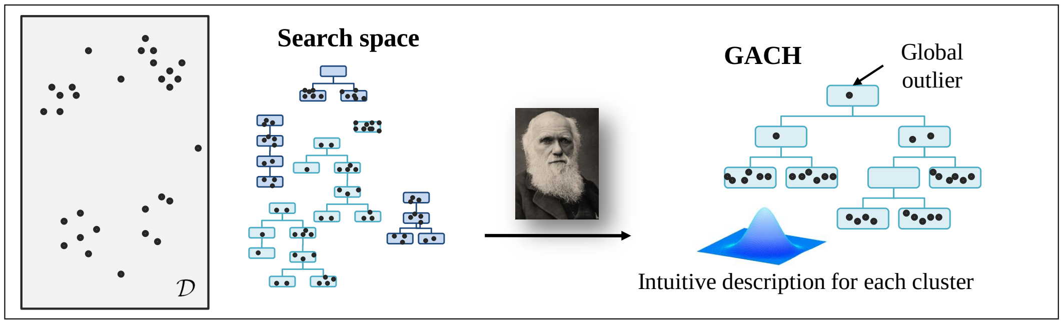


Figure 3.11: The search space of the solutions for the hierarchical clustering problem is exponentially large. Genetic algorithms help to determine the global optimum. GACH is a genetic algorithm for finding the most meaningful cluster hierarchy. Outliers are assigned to appropriate inner nodes. Each cluster content is described by an intuitive description in form of a PDF.

Here we present a novel genetic algorithm for finding cluster hierarchies, called GACH. We use an information-theoretic fitness function to effectively cluster data into meaningful hierarchical structures without requiring the number of clusters as an input parameter. Our major contributions are:

- *Fitness*: The fitness of different chromosomes is optimized using the same MDL-based optimization technique as used for ITCH.
- *No difficult parameter-setting*: Besides the parameters that are specific for a genetic algorithm, GACH requires no expertise about the data (e.g. the number of clusters).
- *Flexibility*: By the use of a GA-based stochastic search GACH thoroughly explores the search space and is therefore flexible enough to find the correct hierarchical cluster structure and is not sensitive to the initialization.
- *Outlier-handling*: Outliers are assigned to the root of the cluster hierarchy or to an appropriate inner node, depending on the degree of outlierness.
- *Model description*: The content of each cluster is described by a PDF.

3.2.1 The Algorithm GACH

In this section we describe the basic components of a genetic algorithm (GA) and introduce necessary modifications to use a GA on cluster hierarchies. Finally, we present GACH as an algorithmic combination of all this components.

Chromosomal Representation of Cluster Hierarchies

Each chromosome specifies one solution to a defined problem. For GACH, a chromosome is the encoding of a hierarchical cluster structure (HCS), which has to address the three following features:

- Storage of k clusters, where k is an arbitrary number of clusters.
- Representation of the hierarchical relationship between clusters.
- Encoding of the cluster representatives, i.e. the parameters of the underlying PDF. For GACH we represent each cluster by a Gaussian PDF. Note that our model can be extended to a variety of other PDFs, e.g. uniform or Laplacian.

Therefore, a chromosomal representation of a HCS is defined as follows:

Definition 3 (Chromosomal HCS)
(1) A ***chromosomal HCS*** HCS_{Chrom} *is a dynamic list storing* k *cluster objects.*
(2) Each cluster C *holds references to its parent cluster and to its subclusters. Besides that, there is a level attribute which facilitates the interpretation of the HCS.*
(3) The parameters of the underlying Gaussian PDF of cluster C*, the mean value* μ_C *and* σ_C*, are modeled as additional parameters of the cluster object* C*.*
(4) Each cluster C *is associated with a weight* W_C*, where* $\sum_{i=0}^{k-1} W_{C_i} = 1$.

The underlying PDF of a cluster C is a multivariate Gaussian in a d-dimensional data space which is defined by the parameters μ_C and σ_C (where μ_C and σ_C are vectors from a d-dimensional space) by the following formula:

$$N(\mu_C, \sigma_C, x) = \prod_{1 \leq i \leq d} \frac{1}{\sqrt{2\pi\sigma_{C,i}^2}} \cdot \mathbf{e}^{-\frac{(x_i - \mu_{C,i})^2}{2\sigma_{C,i}^2}}$$

GACH assigns each point $x \in DB$ *directly* to that cluster $C \in HCS_{Chrom}$ the probability density of which is maximal at the position of x:

$$C(x) = \arg \max_{C \in HCS_{Chrom}} \{W_C \cdot N(\mu_C, \sigma_C, x)\}.$$

Initialization of GACH

In a GA, the initial set of a population consists of a randomly generated set of individuals. Therefore, GACH selects a random number of clusters $\tilde{k}$ for each structure HCS_{Chrom} in a first step. Then a simple k-means algorithm [72] divides the dataset into $\tilde{k}$ clusters representing the leafs of the initial hierarchy. Finally, these clusters are combined by one additional root cluster. Hence, the initialization process results in a 2-level hierarchy that consists of $\tilde{k} + 1$ nodes. Each cluster C is described by random parameters and is associated to a weight $W_C = \frac{1}{k}$.

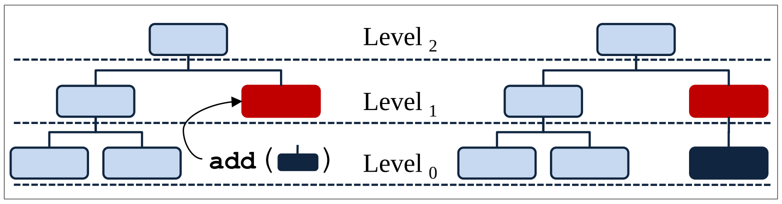
Reproduction

In order to generate the next population, GACH uses several genetic operators that are particularly defined for the hierarchical clustering problem. As our algorithm is initialized by a flat hierarchy generated by a partitioning clustering procedure, GACH needs mutating oparators that add nodes to the hierachy. However, only adding new nodes would not leed to sufficient results because parameters like μ, σ and the cluster weight are chosen randomly. Therefore, we need a further operator which demotes given nodes to different hierarchy levels. Additionally, the central goal of a genetic algorithm is an efficient eradication of the search space. Hence, we need for each mutation operator also the inverse procedure to avoid to get stuck in only a local optimum. Finally, we use the mutations `add`, `delete`, `demote` and `promote`, and the GA-specific operator `crossover`.

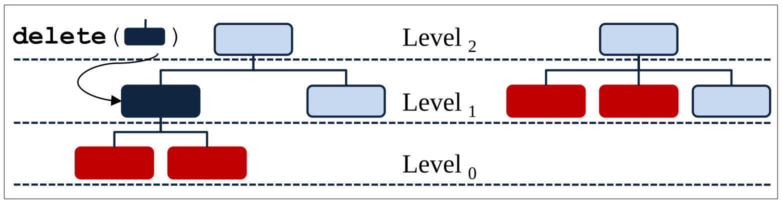
The operator `add` adds direct subclusters to an arbitrary cluster C of the hierarchy with an add rate p_{add} (normally $p_{del} = p_{add}$). The number of added subclusters is bounded by an upper limit value max_{new}. Figure 3.12(a) illustrates an example for the application of the `add` operator to a HCS. One subcluster (marked in dark blue color) is added to the red cluster. Since the cluster content of a added subcluster C_{add} is covered by the cluster content of C, we calculate random parameters based on μ_C and σ_C. In particular, we add a random factor r to both parameters, where r is a vector from a d-dimensional space: $\mu_{C_{add}} = \mu_C + r \qquad \sigma_{C_{add}} = \sigma_C + r$.

The operator `delete` deletes a specific cluster C (except the root) with a deletion rate p_{del} from the HCS. This results in structure HCS$'$ that does not contain the cluster C anymore. (cf. Figure 3.12(b)). Here, the cluster C is marked by dark blue color. The level of each direct and indirect subcluster of C (marked in red) is decreased by 1. The former parent node of C, the root node in our example, becomes the parent node of all direct subclusters of C.

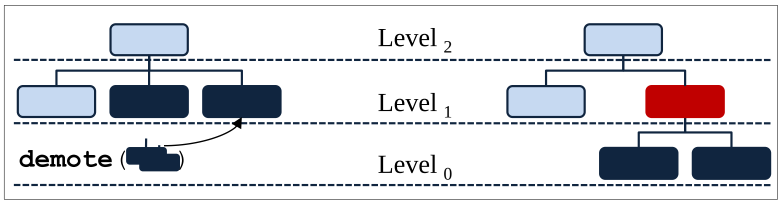
The motivation behind the `demote` operator is the following. Assume a dataset consisting of three clusters C_1, C_2 and C_3, where C_1 holds a large number of objects, clusters C_2 and C_3 are smaller ones but they are locally close to each other. Then one could create a HCS with one root node and C_1, C_2 and C_3 as direct subclusters (cf. Figure 3.12(c)) which provides only a very coarse view of the dataset. But, if we combine the two smaller clusters (marked in dark blue) and demote them with a demote rate p_{dem} to a lower level with a common parent cluster (marked in dark red), we are able to get a more precise description of our data. The parameters of the inserted cluster are obtained by the average of the parameters of the combined clusters. Note that demoting only one cluster is consistent with the `add` operator. Hence, we apply `demote` on at least two clusters.



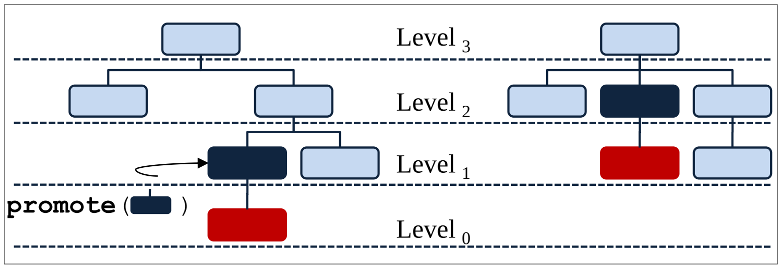
(a) `add` one node do the hierarchy.



(b) `delete` one node of the hierarchy.



(c) `demote` two nodes to a lower level.



(d) `promote` one node to a higher level.

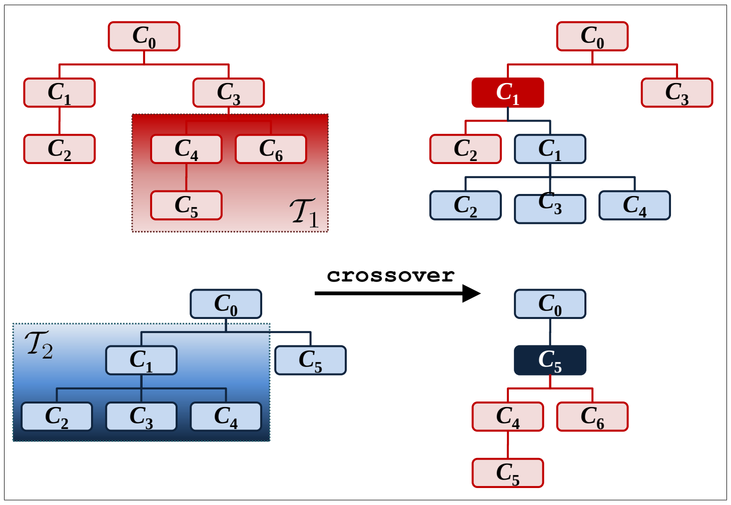
Figure 3.12: Summarization of the mutation operators used for GACH.

The `promote` operator lifts a cluster C from a lower level to the level right above with a promotion rate p_{pro}, if and only if C is at least two levels underneath the root cluster. Consequently all subclusters of C are lifted accordingly. In Figure 3.12(d) the dark blue cluster is promoted from level 1 to level 0. Hence also the red subcluster is lifted one level. The parent of the parent node of the dark blue cluster (here the root node) becomes the parent node of C in the resulting hierarchy HCS$'$, together with the correct rearrangement of all subclusters.

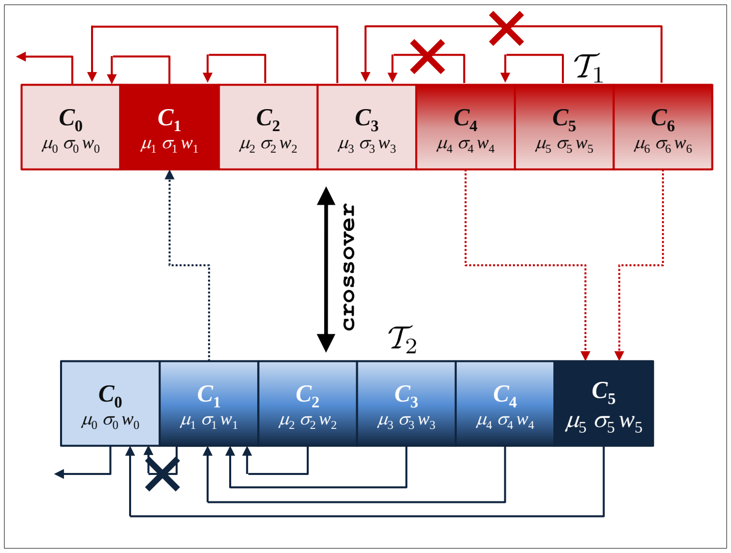
The operator `crossover` exchanges information among two different structures. In general the information of two different chromosomes is combined in order to obtain a new individual with superior quality. GACH performs a crossover between two selected hierarchies HCS_1 and HCS_2 with a crossover rate p_{co} as follows:

1. Remove a selected subtree, denoted by $\mathcal{T}_1$, entirely from HCS_1.
2. Remove a selected subtree, denoted by $\mathcal{T}_2$, entirely from HCS_2.
3. Select a random node in HCS_1 and insert $\mathcal{T}_2$.
4. Select a random node in HCS_2 and insert $\mathcal{T}_1$.

Figure 3.13(a) illustrates this procedure exemplarily for two selected hierarchies. The subtrees $\mathcal{T}_1$ and $\mathcal{T}_2$ are removed from the red and the blue HCS respectively. $\mathcal{T}_1$ is then inserted into the blue HCS as subtree of the dark blue node. Analogously $\mathcal{T}_2$ is inserted as subtree of the dark red cluster in the red HCS. Figure 3.13(b) describes the same procedure w.r.t. a chromosomal representation of both hierarchies. For simplicity, only the pointers to the parent cluster are considered. The recalculation of the parameters μ_C and σ_C of each cluster C is performed analogously to ITCH (cf. Section 3.1.1).



(a) Hierarchical representation.


$\mathcal{T}_1$
C_0 $\mu_0\ \sigma_0\ w_0$
C_1 $\mu_1\ \sigma_1\ w_1$
C_2 $\mu_2\ \sigma_2\ w_2$
C_3 $\mu_3\ \sigma_3\ w_3$
C_4 $\mu_4\ \sigma_4\ w_4$
C_5 $\mu_5\ \sigma_5\ w_5$
C_6 $\mu_6\ \sigma_6\ w_6$
crossover
$\mathcal{T}_2$

(b) Chromosomal representation.

Figure 3.13: The `crossover` operator for two selected hierarchies. The subtree $\mathcal{T}_1$ of the red hierarchy is exchanged with the subtree $\mathcal{T}_2$ of the blue hierarchy, visualized by a hierarchical (3.13(a)) and a chromosomal representation (3.13(b)).

Fitness Function

Following Darwin‘s principle “Survival of the fittest” naturally only individuals with highest fitness can survive and those that are weaker become extinct. A GA adopts this aspect of evolution by the use of a fitness function. GACH uses the $hMDL$ criterion formalized in Section 3.1 which evaluates the fitness of a chromosomal HCS by relating the clustering problem to that of data compression by Huffman Coding. The authors define the coding scheme for a cluster hierarchy as follows:

$$hMDL_{HCS} = \sum_{C \in HCS} \left(cost(C) - nW_C \log_2(W_C) - \log_2(\sum_{x \sqsubseteq \text{parent of } C} W_x) \right)$$

The coding cost for each cluster $C \in HCS$ is determined separately and summed up to the overall coding cost of the complete HCS. Points that are directly assigned to the cluster C together with the parameters μ_C and σ_C of the underlying Gaussian PDF are coded by $cost(C)$. The point to cluster assignment is coded by the so-called ID cost of each data point $x \in C$ and is given by $-nW_C \log_2(W_C)$ where W_C is the weight of cluster C and n the number of points. The binary logarithm is used to represent the code length in bits. Clusters with higher weight are coded by a short code pattern whereas longer code patterns are assigned for smaller clusters with lower weight. The ID costs for the parameters are formalized by $-\log_2(\sum_{x \sqsubseteq \text{parent of } C} W_x)$ whereas constant ID costs are defined for the parameters of the root node. The better the statistical model (the HCS) fits to the data, the higher the compression rate, and thus the lower the coding costs. Using this coding scheme as fitness function ensures the selection of that chromosome HCS_{Chrom} that fits best to the data.

Selection

The selection function chooses the best individuals out of a given set of individuals to form the offspring population w.r.t. their fitness. For GACH we use the well-known weighted roulette wheel strategy [76]. Imagine that each HCS_{Chrom} represents a number on a roulette wheel, where the amount of numbers refers to the size of the population. In addition, we assign a weight to each number on the roulette wheel, depending on the fitness of the underlying chromosome, i.e. the better the fitness of a chromosome, the higher its weight on the roulette wheel, i.e. the higher the chance to get selected for the offspring population. Note that there is the chance that one chromosome is selected multiple times. GACH forms a new population that has as much individuals as the previous population.

Algorithmic Description

Now we are putting the pieces together to define the algorithmic procedure of GACH, summarized in Algorithm 1. First, an initial population is built. This population is evaluated according to the $hMDL$-based fitness function which means that GACH determines the coding cost for each cluster hierarchy of the population. In order to optimize the point to cluster assignment of each HCS and to provide an additional model of the data, we apply a hierarchical EM algorithm on each cluster structure, as suggested for ITCH. That does not imply the typical problems of EM-based methods, as the model selection is performed by the idea of GA. EM is only used to provide an additional feature in this case.

The population resulting from the initialization undergoes several mutation and crossover operations within pop_{max} number of generations in an iterative way. In each iteration the next population is selected according to the weighted roulette wheel strategy and experiences various reproduction procedures. Each operation is processed with a certain probability which is extensively evaluated in Section 3.2.2. After optimizing the point to cluster assignment, GACH determines the fitness of each HCS_{Chrom}. The algorithm terminates if a specified maximum number of new populations pop_{max} is reached. The experiments show that the HCS can be optimized even with a small number of generations.

Algorithm 1 GACH

```
1: $count_{pop} \leftarrow 0$
2: initialize $population(count_{pop})$
3: evaluate $population(count_{pop})$
4: while ($count_{pop} \leq pop_{max}$) do
5:     $count_{pop} \leftarrow count_{pop} + 1$
6:     select $population(count_{pop})$ from $population(count_{pop} - 1)$
7:     reproduce $population(count_{pop})$
8:     evaluate $population(count_{pop})$
9: end while
```

3.2.2 Experimental Evaluation

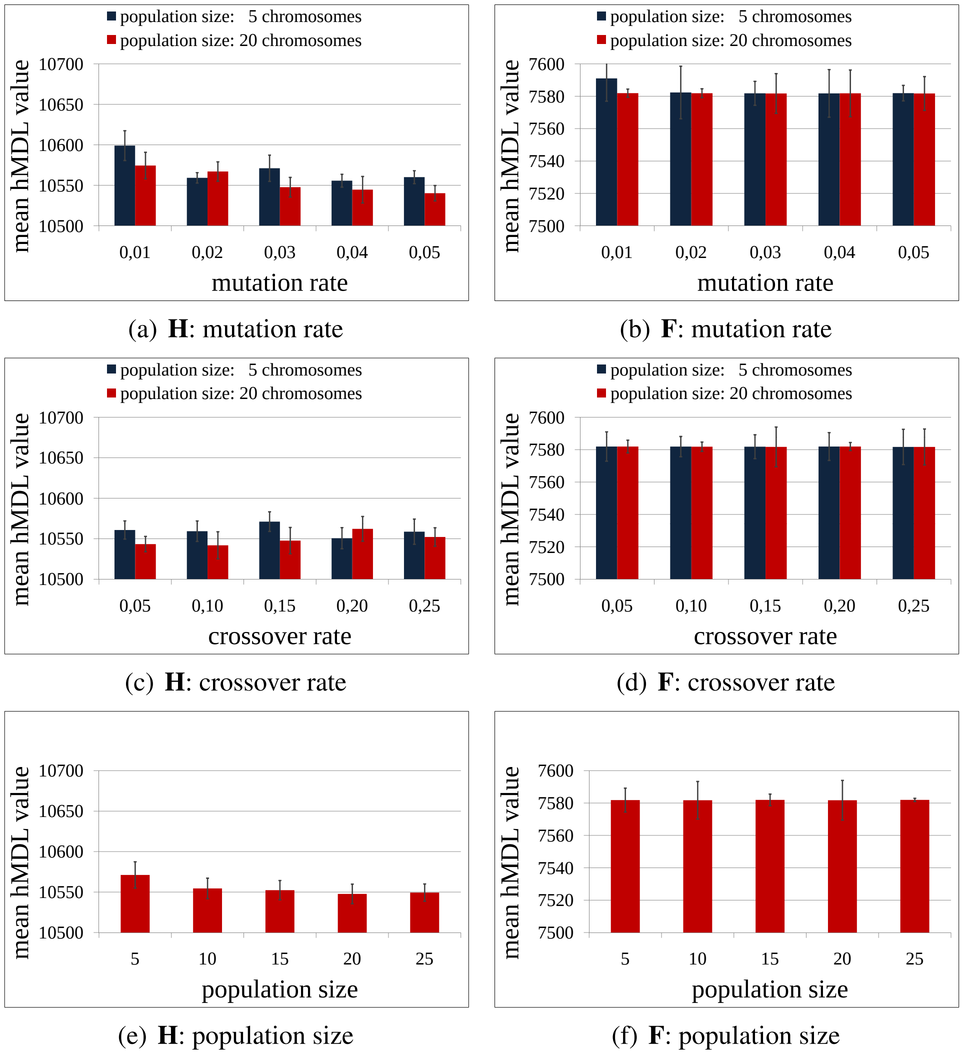
Now we demonstrate that the genetic parameters (mutation rate, crossover rate and population size) do not affect the effectiveness of GACH in a major way. Nevertheless, we provide a suitable parameterization that enables the user to receive good results independent of the used dataset. Based on this, we compared the performance of GACH to several representatives of various clustering paradigms on synthetic and real world data. We selected the most widespread approach to hierarchical clustering Single Link (SL) [53], the more outlier-robust hierarchical clustering algorithm OPTICS [4] (requiring $MinPts$ and ϵ), with optimal parameters w.r.t. accuracy. Furthermore, we chose RIC [10], an outlier-robust and information-theoretic clusterer, and finally ITCH, introduced in Section 3.1, that suffers from the problem that the result only represents a local optimum in some cases. As ITCH depends on its initialization, we used the best out of ten runs in this case.

Evaluation of Genetic Parameters

We applied GACH on two different datasets to evaluate the mutation and crossover rates and the impact of the population size on the quality of the results w.r.t. the fitness function, introduced in Section 3.2.1. One dataset consists of 1,360 (2d)-data points that form a true hierarchy of six clusters. The second dataset covers 850 (2d)-data points that are grouped in two flat clusters. For each experiment, we present the mean $hMDL$ value and the corresponding standard deviation over ten runs. GACH turned out to be very robust and determines very good clustering results ($Prec > 90\%$, $Rec > 90\%$) independent of the parameterizations.

Different Mutation Rates. We evaluated different mutation rates ranging from 0.01 to 0.05 on two different population sizes and a fixed crossover rate of 0.15. As a mutation is performed by one of the four operations `delete`, `add`, `demote` or `promote` the mutation rate is the sum of p_{del}, p_{add}, p_{dem} and p_{pro} (cf. Section 3.2.1). As `demote` and `promote` turned out to be essential for the quality of the clustering results p_{dem} and p_{pro} are typically parameterized by a multiple of p_{del} or p_{add} due to the fact that the correct number of clusters is determined very fast by the $hMDL$-based fitness function. However, finding the best HCS for a given number of clusters is much more challenging. Figures 3.14(a) and 3.14(b) demonstrate that the mutation rate has no outstanding effect on the clustering result, neither on a hierarchical nor on a flat dataset. Higher mutation rates result in higher runtimes (3,388 ms for a mutation rate of 0.05 vs. 1,641 ms for a mutation rate of 0.01 on a hierarchical dataset, population size = 5). However, a higher mutation rate provides more flexibility. Hence, we achieved slightly better results with a mutation rate of 0.05 ($hMDL = 10,520$) compared to a mutation rate of 0.01 ($hMDL = 10,542$).

Different Crossover Rates. We compared different crossover rates p_{co} ranging from 0.05 to 0.25 in combination with a mutation rate of 0.05 on two different population sizes. Figures 3.14(c) and 3.14(d) show that the performance of GACH is almost stable w.r.t. the different parameterizations of p_{co}. Especially on a flat dataset a higher p_{co} has no impact on the clustering result. GACH achieved a nearly optimal $hMDL$ value in almost every run, even for relatively small population sizes. Higher p_{co} values enable GACH to examine the search space more effectively as the crossover between two strong individuals produces an even fitter individual. Therefore, we need fewer generations to find good clustering results, e.g. the result of GACH on the hierarchical dataset using five individuals was determined after 75 generations (1993 ms per generation) with $p_{co} = 0.05$, and after 61 generations (2,553 ms per generation) with $p_{co} = 0.25$.



(a) **H**: mutation rate (b) **F**: mutation rate

(c) **H**: crossover rate (d) **F**: crossover rate

(e) **H**: population size (f) **F**: population size

Figure 3.14: Mean fitness of resulting clusterings on (**H**)ierarchical and (**F**)lat datasets w.r.t. the genetic parameters mutation rate, crossover rate and population size.

Different Population Sizes. We tested the impact of the population size on the quality of the clustering result. We used populations that cover 5, 10, 15, 20 and 25 hierarchical cluster structures in combination with a mutation rate of 0.05 and a crossover rate of 0.15. Figures 3.14(e) and 3.14(f) show again the mean $hMDL$ value over ten runs for each population size on two different datasets. Especially the plot of the hierarchical dataset demonstrates that a higher population size tends to produce better results, which can be explained by the fact that a higher population size provides more variation opportunities whereby a global optimum can be reached easier. However, a large number of chromosomes cause a considerable amount of runtime. One generation using five chromosomes took 2462 ms on average, the computation of a generation on 25 chromosomes took 9,229 ms. Hence we use a population size consisting of ten cluster structures in combination with a mutation rate of 0.05 and $p_{co} = 0.15$ in the following experiments.

Competitive Performance of GACH

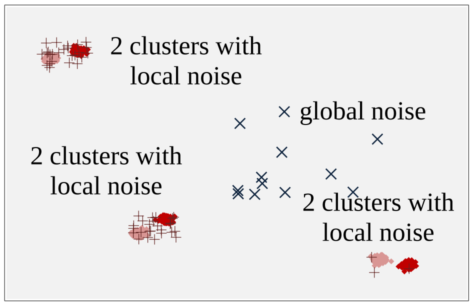
For these experiments we use two different synthetic datasets DS_1 and DS_2. DS_1 is composed of 987 (2d)-data points that form a hierarchy of six clusters surrounded by local and global noise (cf. Figure 3.15(a)). DS_2 consists of 1,950 (2d)-data points that are grouped in three flat strongly overlapping clusters (cf. Figure 3.15(b)). For each dataset, the clustering results were assessed against a ground-truth classification and evaluated the results w.r.t. six different validity measures: the recently proposed information-theoretic methods [77] Normalized Mutual Information (NMI), Adjusted Mutual Information (AMI) and Expected Mutual Information (EMI). All these measures have a maximum value of 1 (which refers to a perfect clustering) and a minimum value of 0. Furthermore, we chose the well-known Precision (Prec) and Recall (Rec) and the DOM [30] value that reflects the coding costs to encode the class labels when the cluster labels are known, i.e. low DOM values represent good clustering results.

Evaluation w.r.t. Dataset $\mathbf{D_1}$. These experiments demonstrate that GACH performs at least as good as the parameter dependent approach OPTICS ($MinPts = 6$, $\epsilon = 0.9$) and ITCH, while being much more accurate than RIC and SL. The reachability plot of OPTICS is provided in Figure 3.15(c), the SL dendrogram is shown in Figure 3.15(e). Although DS_1 seems to be an easy to cluster dataset, SL and RIC fail in assigning the local and global outliers to the correct cluster. Both, GACH and ITCH were able to determine the right cluster structure. However, GACH outperformed ITCH w.r.t. accuracy, as GACH results in a different points to clusters assignment. Therefore, GACH shows the best performance on DS_1 concerning the information-theoretic measures NMI, AMI and EMI. 94% of all data points are assigned to the true cluster and 95% of the cluster contents were detected correctly by GACH. Finally, this result can be coded most efficiently as stated by the low DOM value of 0.4193. This evaluation is summarized in Table 3.5.

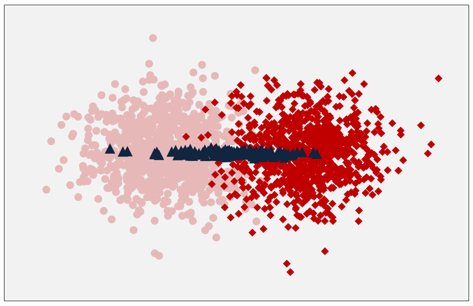
Table 3.5: Performance of GACH on DS_1.

	GACH	ITCH	RIC	OPTICS	SL
NMI	0.9346	0.9265	0.8673	0.9045	0.9110
AMI	0.9159	0.8999	0.7678	0.8662	0.8997
EMI	0.0004	0.0004	0.0004	0.0004	0.0002
PREC	0.9404	0.9222	0.6438	0.8626	0.9555
REC	0.9514	0.9422	0.7720	0.9200	0.9169
DOM	0.4193	0.4454	0.6638	0.4960	0.4765

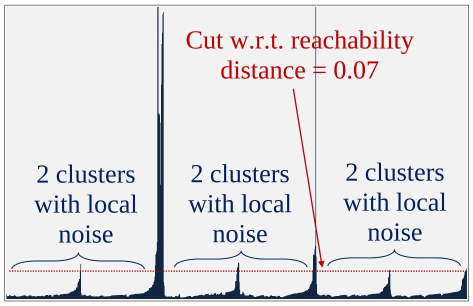
Evaluation w.r.t. Dataset $\mathbf{D_2}$. Neither OPTICS nor SL were able to detect the true cluster structure of DS_2. Both fail because of a massive Single Link effect and therefore the reachability plot provided by OPTICS (cf. Figure 3.15(d)) and the dendrogram produced by SL (cf. Figure 3.15(f)) do not uncover any cluster structure which leads to a clustering where all objects belong to one single cluster. RIC determines only one single cluster. ITCH separates the two red Gaussian clusters but fails in assigning the data points generated by the blue cluster



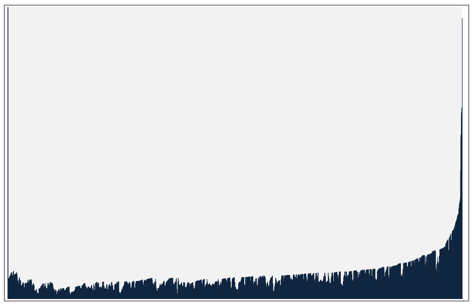
(a) Plotted DS_1.



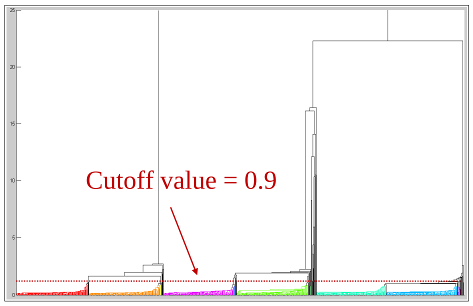
(b) Plotted DS_2.



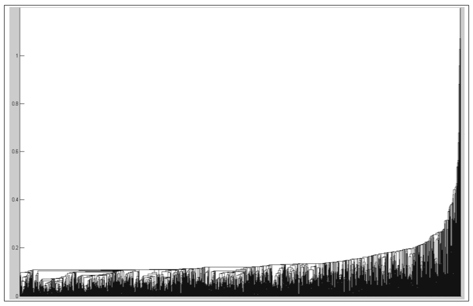
(c) OPTICS on DS_1.



(d) OPTICS on DS_2.



(e) SL on DS_1.



(f) SL on DS_2.

Figure 3.15: Competitive evaluation of GACH on two different synthetic datasets. DS_1 forms a hierarchy including local and global noise, DS_2 is a flat dataset of three overlapping clusters.

correctly. Hence, GACH turned out to be the only algorithm that handles this datasets with strongly overlapping clusters successfully. It shows the best values w.r.t. information-theoretic criterions, while being very accurate. Its result causes only a coding cost of 0.3325 compared to more than 0.9 for almost all other approaches. This is summarized in Table 3.6.

Table 3.6: Performance of GACH on DS_2.

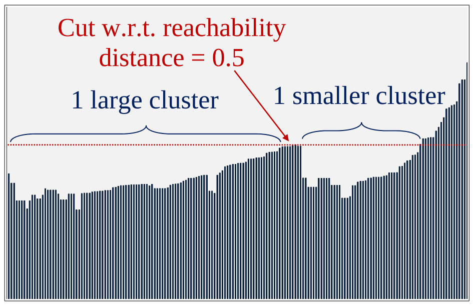
	GACH	ITCH	RIC	OPTICS	SL
NMI	0.6698	0.6316	0.0000	0.0000	0.0000
AMI	0.5877	0.4030	0.0000	0.0000	0.0000
EMI	0.1476	0.2256	0.2379	0.2379	0.2379
PREC	0.9184	0.8227	0.2130	0.2130	0.2130
REC	0.9226	0.8913	0.4615	0.4615	0.4615
DOM	0.3325	0.4226	0.9184	0.9184	0.9184

Application of GACH on Real World Data

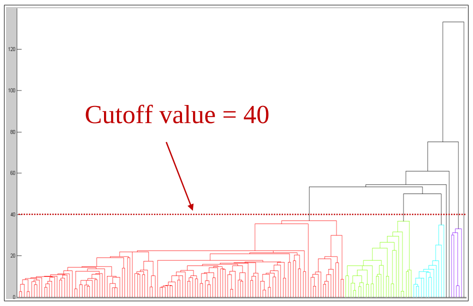
We tested the practical application of GACH on the *Wine* dataset, available at UCI[2]. This dataset contains 178 (13-d)-data objects resulting from a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The dataset provides a ground-truth classification that structures the data into one root node covering the whole dataset and three subclusters defining the three cultivars. GACH determined this structure resulting in high validity values.

The competitors did not even find the right number of clusters. OPTICS, for example detects two clusters (cf. Figure 3.16(a)) and RIC even merges all data points in only one single cluster. The hierarchy of SL covers four disjoint clusters w.r.t. a cutoff value of 40. These results demonstrate that GACH finds a global optimum w.r.t. accuracy in real world data and is therefore applicable in many domains.

[2]`http://archive.ics.uci.edu/ml/datasets/Wine`



(a) OPTICS on *Wine*.



(b) SL on *Wine*.

Figure 3.16: Competitive evaluation of GACH on the real-world *Wine* dataset.

Table 3.7: Performance of GACH on the real-world *Wine* dataset.

	GACH	ITCH	RIC	OPTICS	SL
NMI	0.7886	0.7615	0.0000	0.5079	0.3852
AMI	0.7813	0.6912	0.0000	0.4817	0.3485
EMI	0.0013	0.0014	0.0000	0.0013	0.0018
PREC	0.9401	0.9737	0.1591	0.7466	0.5309
REC	0.9326	0.8596	0.3989	0.6966	0.5337
DOM	0.3631	0.3285	1.1405	0.6853	1.0740

3.2.3 Conclusions

We proposed a genetic algorithm for finding cluster hierarchies, called GACH. GACH combines the benefits of genetic algorithms, information theory and model-based clustering. As GACH uses a MDL-based fitness function, it can be easily applied to real world applications without requiring any expertise about the data. Due to the fact that GACH integrates an EM-like strategy, the content of all clusters is described by an intuitive description. Outliers are assigned to appropriate inner nodes.

Chapter 4

Integrative Data Mining

Integrative mining of heterogeneous data is one of the grand challenges for data mining in the next decade. Recently, the need for integrative data mining techniques has been emphasized in panel discussions, e.g. at KDD 2006 [83], in workshops [109], and in position papers [108, 59]. Integrative data mining is among the top ten challenging problems in data mining identified in [108]. Moreover it is essential for solving many of the other top 10 challenges, including data mining in social networks and data mining for biological and environmental problems.

4.1 INTEGRATE: A Clustering Algorithm for Heterogeneous Data

We address the question of how to find a natural clustering of data with mixed type attributes. In everyday life, huge amounts of such data are collected, for example from credit assessments. The collected data include numerical attributes, such as credit amount and age, as well as categorical attributes, such as personal status and the purpose of the credit. A cluster analysis of credit assessment data is interesting, e.g., for target marketing. We could give numerous more examples for data with mixed type attributes from various applications including medical, bioinformatics and web usage. However, finding a natural clustering of such data is a non-trivial task. We identified two major problems:

- *Problem 1:* Much previous knowledge required.
- *Problem 2:* No adequate support of mixed type attributes.

To cope with these two major problems, we propose INTEGRATE, a parameter-free technique for integrative clustering of data with mixed type attributes. The major benefits of our approach can be summarized as follows:

1. Natural balance of numerical and categorical information in clustering supported by information theory;
2. Parameter-free clustering;
3. Making most effective usage of numerical as well as categorical information;
4. Scalability to large datasets.

4.1.1 Minimum Description Length for Integrative Clustering

Notations. In the following we are considering a dataset DS with n objects. Each object x is represented by d attributes. Attributes are denoted by capital letters and can be either numerical features or categorical variables with two or more values. For a categorical attribute A, we denote a possible value of A by a. The result of our algorithm is a disjoint partitioning of DS into k clusters $C_1, \cdots, C_k$.

Likelihood and Data Compression. One of the most challenging problems in clustering data with mixed type attributes is selecting a suitable distance function, or unifying clustering results obtained on the different representations of the data. Often, the weighting between the different attribute types needs to be specified by parameter settings, cf. Section 2.2. The minimum description length (MDL) principle provides an attractive theoretical foundation for parameter-free integrative clustering avoiding this problem. Recently, the MDL-principle and related ideas have been successfully applied to avoid crucial parameter settings in clustering vector data. For hierarchical data mining we contributed $hMDL$ (cf. Section 3.1), but to the best of our knowledge MDL has not been applied to integrative clustering so far. Also for heterogeneous data, it is very beneficial to regard clustering as a data compression problem, naturally balancing the influence of categorical and numerical attributes.

Coding Categorical Data

To give a simple example, assume we need to transfer 1,000 1-dimensional categorical data objects. Each object is represented by a categorical attribute A with two possible values $\{red, blue\}$. It can be shown that the code length to transfer this data is lower bounded by the entropy of the attribute A. Thus, the coding costs CC of attribute A are provided by:

$$CC(A) = -\sum_{a \in A} p(a) \cdot \log_2 p(a).$$

Here, a denotes each possible value or outcome of the categorical attribute. By the application of the binary logarithm we obtain the code length in bits. If we have no additional knowledge on the data we have to assume that the probabilities for red and $blue$ are both 0.5. With this assumption, we need one bit to encode each data object and the communication costs would be 1,000 bits. Clustering, however, provides high-level knowledge on the data which allows for a much more effective way to reduce the communication costs. Even if the probabilities for the different outcomes of the categorical attributes are approximately equal considering the whole dataset, often different clusters with non-uniform probabilities can be found.

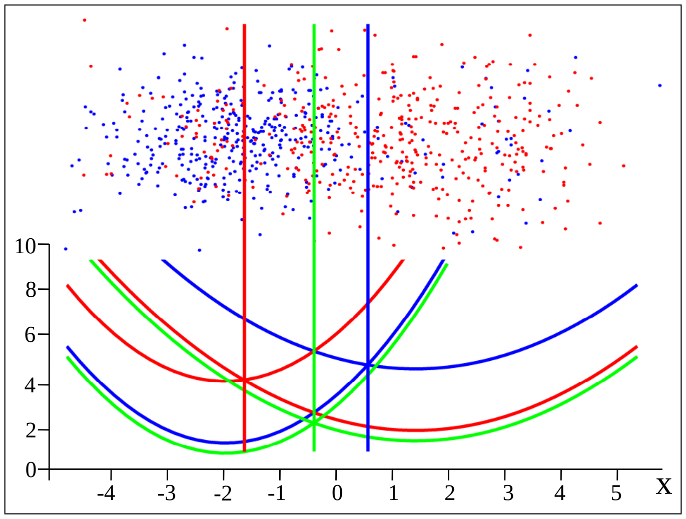


Figure 4.1: Top: Example dataset with two numerical and one categorical attribute with the outcomes *red* and *blue*. Bottom: Cost curves assuming two clusters: Considering the numerical information only (green), integrating numerical and categorical information (red, blue). For each outcome and each cluster, we have a unique cost curve. Intersection points mark the resulting cluster borders.

As an example, refer to the data displayed in Figure 4.1. The data is represented by two numerical attributes that will be taken into account in the next paragraph, and one categorical attribute. The categorical attribute has two possible values, red and $blue$, which are visualized by the corresponding colors. Considering all data, the probabilities for red and $blue$ are both 0.5. However, it is evident that the outcomes red and $blue$ are not uniformly distributed in all areas of the data space. Rather, we have two clusters, one mainly hosts the red objects, and the other the blue ones. In fact, the data has been generated such that in the cluster displayed on the left, we have 88% of blue objects and 12% of red objects. For the cluster on the right, the ratio has been selected reciprocally. This clustering drastically reduces the entropy and therefore the coding costs of the categorical attribute to $CC(A)$ = 0.53 bits per data object, which corresponds to the entropy of A in both clusters. However, we would need to transfer the clustering result itself, including the cluster identifiers to the receiver. Before discussing how to encode the clustering result, we will consider how to encode numerical data.

Coding Numerical Data

If our dataset contains an additional numerical attribute B we can also use the relationship between likelihood and data compressibility to reduce the communication costs. To specify the probability of each data object considering attribute B, we assign a probability density function (PDF) to B. Here, we apply a Gaussian PDF for each numerical attribute. However, let us note that our ideas can be straightforwardly extended to other types PDF, e.g. Laplacian or Generalized Gaussian. Thus, the probability density function of a numerical attribute B is provided by:

$$p(b) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(b-\mu_B)^2}{2\sigma_B^2}\right).$$

If the data distribution of attribute B is Gaussian with mean μ_B and standard deviation σ_B, we minimize the communication costs of the data by a coding scheme which assigns short bit strings to objects with coordinate values that are in the area of high probability and longer bit strings to objects with lower probability according to the idea of Huffman-Coding. Let us note that we do not consider how to actually construct such code. Rather, we are interested in the code length as a measure for clustering quality. The coding costs of attribute B are provided by:

$$CC(B) = -\int p(b) \log_2 \, p(b) \mathbf{d}b.$$

Again, if we have no knowledge on the data, we would have to assume that each attribute is represented by a single Gaussian with mean and standard deviation determined from all data objects. As discussed for categorical data, clustering can often drastically reduce the communication costs. Most importantly, relating clustering to data compression allows us a unified view on data with mixed type attributes. Consider again the data displayed in Figure 4.1. In addition to the categorical attribute we now consider the numerical x-coordinate, denoted by X. To facilitate presentation, we ignore the y-coordinate which is processed analogously. The two green curves at the bottom represent the coding costs of the two clusters considering X. For both curves, the cost minimum coincides with the mean of the Gaussian which generated the data. The cluster on the right has been generated with slightly larger variance, resulting in slightly higher coding costs. The intersection of both cost curves represents the border between the two clusters provided by X, indicated by a green vertical line. In addition, for each cluster and each outcome of the categorical attribute, we have included a cost curve (displayed in the corresponding colors). Again, the intersection points mark the cluster borders provided by the categorical attribute. Consider, e.g., the red vertical line. Red objects with a value in X beyond that point are assigned to the cluster on the right. Thus, in the area between the red and the blue vertical line, the categorical value is the key information for clustering. Note that all borders are not fixed but optimized during the run of our algorithm.

A Coding Scheme for Integrative Clustering

As mentioned, in addition to the coding costs for categorical and numerical attributes of the data objects, we need to elaborate a coding scheme describing the clustering result itself. The additional coding costs for encoding the clustering result can be classified into two categories: the *parameter costs* required to specify the cluster model and the *id-costs* required to specify the cluster-id for each object, i.e. the information to which cluster the object belongs.

For the parameter costs, we focus on the set of objects belonging to a single cluster C. To specify the cluster model, we need for each categorical attribute A to encode the probability of each value or outcome a. For a categorical attribute with $|A|$ possible values, we need to encode $|A|-1$ probabilities since the remaining probability is implicitly specified. For each numerical attribute B we need to encode the parameters μ_B and σ_B of the PDF. Following a central result from the theory of MDL [90], the parameter costs to model the $|C|$ objects of the cluster can be approximated by $p/2 \cdot \log_2 |C|$, where p denotes the number of parameters. The parameter costs depends logarithmically on the number of objects in the cluster. The considerations behind this are that for clusters with few objects, the parameters do not need to be coded with very high precision. To summarize, the parameter costs for a cluster C are provided by:

$$PC(C) = \frac{1}{2} \cdot ((\sum_{A_{cat}} |A| - 1) + |B_{num}| \cdot 2)) \cdot \log_2 |C|.$$

Here A_{cat} stands for all categorical attributes and B_{num} for all numerical attributes in the data. Besides the parameter costs, we need to encode the information to which of the k clusters each object belongs. Also for the id-costs, we apply the principle of Huffman coding which implies that we assign shorter bitstrings to the larger clusters. Thus, the id-costs IDC of a cluster C are provided by:

$$IDC(C) = \log_2 \frac{n}{|C|}.$$

Putting it all together, we are now ready to define $iMDL$, our information-theoretic optimization goal for integrative clustering.

$$iMDL = \sum_{C} (\sum_{A} |C| \cdot CC(A)) + PC(C) + IDC(C).$$

For all clusters C we sum up the coding costs for all numerical and categorical attributes A. We have to add the parameter costs and the id-costs of the cluster to the aforementioned coding costs, denoted by $PC(C)$ and $IDC(C)$, respectively. Finally, we sum up these three terms, coding costs, parameter costs and id-costs for all k clusters.

4.1.2 The Algorithm INTEGRATE

Now we present the highly effective k-means based algorithm INTEGRATE for clustering mixed type attributes that is based on the MDL-based criterion $iMDL$, defined in Section 4.1.1. INTEGRATE is designed to find the optimal clustering of a dataset DS, where each object x comprises both numerical and categorical attributes by optimizing the overall compression rate. First, INTEGRATE builds an initial partitioning of k clusters. Each cluster is represented by a Gaussian PDF in each numerical dimension B with μ_B and σ_B, and a probability for each value of the categorical attributes. All objects are then assigned to the k clusters by minimizing the overall coding costs $iMDL$. In the next step, the parameters of each cluster are recalculated according to the assigned objects. That implies μ and σ in each numerical dimension and the probabilities for each value of the categorical attributes, respectively. After initialization the following steps are performed repeatedly until convergence. First, the costs for coding the actual cluster partition are determined. Second, the assignment of objects to clusters is performed in order to decrease the $iMDL$ value. Third, the new parameters of each cluster are recalculated. INTEGRATE terminates if no further changes of cluster assignments occur. Finally, we receive the optimal clustering for DS represented by k clusters according to minimum coding costs.

Initialization

The effectiveness of an algorithm often heavily depends on the quality of the initialization, as it is the case that many algorithms can get stuck in a local optimum. Hence, we propose an initialization scheme to avoid this effect. We have to find initial cluster representatives that correspond best to the final representatives. An established method for partitioning methods is to initialize with randomly chosen objects of DS. We adopt this idea and take the μ of the numerical attributes of k randomly chosen objects as cluster representatives. During initialization, we set $\sigma = 1.0$ in each numerical dimension. The probabilities of the values for the categorical attributes are set to $\frac{1}{|a|}$. Then a random set of $\frac{1}{z}n$ objects is selected, where n is the size of DS and z = 10 turned out to give satisfying results. Finally, we chose the clustering result that minimizes $iMDL$ best, within m initialization runs. Typically $m = 100$ runs suffice for an effective result. As only a fraction of DS is used for the initialization procedure, our method is not only effective but also very efficient.

Automatically Selecting the Number of Clusters k

Now we propose a further improvement of the effectiveness of INTEGRATE. Using $iMDL$ for mixed type data we can avoid the parameter k. As an optimal clustering that represents the underlying data structure best has minimum coding costs, $iMDL$ can also be used to detect the number of clusters. For this purpose, INTEGRATE uses $iMDL$ no longer exclusively as selection criterion for finding the correct object to cluster assignment. Rather we now estimate the coding costs for each k where k is selected in a range of $1 \leq k \leq n$. For efficiency reasons INTEGRATE performs this iteration step on a $z\%$ sample of DS. The global minimum of this cost function gives the optimal k and thus the optimal number of clusters.

4.1.3 Experimental Evaluation

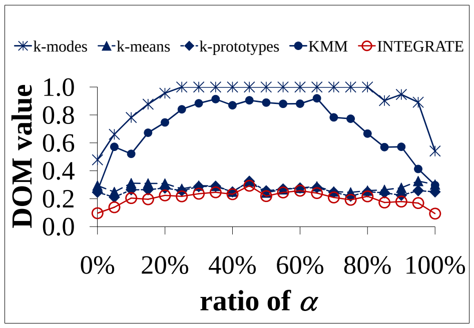
Since INTEGRATE is a hybrid approach combining the benefits of clustering numeric and categorical attributes, we compare to algorithms of both categories and algorithms that can also handle mixed type attributes. In particular, we selected the popular k-means algorithm [72], the widely used method k-modes [49], the k-means-based method by Ahmad and Dey [3] denoted by KMM and k-prototypes [47]. For k-means and k-modes the numerical and categorical attributes were ignored, respectively. For evaluation we used the validity measure by [30] referred to as DOM in the following (smaller values indicate a better cluster quality), which has the advantage that it allows for clusterings with different numbers of clusters and integrates the class labels as “ground truth”. In each experiment, we report the average performance of all clustering algorithms over ten runs.

Synthetic Data

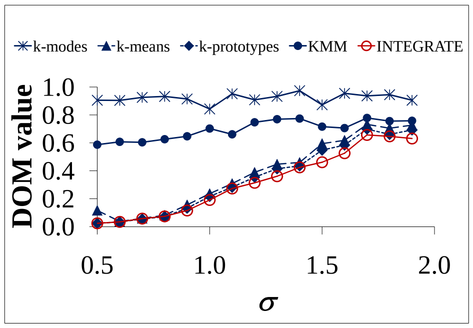
If not specified otherwise, the artificial datasets include three Gaussian clusters with each object having two numerical and one categorical attribute. To validate the results we added a class label to each object which was not used for clustering.

Varying Ratio of Categorical Attribute Values. We generated 3-dimensional synthetic datasets with 1,500 points including two numerical and one two-valued categorical attribute. We varied the ratio for each categorical attribute value from 0% to 100% clusterwise in each dataset. Without need for difficult parameter setting our proposed method performs best in all cases (cf. Figure 4.2(a)). Even in the case of equally (50%) distributed values, where the categorical attribute gives no information for separating the objects, the cluster quality of INTEGRATE is best compared to all other methods. As k-means does not take the categorical attributes into account the performance is relatively constant.

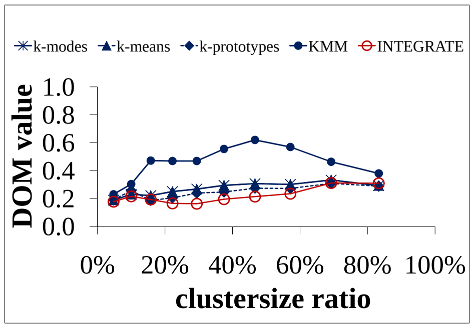
Varying Variance of Clusters. This experiment aims at comparing the performance of the different methods on datasets with varying variances. In particular, we generated synthetic datasets each comprising 1,500 points including two numerical and one two-valued categorical attribute that form three Gaussian clusters with a variance ranging from 0.5 to 2.0. Figure 4.2(b) shows that INTEGRATE outperforms all competitors in all cases, in which each case reflects different degree of overlap of the three clusters. Even at a variance of 2.0 where the numerical attributes carry nearly no cluster information our proposed method shows best cluster quality. In this case the categorical attributes are used to separate the clusters. On the contrary, k-modes performs worst as it can only use the categorical attribute as single source for clustering.



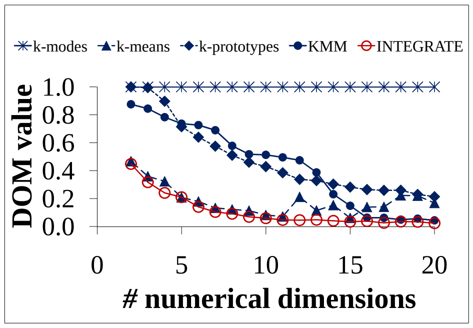
(a) Ratio of categorical attributes.



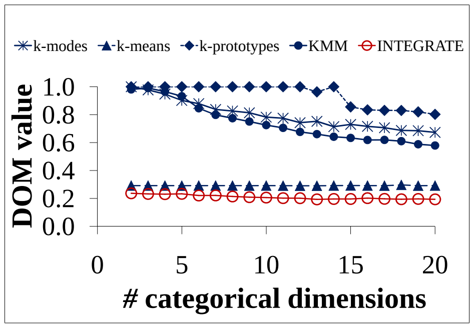
(b) Variance of clusters.



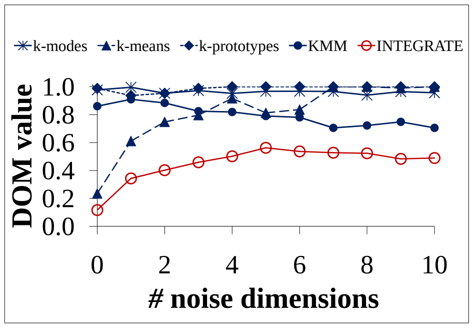
(c) Varying clustersize.



(d) Numerical dimensions.



(e) Categorical dimensions.



(f) Noise dimensions.

Figure 4.2: Mean DOM value of resulting clusterings on synthetic data w.r.t. varying ratio of categorical attribute values, variances of clusters and clustersizes, and different numbers of numerical, categorical and noise dimensions.

Varying Clustersize. In order to test the performance of the different methods on datasets with unbalanced clustersize we generated three Gaussian clusters with different variance and varied the ratio of number of points per cluster from 0% to 100% in five steps. It is obvious from Figure 4.2(c) that INTEGRATE separates the three clusters best even with highly unbalanced cluster sizes. Only in the case of two very small clusters and one big cluster k-modes shows a slightly better cluster validity.

Varying Number of Numerical Dimensions. In this experiment we leave the number of categorical attributes to a constant value and successively add numerical dimensions to each object that are generated with a variance of σ=1.8. INTEGRATE shows best performance in all cases (cf. Figure 4.2(d)). All methods show a slight increase in cluster quality when varying the numerical dimensionality, except k-modes that performs constantly as it does not consider the numerical attributes.

Varying Number of Categorical Dimensions. For each object we added three-valued categorical attributes where we set the probability of the first value to 0.6 and the probability of the two remaining values to 0.2, respectively. Figure 4.2(e) illustrates that our proposed method outperforms the other methods and even k-modes by magnitudes which is a well-known method for clustering categorical data. Whereas KMM shows a heavy decrease in clustering quality in the case of two and four additional categorical attributes, our method performs relatively constant. Taking the numerical attributes not into account the cluster validity of k-means remains constant.

Noise Dimensions. Figure 4.2(f) illustrates the performance of the different methods on noisy data. It is obvious that INTEGRATE outperforms all compared methods when adding non-clustered noise dimensions to the data. k-means shows a

Table 4.1: Performance of INTEGRATE on real-world datasets.

		INTEGRATE	k-MEANS	k-MODES	KMM	k-PROTOTYPES
HEART	μ	1.23	1.33	1.26	1.24	1.33
DISEASE	σ	0.02	0.01	0.03	0.02	0.00
CREDIT	μ	0.61	0.66	0.70	0.63	0.66
APPROVAL	σ	0.03	0.00	0.00	0.09	0.00

highly increase in the DOM values. Even in the case of nine noise dimensions INTEGRATE leads to the best clustering result.

Real Data

Finally, we show the practical application of INTEGRATE on real world data, available at the UCI repository[1]. We chose two different datasets with mixed numerical and categorical attributes. An additional class attribute allows for an evaluation of the results. Table 4.1 reports the μ and σ of the DOM value of all methods within ten runs. For all compared methods we set k to the number of suggested classes.

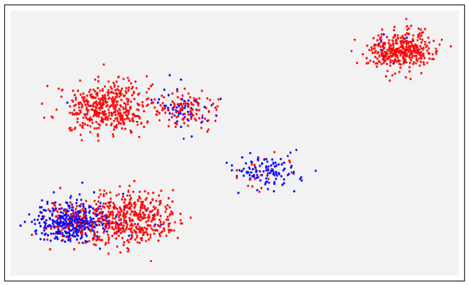
Heart Disease. The Heart-Disease dataset comprises 303 instances with six numerical and eight categorical attributes each labeled to an integer value between 0 and 4 which refers to the presence of heart disease. Without any prior knowledge on the dataset we obtained best cluster validity of 1.23 with INTEGRATE. KMM performed slightly worse. However, the runtime of INTEGRATE is 0.1 seconds compared to KMM which took 2.8 seconds to return the result.

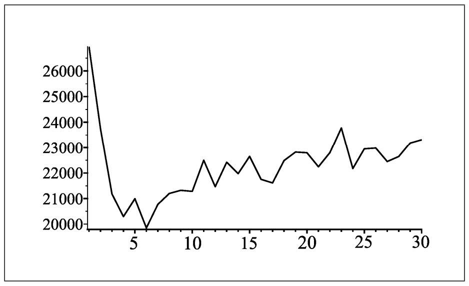
Credit Approval. The Credit Approval dataset contains results of credit card applications. It has 690 instances, each being described by six numerical and nine categorical attributes and classified to the two classes 'yes' or 'no'. With a mean DOM value of 0.61 INTEGRATE separated the objects best into two clusters in only 0.1 seconds without any need for setting input parameters.

[1]`http://archive.ics.uci.edu/ml/`

Finding the optimal k

By means of the dataset illustrated in Figure 4.3(a), we highlight the benefit of INTEGRATE for finding the correct number of clusters. The dataset comprises six Gaussian clusters, where each object has two numerical and one categorical attribute with two different values that are marked in "red" and "blue". Figure 4.3(b) shows the $iMDL$ value of the data model for different values of k. The cost function has its global minimum, which refers to the optimal number of clusters, at $k = 6$. In the range of $1 \leq k \leq 4$ the plotted function shows an intense decrease in the coding costs and for $k > 6$ a slight increase of the coding costs as in these cases the data does not optimally fit into the model and therefore causes high costs. Note, that there is a local minimum at $k = 4$, which would also refer to a meaningful number of clusters.


(a) The dataset.



(b) $iMDL$ for $1 \leq k \leq 30$.

Figure 4.3: Coding costs for different k according to a dataset of six clusters.

4.1.4 Conclusions

We addressed integrative clustering of data covering both numerical and categorical attributes. In addition, we gave in a solution to avoid difficult parameter settings, guided by the idea of data compression. On top of that concept, we have proposed INTEGRATE, an information-theoretic integrative clustering method, that allows for a natural weighting of numerical and categorical information. In addition, INTEGRATE is fully automatic and shows high efficiency and is therefore scalable to large datasets.

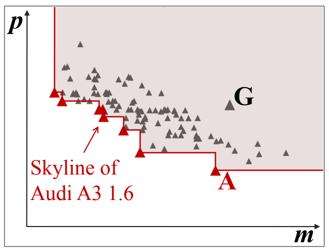
Chapter 5

Mining Skyline Objects

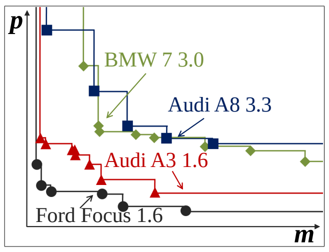
Skyline queries are an important area of current database research, and have gained increasing interest in recent years [19, 93, 58]. Most papers focus on efficient algorithms for the construction of a *single* skyline which is the answer of a user's query. We extend the idea of skylines in such a way that multiple skylines are treated as objects for data exploration and data mining.

5.1 SkyDist: An Effective Similarity Measure for Skylines

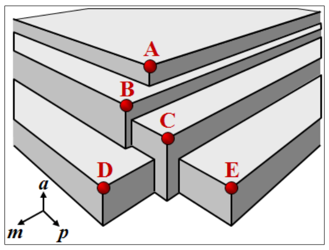
One of the most prominent applications of the skyline operator is to support complex decisions. As an example consider an online marketplace for used cars, where the user wants to find out offers which optimize more than one property of the car such as p (price) and m (mileage), with an unknown weighting of the single conditions. The result of such a query has to contain all offers which may be of interest: not only the cheapest offer and that with lowest mileage but also all offers providing an outstanding combination of p and m. This concept is illustrated in Figure 5.1(a) which displays a set of database objects representing all car offers (lets say for an Audi A3 1.6) described by the attributes p and m. However,



(a) Database objects and their skyline.



(b) Skylines of different car models.



(c) A 3-dimensional skyline.

Figure 5.1: Motivating examples for performing data mining on skylines.

not all of these offers are equally attractive to the user. For instance, offer A has both a lower price *and* a lower mileage than G and many other offers. Therefore, we say that A *dominates* G (in symbols $A \prec G$), because A is better than G w.r.t. any possible weighting of the criteria price and mileage. The *skyline* contains all objects which are *not dominated* by any other object in the database. In Figure 5.1(a), the skyline objects exhibiting an outstanding combination of price and mileage are highlighted in red.

For the used car market, the skyline of each car model has a particular meaning: Many arbitrarily bad offers may be present in the database but only the offers in (or close to) the skyline have a high potential to find a customer. The skyline of the offers marks to some degree the fair value of a car for each mileage in the market. Therefore, the skyline characterizes a car model (or higher-order objects of other applications) in a highly expressive way.

This high expressiveness leads us to the idea of treating the skylines themselves as objects with the corresponding similarity measure, called *SkyDist*. Figure 5.1(b) illustrates the skylines of four different car models derived from an online automotive market. Car models which exhibit similar skylines (like Audi A3 1.6 and Ford Focus 1.6) may be considered as similar: A recommender system might find out that the Focus is a perfect alternative to the Audi A3. The different car models may be subject to clustering, classification, outlier detection, or other supervised or unsupervised data mining tasks, using a similarity measure which is

built upon the skyline.
Since the different 2-dimensional skylines in Figure 5.1(b) look like well-known structures (like time series or trajectories), we may gain the impression that appropriate techniques could be helpful for the definition of the skyline similarity. However, when taking a closer look at a skyline in a 3- or higher dimensional data space as shown in Figure 5.1(c) with the attributes p (price), m (mileage), and a (age), it becomes obvious that this is not sufficient. Unfortunately, there exists no natural dimension which can be used for ordering the skyline objects. Therefore, our solution is based on a different concept, namely the symmetric volume difference of the two skylines to be compared.

5.1.1 SkyDist

At the beginning of this section we define the essential concepts underlying SkyDist in general. Then we motivate the idea behind assigning SkyDist for 2-dimensional skylines and conclude with a generalization according d-dimensional skylines, where $d \geq 2$.

Theoretical Background

Consider a set of database objects, each of which is already associated with an individual skyline. For instance, each car type is associated with the skyline of the offers posted in a used car market. An effective distance measure for a pair of skyline objects should be useful in the sense that the characteristic properties of the skyline concept are suitably reflected. Whenever two skylines are similar in an intuitive sense, then SkyDist should yield a small value. In order to define such a reasonable distance measure, we recall here the central concept of the classical skyline operator, the *dominance relation* which can be built on the preferences on attributes $\mathcal{D}_1, \ldots, \mathcal{D}_d$ in a d-dimensional numeric space $\mathcal{D}$.

Definition 4 (Dominance) *For two data points u and v, u* ***dominates*** *v, denoted by $u \prec v$, if the following two conditions hold:*

$\forall$ dimensions $\mathcal{D}_{i\in\{1,\ldots,d\}}$: $u.\mathcal{D}_i \leq v.\mathcal{D}_i$
$\exists$ dimension $\mathcal{D}_{j\in\{1,\ldots,d\}}$: $u.\mathcal{D}_j < v.\mathcal{D}_j$.

Definition 5 (Skyline Point / Skyline) *Given a set of data points DS, a data point u is a **skyline point** if there exists no other data point $v \in DS$ such that v dominates u. The **skyline** on DS is set of size s containing all skyline points.*

Two skylines are obviously *equal* when they consist of identical points. Intuitively, one can say that two skylines are similar, whenever they consist of similar points. But in addition, two skylines may also be considered similar if they dominate approximately the same points in the data space. This can be grasped in a simple and efficient way by requiring that the one of the two skylines should not change much whenever the points of the other skyline are inserted into the first one and vice versa. This leads us to the idea to base SkyDist on the set-theoretic difference among the parts of the data space which are dominated by the two skylines. For a more formal view let us define the terms *dominance region* and *non-dominance region* of a skyline.

Definition 6 (Dominance Region of a Skyline Point) *Given a skyline point $x_{i\in\{1,\ldots,s\}}$ of a skyline $X = \{x_1, \ldots, x_s\}$. The **dominance region of** x_i, denoted by $\mathcal{DOM}_{x_i}$, is the data space, where every data point $u \in \mathcal{DOM}_{x_i}$ complies the condition $x_i \prec u$.*

Definition 7 (Dominance Region of a Skyline) *Given the set of all skyline points of a skyline X. The **dominance region of the skyline** X, denoted by $\mathcal{DOM}_X$, is defined over the union of the dominance regions of all skyline points $x_{i\in\{1,\ldots,s\}}$.*

Figure 5.2 illustrates this notion for two given skylines X and Y. The green and blue areas show the dominance regions of skylines X and Y, respectively.

Definition 8 (Non-Dominance Region of a Skyline) *Given a numeric space $\mathcal{D} = (\mathcal{D}_1, \ldots, \mathcal{D}_d)$ and the dominance region $\mathcal{DOM}_X$ of a skyline X. The **non-dominance region of** X, denoted by $\overline{\mathcal{DOM}_X}$, is $\mathcal{D} \setminus \mathcal{DOM}_X$.*

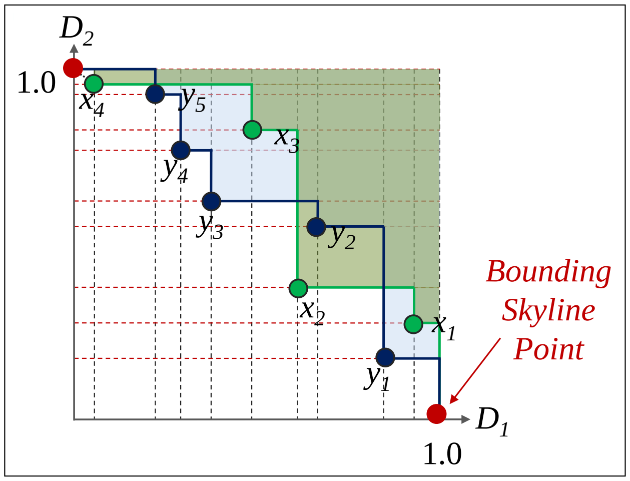


Figure 5.2: Dominance regions of two skylines X and Y.

The basic idea behind SkyDist is to determine the data space that is represented by all possible data points that are located in the dominance region of one skyline and, at the same time, the non-dominance region of the other skyline. More formally we can say that SkyDist is the symmetric volume difference between two skylines which can be specified by the following equation.

$$SkyDist(X, Y) = \text{Vol}((\mathcal{DOM}_X \setminus \mathcal{DOM}_Y) \cup (\mathcal{DOM}_Y \setminus \mathcal{DOM}_X))$$

In order to determine the value of the distance of two skylines X and Y based on the concept described above, we have to limit the corresponding regions in each dimension. Therefore, we introduce the notion of a bounding skyline point.

Definition 9 (Bounding Skyline Point) *Given a skyline X in a d-dimensional numeric space $\mathcal{D} = (\mathcal{D}_1, \ldots, \mathcal{D}_d)$, where $x_i \in [0, 1]$. The **bounding skyline point** of skyline X in dimension i, denoted by x_{Bound_i}, is defined as follows.*

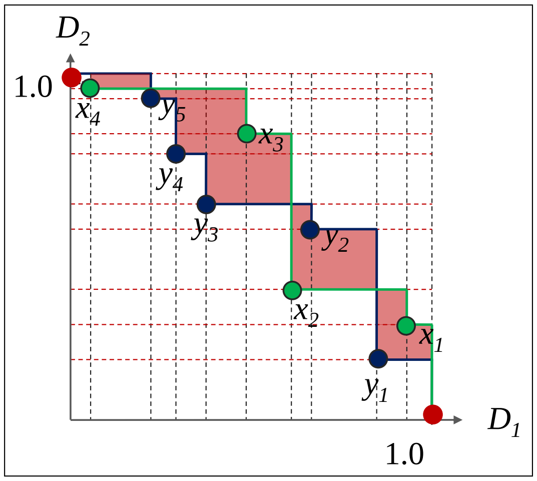


Figure 5.3: SkyDist in 2-dimensional space.

$$x_{Bound_i}.\mathcal{D}_j = \begin{cases} 1, & \mathit{if}\; j = i \\ 0, & \mathit{otherwise} \end{cases}$$

The bounding skyline points for two 2-dimensional skyline objects X and Y are marked in Figure 5.2 in red color. We remark that this concept is also applicable if the domain of one or more dimensions is not bounded. In this case the affected dimensions are bounded by the highest value that occurs in either skyline object X or Y in the according dimension. The coordinates of the remaining skyline points are then scaled respectively.

SkyDist by Monte-Carlo Sampling

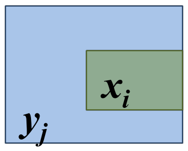
First we want to give an approximation of the distance of two skylines by a Monte-Carlo Sampling approach (MCSkyDist). SkyDist is approximated by randomly sampling points and computing the ratio between samples that fall into the region defined by Equation to compute SkyDist and the ones that do not. Let us consider Figure 5.3. The region marked in red illustrates the region underlying SkyDist of two Skylines X and Y. This region is the dominance region of skyline X and simultaneously the non-dominance region of skyline Y, thus the distance of skyline X to Y. A user defined number of points is randomly sampled and the amount of samples that fall into the SkyDist region is determined. The ratio between the samples located in the SkyDist region and the ones that do not give an approximation of the distance of the two skylines. We use this technique in our experiments as a baseline for comparing a more sophisticated approach.

SkyDist for 2-Dimensional Skylines

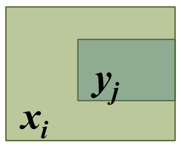
Now we describe how to compute the exact distance of two skylines X and Y in 2-dimensional space. Let the skyline points of both skylines X and Y be ordered by one dimension, e.g. $\mathcal{D}_2$. Note that the dominance region of a skyline X is composed by the dominance regions of all of its skyline points. For the computation of SkyDist, we consider only the dominance regions that are exclusive for each skyline point. Meaning that when we look at a particular skyline point x_i, we assign the dominance region of x_i and discard all dominance regions of skyline points x_j, where $x_j.\mathcal{D}_2 > x_i.\mathcal{D}_2$. In the 2-dimensional space, nine different cases can occur during the computation of SkyDist according to X and another skyline object Y (cf. Figure 5.4). Note that some cases are symmetric and can thus be treated in a similar way.

- **Case 1**: $x_i.\mathcal{D}_1 > y_j.\mathcal{D}_1 \wedge x_i.\mathcal{D}_2 > y_j.\mathcal{D}_2 \wedge x_{i+1}.\mathcal{D}_2 \leq y_{j+1}.\mathcal{D}_2$
- **Case 2**: $x_i.\mathcal{D}_1 \leq y_j.\mathcal{D}_1 \wedge x_i.\mathcal{D}_2 < y_j.\mathcal{D}_2 \wedge x_{i+1}.\mathcal{D}_2 \geq y_{j+1}.\mathcal{D}_2$
- **Case 3**: $y_j.\mathcal{D}_2 \leq x_i.\mathcal{D}_2 < y_{j+1}.\mathcal{D}_2 \vee x_i.\mathcal{D}_2 \leq y_j.\mathcal{D}_2 < x_{i+1}.\mathcal{D}_2$
  - **Case 3.1**: $y_j.\mathcal{D}_2 \leq x_i.\mathcal{D}_2 < y_{j+1}.\mathcal{D}_2 \wedge x_i.\mathcal{D}_1 \leq y_j.\mathcal{D}_1$
  - **Case 3.2**: $y_j.\mathcal{D}_2 \leq x_i.\mathcal{D}_2 < y_{j+1}.\mathcal{D}_2 \wedge x_i.\mathcal{D}_1 > y_j.\mathcal{D}_1$
  - **Case 3.3**: $x_i.\mathcal{D}_2 \leq y_j.\mathcal{D}_2 < x_{i+1}.\mathcal{D}_2 \wedge x_i.\mathcal{D}_1 \leq y_j.\mathcal{D}_1$
  - **Case 3.4**: $x_i.\mathcal{D}_2 \leq y_j.\mathcal{D}_2 < x_{i+1}.\mathcal{D}_2 \wedge x_i.\mathcal{D}_1 > y_j.\mathcal{D}_1$

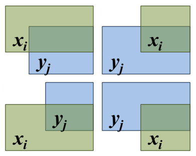
- **Case 4**: $x_i.\mathcal{D}_1 > y_j.\mathcal{D}_1 \wedge x_i.\mathcal{D}_2 < y_j.\mathcal{D}_2 \wedge x_{i+1}.\mathcal{D}_2 \geq y_{j+1}.\mathcal{D}_2$
- **Case 5**: $x_i.\mathcal{D}_1 \leq y_j.\mathcal{D}_1 \wedge x_i.\mathcal{D}_2 > y_j.\mathcal{D}_2 \wedge x_{i+1}.\mathcal{D}_2 \leq y_{j+1}.\mathcal{D}_2$
- **Case 6**: $x_{i+1}.\mathcal{D}_2 \leq y_j.\mathcal{D}_2 \vee x_i.\mathcal{D}_2 \geq y_{j+1}.\mathcal{D}_2$



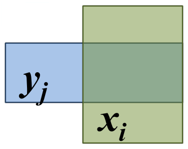
(a) Case 1.



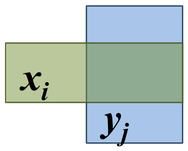
(b) Case 2.



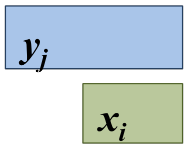
(c) Case 3.



(d) Case 4.



(e) Case 5.



(f) Case 6.

Figure 5.4: Different cases that occur during the computation of SkyDist in 2-dimensional space.

The following formula computes the corresponding volume of the region referring to SkyDist of two skyline objects $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$. Both skylines include their bounding skyline points.

$$
\begin{aligned}
&SkyDist(X,Y) = \\
&= \text{Vol}\Bigg(\sum_{i=1}^{n-1}\sum_{j=1}^{m-1} |1 - x_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)| - \\
&- \begin{cases}
|1 - x_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 1} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 2} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 3.1} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 3.2} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 3.3} + \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 3.4} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - y_{j+1}.\mathcal{D}_2) - (1 - y_j.\mathcal{D}_2)|, & \text{if case 4} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - x_{i+1}.\mathcal{D}_2) - (1 - x_i.\mathcal{D}_2)|, & \text{if case 5} \\
0, & \text{if case 6}
\end{cases} \\
&+ \sum_{j=1}^{m-1}\sum_{i=1}^{n-1} |1 - y_j.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)| - \\
&- \begin{cases}
|1 - y_j.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 1} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 2} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 3.1} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - y_j.\mathcal{D}_2) - (1 - x_{i+1}.\mathcal{D}_2)|, & \text{if case 3.2} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 3.3} \\
|1 - y_i.\mathcal{D}_1| \cdot |(1 - x_i.\mathcal{D}_2) - (1 - y_{j+1}.\mathcal{D}_2)|, & \text{if case 3.4} \\
|1 - y_j.\mathcal{D}_1| \cdot |(1 - x_{i+1}.\mathcal{D}_2) - (1 - x_i.\mathcal{D}_2)|, & \text{if case 4} \\
|1 - x_i.\mathcal{D}_1| \cdot |(1 - y_{j+1}.\mathcal{D}_2) - (1 - y_j.\mathcal{D}_2)|, & \text{if case 5} \\
0, & \text{if case 6}\Bigg)
\end{cases}
\end{aligned}
$$

Figure 5.3 illustrates the computation of SkyDist according skyline objects X and Y (cf. Figure 5.2). The calculation is implemented via a sweep-line algorithm. Therefore we refer to SLSkyDist in the following. This sort of algorithms use a conceptual sweep-line to assess the distance of the two skylines. For this purpose we store the skyline points of both skylines X and Y in a heap structure, called event point schedule (EPS), ordered by one of the two dimensions (e.g. $\mathcal{D}_2$) ascending. The idea behind the sweep-line algorithm is to imagine that a line is swept across the plane, stopping at every point that is stored in the EPS. In our example, the sweep line moves along the axis of $\mathcal{D}_2$.

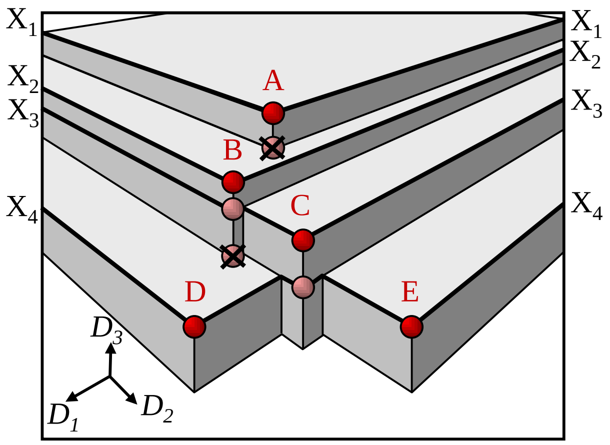


Figure 5.5: A sequence of 2-dimensional skylines, forming a 3-dimensional skyline.

A Plane-Sweep Approach for the High-dimensional Case

The idea behind algorithms following the plane-sweep paradigm, which originates from the field of computational geometry, is to imagine that a line (often a vertical line) is swept or moved across the plane, stopping at some points. The general approach of these algorithms then is to transform a d-dimensional problem into a sequence of problems of dimensionality $(d-1)$, which are then solved recursively until the dimensionality is small enough to apply an explicit solution (in our case $d = 2$). For skyline-based problems the plane-sweep paradigm is often very adequate because we can consider a d-dimensional skyline as a sequence of skylines with dimensionality $(d - 1)$, as can be seen from Figure 5.5: Here, we use $\mathcal{D}_3$ (in decreasing order) as the ordering dimension and build a sequence of 2-dimensional skylines (each in the $\mathcal{D}_1/\mathcal{D}_2$-plane). We first describe how a single skyline is processed according to the plane-sweep paradigm (assuming no particular computational task to be applied while traversing), and then we demonstrate, how two skylines can be traversed simultaneously in order to compute their SkyDist.

Traversal. Plane-sweep algorithms basically use two different data structures. First, the event point schedule (EPS) contains all points, ordered by the last coordinate (decreasingly). Second, in each stop of the sweeping plane, we want to obtain a valid skyline in the space spanned by the remaining coordinates. This sub-skyline is stored in the sweep-plane status (SPS). In Figure 5.5, this refers to the four sub-skylines $X_1, ..., X_4$. More precisely, in each stop of the sweep plane, this $(d-1)$-dimensional sub-skyline is updated by (1) projecting the current point of the EPS into the $(d - 1)$-dimensional space, (2) inserting the projected point into the skyline in the SPS, (3) deleting those points from the skyline in the SPS which are dominated by the new point in the $(d - 1)$-dimensional space, and (4) calling the traversal-algorithm recursively for the obtained $(d - 1)$-dimensional skyline. In our example, the EPS has the order (A, B, C, D, E). We start with an empty SPS, and at the first stopping point, the $\mathcal{D}_1/\mathcal{D}_2$-projection of A is inserted into the SPS to obtain X_1. No point is dominated. Hence, we call the traversal algorithm for the obtained 2-dimensional skyline (A). At the next stop, the projection of B is inserted. Since the projection of B dominates the projection of A (in our figure this fact is symbolized by the canceled-out copy of A), X_2 contains only point B. After the recursive call, in the next stop, the projection of C is inserted which does not dominate object B in the skyline, and, therefore, the next skyline $X_3 = (B, C)$. Finally, D and E are inserted into the skyline in the SPS (which can be done in one single stop of the sweep-plane or in two separate stops in any arbitrary order), to obtain $X_4 = (D, C, E)$, since B is dominated by D in the $(d-1)$-dimensional projection.

Simultaneous Traversal for SkyDist. The computation of SkyDist(X, Y) requires a simultaneous traversal of the two skylines X and Y to be compared. That means that the EPS contains the points of both X and Y, simultaneously ordered by the last coordinate. Each of the stops of the sweep-plane corresponds either to a point of X or a point of Y, and the corresponding sub-skyline in the SPS must be updated accordingly, as defined in the last paragraph by (1) projecting the point, (2) inserting the point in the corresponding sub-skyline for either X_i or Y_i, (3) canceling out the dominated points of the sub-skyline, and finally making the recursive call for $(d-1)$.

Having developed this algorithmic template, we can easily obtain SkyDist(X, Y), because each stop of the sweep-plane defines a disklet, the thickness of which is given by the difference between the last and the current event point (taking only the coordinate which is used to order the EPS). This thickness must be multiplied with the $(d-1)$-dimensional volume which is returned by the recursive call that computes SkyDist of the $(d-1)$-dimensional sub-skylines X_i and Y_i. All volumes of all disklets of the obtained sub-skylines have to be added. This works no matter whether current and the previous event points belong to the same or different skylines. Also in the case of tie situations where both skylines have identical values in the ordering coordinates or where some points in the same skyline have identical values, our algorithm works correctly. In this case, some disklets with thickness 0 are added to the overall volume, and, therefore, the order in which the corresponding sub-skylines are updated, does not change anything.

Table 5.1: Runtime analysis for SLSkyDist and MCSkyDist.

	# DATA POINTS	# SKYLINE POINTS OF X	# SKYLINE POINTS OF Y	SLSKYDIST	MCSKYDIST
2D	10	4	3	15 ms	47 ms
	100	5	6	16 ms	47 ms
	1000	9	5	15 ms	63 ms
	10000	7	12	15 ms	78 ms
3D	10	5	5	31 ms	62 ms
	100	18	16	47 ms	109 ms
	1000	29	19	63 ms	125 ms
	10000	68	39	78 ms	266 ms
4D	10	5	8	47 ms	78 ms
	100	25	36	172 ms	141 ms
	1000	80	61	203 ms	297 ms

5.1.2 Experimental Evaluation

We evaluated SkyDist on synthetic and real world data. SkyDist turned out to be highly effective and efficient. Hence, data mining on skylines provides novel insights in various application domains. For skyline construction, the approach of [19] is applied.

Stability and Efficiency

To evaluate the stability of the baseline approach MCSkyDist, we generated synthetic data covering 1,000 uniformly distributed 2-dimensional data objects and varied the sample rate in a range of 1 to 50,000 and quantify the accuracy of SkyDist in each run. These experiments indicate that the accuracy of MCSkyDist is very robust w.r.t. the number of samples. Its results achieve a constant value even with a small sample rate. Actually with 1,000 samples the same result as using SLSkyDist can be achieved.

Furthermore, we generated skylines for datasets with varying number of points and dimensionality. As in most real world applications, we are interested in the skyline w.r.t. only a few selected dimensions, we focus on skylines up to dimensionality $d = 4$ here. The results are summarized in Table 5.1. In the 2-dimensional case the runtime of SkyDist remains constant even with increasing dataset size. This is due to the fact, the number of skyline points remains relatively constant even though the size increases. It has been shown in [68] that for independent distributed data the number of skyline objects is $O(\log_2 n)$. Also in the 3- and 4-dimensional case it is evident that considering the skyline points instead of all data points is very efficient. It takes 78 and 266 ms for SLSkyDist and MCSkyDist (sample rate: 1,000) respectively to return the result for two skylines X and Y each determined out of 10,000 data points. MCSkyDist and SLSkyDist both scale linear with increasing dimensionality. SLSkyDist outperforms MC-SkyDist by a factor of two which confirms the efficacy and scalability of an exact computation of SkyDist even for large datasets with more than two dimensions.

Clustering Skylines of Real World Data

To demonstrate the potential of SkyDist for data mining, we demonstrate that interesting reasonable knowledge can be obtained by clustering skylines with SkyDist. In particular, we focus on two case studies from different applications and we supply three different clustering algorithms (PAM [56], Single Link (SL) [72] and DBSCAN [31]) with SkyDist.

Case Study 1: Automotive Market. The data used in this experiment is obtained from the online automotive market place[1]. The resulting dataset comprises in total 1,519 used cars constructed in the year 2000. Thus, each data point represents a specific offer of a used car which are labeled to one of three classes *compact*, *medium-sized* and *luxury*, respectively. This information allows for an evaluation of the results. Each car model is represented by one 2-dimensional skyline using the attributes $mileage$ and $price$. Hence each skyline point represents a specific offer that provides an outstanding combination of these attributes, and therefore it

[1]`http://www.autoscout24.de`

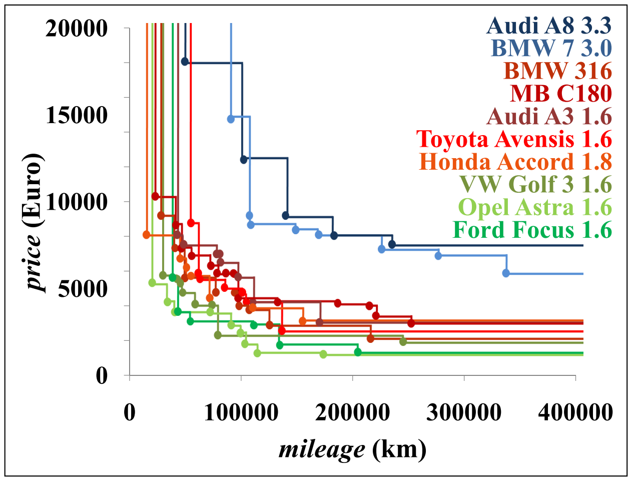
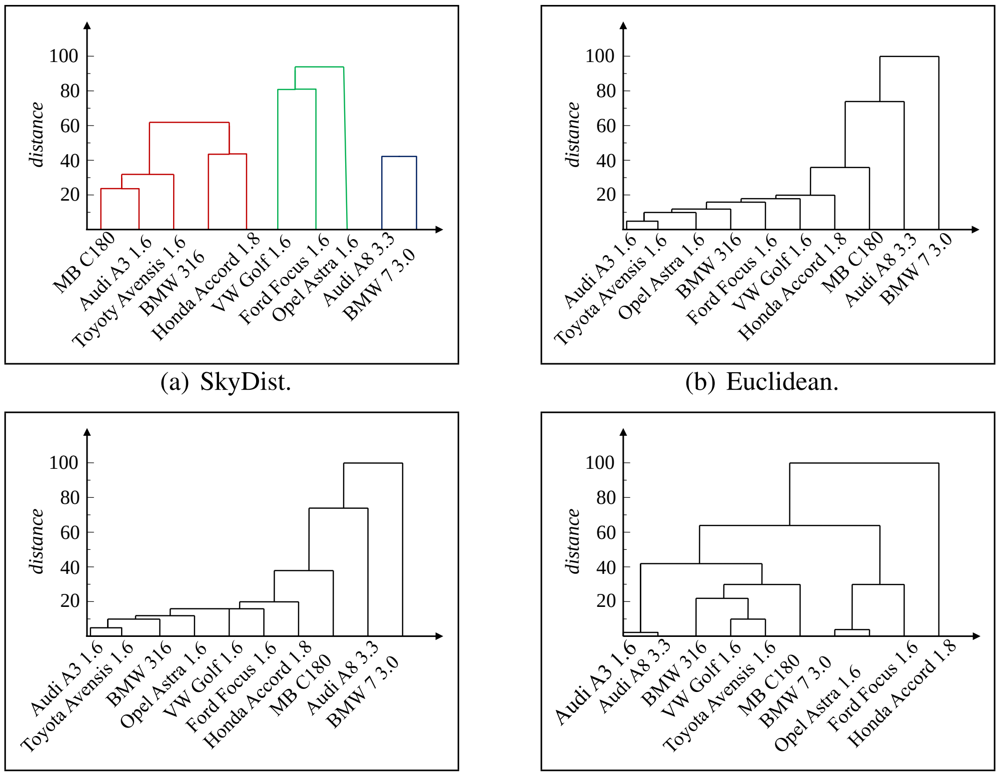


Figure 5.6: Clustering of car models represented by their skyline.

is interesting for the user. PAM, DBSCAN and SL combined with SkyDist create an identical clustering result (cf. Figure 5.6) using the following parameterization: PAM ($k = 3$), DBSCAN ($\epsilon = 10$, $MinPts = 2$) and the SL dendrogram with a vertical cut at $maxDistance = 90$. All algorithms produce 100% class-pure clusters w.r.t. the labeling provided by the online market place. As expected the Audi A8 and BMW 7 of class *luxury* are clustered together in the blue cluster with a very low distance. Also the Mercedes Benz C180 (MB C180) and the Audi A3 belong to a common cluster (marked in red) and show a larger distance to the Toyota Avensis or the Honda Accord. These two car models are clustered together in the green cluster, whose members usually have a cheaper price. The clustering result with SkyDist is very informative for a user interested in outstanding combinations of $mileage$ and $price$ but not fixed on a specific car model. By clustering, groups of models with similar skylines become evident.

SkyDist vs. Conventional Metrics. Conventional metrics can in principle be applied for clustering skylines. For this purpose we represent each car model as a vector by calculating the average of the skyline points of the respective car model



(a) SkyDist. (b) Euclidean.

(c) Manhattan. (d) Cosine.

Figure 5.7: SL dendrogram using SkyDist in comparison to conventional metrics.

in each dimension. Then we cluster the resulting vectors with SL using the Euclidean, Manhattan and Cosine distance. In contrast to clustering skylines using SkyDist (cf. Figure 5.7(a)), Figures 5.7(b), 5.7(c) and 5.7(d) demonstrates that no clear clusters are identifiable in the dendrogram and the result is not very comprehensible. In Figure 5.7(b) and 5.7(c) it can easily be seen that Euclidean and Manhattan distance lead to similar results. Both show the so called Single Link effect, where no clear clusters can be identified. Using the Cosine distance avoids this effect but does not produce meaningful clusters either. Precisely, the luxury car model BMW 7 has minimum distance to the Opel Astra of class *compact* but has an unexpected high distance to the luxury Audi A8. Furthermore the Honda

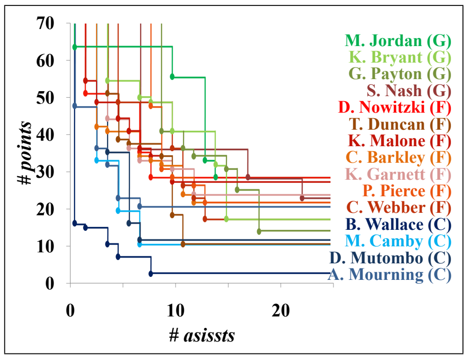
Accord is treated as an outlier. When comparing the results produced by using the conventional metrics to the clusterings using SkyDist it becomes evident, that SkyDist outperforms the other metrics and is much better suited for skylines comparison.

Case Study 2: Performance Statistics of Basketball Players. The NBA game-by-game technical statistics are available online[2]. We focus on the years 1991 to 2005. To facilitate demonstration and interpretation, we select players who have played at minimum 500 games and are noted for their skills and got various awards. The players are labeled with the three different basketball positions *guard* (G), *forward* (F) and *center* (C). The individual performance skyline of each player represents the number of assists and points. We cluster the skylines using SkyDist. SL with a vertical cut of the dendrogram at $maxDistance = 96$ and DBSCAN ($\epsilon = 4$, $MinPts = 2$) result in the same clustering. Figure 5.8(b) shows that the players cluster very well in three clusters that refer to the labels G, F and C. With PAM ($k = 3$) (cf. Figure 5.8(a)) only Steve Nash (G) clusters into the red *forward* cluster. This can be explained by the fact, that this player has performance statistics as a player in the position forward concerning number of points and assists.

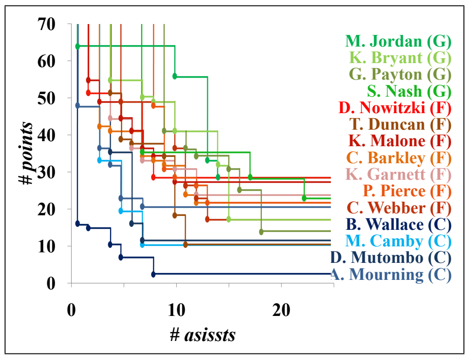
5.1.3 Conclusions

With SkyDist, we presented one of the first approaches for data mining on skyline objects. Inspired by the success of the skyline operator for multi-criteria decision making, we demonstrated that the skyline of a dataset is a very useful representation capturing the most interesting characteristics. Hence, data mining on skylines is very promising for knowledge discovery. We presented a baseline approach to compute SkyDist that approximates the distance and an exact sweep-line based computation. SkyDist can easily be integrated into many data mining techniques. Real world case studies on clustering skylines demonstrated that data mining on

[2]`http://www.NBA.com`



(a) PAM.



(b) SL, DBSCAN.

Figure 5.8: Clustering of NBA Players represented as skylines.

skylines enabled by SkyDist yields interesting novel knowledge. Moreover, SkyDist is efficient and thus scalable to large datasets.

Chapter 6

Graph Mining

Graphs or networks are very general data structures. This is interesting for two reasons: First, from a system-wide perspective, a graph represents a system and the interactions between its components. Second, from a component-centered point of view, a graph describes all relationships that link this object to the rest of the system. Hence, graphs and networks provide advantages in multiple application domains.

The influence of chemical compounds on physiological processes, e.g. toxicity, or effectiveness as a drug and features of molecular structures are studied in the fields of chemoinformatics [36] and bioinformatics. Furthermore a high number of structured data comprises graph models of molecular structures, from RNA to proteins [7], and of networks, which include protein-protein interaction networks [106], metabolic networks [54], regulatory networks [24], and phylogenetic networks [51]. Another important source of graph structured data is social network analysis [101]. Here, nodes represent individuals and edges stand for interactions between them. Psychologists often want to study the complex social dynamics between humans, and medical scientist want to uncover the spreading behaviour of a virus. Industry aims for analyzing these networks for marketing purposes, as detecting influential individuals, often referred to as 'key-players' or 'trend-setters', is relevant for marketing, as companies could then focus their advertising efforts on persons known to influence the behavior of a larger group of people. A further application area for graph models is the internet which is a network and hence a graph itself. HTML documents are the nodes, and hyperlinks connect these nodes. In fact, Google exploits the Internet structure in its famous PageRank algorithm [71] for ranking websites.

In this chapter, we first address the question of finding patterns in brain networks of patients with medical dysfunctions in Section 6.1. In Section 6.2, we perform frequent subgraph discovery in dynamic biological networks.

6.1 Frequent Subgraph Mining in Brain Network Data

Understanding the mechanisms that govern neural processes in the brain is an important topic for medical studies. Recently the emergence of an increasing number of clinical diagnostics enable a global analysis of complex processes and therefore allow a better understanding of numerous diseases. We focus on the mechanisms in the brain that are associated with somatoform pain disorder. Somatization disorders constitute a large, clinically important health care problem that urgently needs deeper insight [103].

Earlier studies have revealed that different subunits within the human brain interact among each other, when particular stimuli are transmitted to the brain. Hence the different components form a network. From an algorithmic point of view, interacting subunits act as specific subgraphs within the network. From a medical perspective, it is interesting to ask to which degree interactions of subunits are correlated with medical disorders.

We study the question whether brain compartments of patients with somatoform pain disorder form subgraphs that differ from brain compartments of subjects that do not suffer from this disease. For this purpose, we analyze task-fMRI scans (a special form of neuroimaging) of the brain of ten subjects, six patients with somatoform pain disorder and four healthy controls that attended the study of [41].

In this study both groups underwent alternate phases of non-pain and pain stimuli during the fMRI scanning. We construct a brain co-activation network for each subject where each node represents a voxel in the fMRI image. Voxels are grouped together in 90 so-called regions of interest (ROIs) using the template of [96]. We apply two approaches, the efficient heuristic approach GREW [63] and the exhaustive sampling technique FANMOD [102] to uncover frequent subgraphs in each of these networks. While a heuristic approach allows for finding subgraphs of arbitrary size, an exhaustive sampling algorithm can be applied to find *all* frequent subgraphs, but in this case we are limited in the size of the subgraphs. Both algorithms are designed to operate on one large graph and to find patterns corresponding to connected subgraphs that have a large number of vertex-disjoint embeddings. We demonstrate that patients with somatoform disorder show activation patterns in the brain that are different from those of healthy subjects.

6.1.1 Performing Graph Mining on Brain Network Data

Basics of Graph Theory

We start with a brief summary of necessary definitions from the field of graph mining.

Definition 10 (Labeled Graph / Network) *A* **labeled graph** *is represented by a 4-tuple $G = (V, E, L, l)$, where V is a set of vertices (i.e. nodes), $E \subseteq V \times V$ is a set of edges, L is a set of labels, and $l : V \cup E \rightarrow L$ is a mapping that assigns labels to vertices V and edges E. If labels are not of decisive importance, we will use the* short *definition of a graph $G = (V, E)$. In the following we also use* ***network*** *as a synonym for graph.*

Definition 11 (Subgraph) *Let $G_1 = (V_1, E_1, L_1, l_1)$ and $G_2 = (V_2, E_2, L_2, l_2)$ be labeled graphs. G_1 is a* **subgraph** *of G_2 ($G_1 \sqsubseteq G_2$) if the following conditions hold: $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $L_1 \subseteq L_2$, $\forall x \in V_1 \cup E_1 : l_1(x) = l_2(x)$. If G_1 is a subgraph of G_2, then G_2 contains G_1.*

Definition 12 (Isomorphism) *Two graphs are* **isomorphic** *if there exists a bijection f between the nodes of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ such that $(v_{1a}, v_{1b}) \in E_1$ iff $(v_{2a}, v_{2b}) \in E_2$ where $v_{2a} = f(v_{1a})$ and $v_{2b} = f(v_{1b})$. If G_1 is isomorphic to G_2, we will refer to (v_{1a}, v_{1b}) and (v_{2a}, v_{2b}) as* **corresponding edges** *in the following.*

Note that we do not consider the labeling in this definition, as we are interested in isomorphic graphs that potentially show a diverse labeling in Chapter 6.2.

The problem to decide whether two graphs are isomorphic, i.e. the so-called *graph isomorphism problem*, is not yet known to be NP-complete or in P. Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the *subgraph isomorphism problem* consists in finding a subgraph of G_2 that is isomorphic to G_1. This problem is known to be NP-complete [35].

Definition 13 (Embedding) *If graph G_1 is isomorphic to a subgraph S of graph G_2, then S is referred to as an* **embedding** *of G_1 in G_2.*

Definition 14 (Frequent Subgraph / Motif) *A graph G_1 is a* **frequent subgraph** *of graph G_2 if G_2 contains at least t embeddings of G_1, where t is a user-set frequency threshold parameter. Such a frequent subgraph is often called a* **motif**.

Definition 15 (Union Graph) *Given a sequence of n graphs $G_{seq} = \{G_1, \cdots, G_n\}$ with $G_i = (V_i, E_i)$ and $V_i = V_1$ for all $1 \leq i \leq n$. Then the* **union graph** *$UG(G_{seq})$ of G_{seq} is defined as $UG(G_{seq}) = (V_{UG}, E_{UG}, \ell)$, $E_{UG} = \cup_{1 \leq i \leq n} E_i$. The definition of the labeling function ℓ can be chosen in accordance with the application specific requirements.*

An example for the transformation of a time series of graphs into a union graph is depicted in Figure 6.1. Note that the union of all edges of the time series is the set of edges of the union graph.

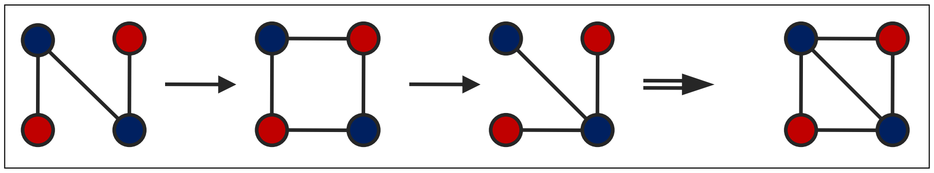


Figure 6.1: Transformation from a time series of three labeled graphs (*red* and *blue*) into the corresponding union graph.

Construction of Brain Co-Activation Networks Out of fMRI Time Series

As we are interested in topological patterns that are characteristic for patients with somatoform pain disorder, we perform graph mining on brain co-activation networks. Each detected motif represents interacting regions of the brain.
In order to receive such network models, fMRI time series data can be transformed in the following manner. We define the voxels of the fMRI image data as the vertex set. The measured value of a voxel indicates the degree of blood circulation in the particular brain region. The *darker* the voxel, the *more* blood is present in the compartment, thus the *higher* the activation is. Edges between two vertices v_1 and v_2 stand for a similar level of activation of the two corresponding voxels. We distinguish two categories of activation levels for each voxel v at each time point i, denoted by v_i. $a(i)$ stands for its activation level at time point i. We determine an $activity\text{–}score$ for each voxel v by comparing the median activation level of v across the time series with $a(i)$ in order to assign activation categories.

$$activity\text{–}score(v_i) = \frac{a(i) - median_{j\in\{1,..,n\}}a(j)}{median_{k\in\{1,..,n\}}\left|(a(k) - median_{j\in\{1,..,n\}}a(j)\right|}$$

We use a median-based $activity\text{–}score$ rather than a mean-based as we want to detect unusually high activation levels. In contrast to a mean-based $activity\text{–}score$ a median-based $activity\text{–}score$ is more robust w.r.t. these extremes and better suited for detecting them, as validated in initial experiments (not shown here). The resulting activation levels *high* and *not significant* can then be formalized as follows:

- **high activation**:
  $a(i)$ is significantly higher than the median activation level of v.
  ($activity\text{–}score(v(i)) \geq$ 7.0)

- **no significant activation**:
  $a(i)$ is not significantly higher than the median activation level of v.
  ($activation\text{–}score(v(i)) <$ 7.0).

Edges between vertices v_1 and v_2 are assigned if v_1 and v_2 both show high activation. Finally, we perform frequent subgraph mining on the resulting union graph that is formed by the given time series according Definition 15.

Performing Frequent Subgraph Mining on Brain Co-Activation Networks

In order to find frequent subgraphs in our network, we have to group our nodes and assign each group a labeling. A meaningful labeling of nodes when considering brain networks is a mapping of the nodes to their corresponding brain compartments. Hence, subgraphs in those networks represent a complex of brain compartments that show a similar activation profile. We remove edges between nodes that share the same label, as a correlated degree of activation within one region is trivial, and we are interested in similar activity of different regions.

Then we apply two large graph mining algorithms for finding subgraphs in our labeled graphs. First, we employ the heuristic approach GREW [63], as the runtime effort for exhaustive enumeration grows exponentially in the size of the subgraphs. This enables us to find frequent subgraphs in a large graph rather than to restrict ourselves to small frequent subgraphs, as described in Section 2.4. Nevertheless, as we want to avoid missing out on embeddings of subgraphs that consist of a small number of vertices (up to six), we also employ the exhaustive enumeration strategy FANMOD by [102].

Evaluation of Detected Frequent Subgraphs

To find subgraphs that are characteristic for a certain disease, we have to analyze the subgraphs according to their appearance. Therefore we want to detect subgraphs that occur in patients but not in the control group and vice versa. Another class of subgraphs that might be interesting are subgraphs that occur in all subjects that attend a certain study.

A further aspect that should be considered is the label distribution across subgraphs, as voxels that belong to larger ROIs of a particular subject s have a higher probability to appear in a motif than a label that covers a small number of nodes. Hence, we have to define the normalized frequency of a node label l, denoted by $freq_{norm}(l)$.

$$freq_{norm}(l) = \sum_{s \in S} \frac{freq_{m \in \{1,\dots,n\}}(l) \cdot \#Embeddings(m|s)}{freq_{Background}(l|s)},$$

where S denotes the set of all subjects and $freq_{m \in \{1,\dots,n\}}$ stands for the number of occurrences of label l in a motif m. This number has to be multiplied by the number of isomorphic subgraphs of m found in a subject s, its embeddings found in s. The $freq_{Background}(l|s)$ describes the number of occurrences of label l w.r.t. all vertices of the network that refers to subject s. In our case this is equivalent to the *size* of a ROI. Finally, we sum up over all subjects given in the dataset.

6.1.2 Experimental Evaluation

The experimental section is organized as follows. First, we summarize the construction of the network models, including a detailed description of the used dataset and the labeling scheme for the vertices of the networks. Then we perform motif discovery using the heuristic algorithm GREW [63] to allow for finding subgraphs of arbitrary size. As an evaluation of these results indicates that a multitude of the subgraphs found by GREW consist of only a small number of vertices, we go a step further and additionally apply the exhaustive sampling technique FANMOD by [102] on the network models. Finally, we compare the results of both algorithms and give representative subgraphs for the group of subjects that suffer from the somatoform pain disorder and typical subgraphs for the group of healthy controls.

Construction of the Brain Network Models

We created networks for ten subjects that attended the studies of [41]. The resulting networks comprise 66 to 440 nodes with 90 different classes of node labels and 358 to 13,548 edges. In addition, the network models indicate different number of edge types. These refer to the concatenation of the labels of the adjacent nodes. All edges are undirected because in our specific application, where both adjacent voxels have to show high activation to be connected by an edge, a direction makes no sense. The exact statistics of each subject are depicted in Table 6.1.

Table 6.1: Statistics of the brain network models for each subject.

SUBJECT	# VERTICES	# LABELS	# EDGES	# EDGE TYPES	# SUBGRAPHS BY GREW
PAT 1	102	31	453	91	38
PAT 2	185	32	1,961	84	706
PAT 3	241	35	3,506	90	505
PAT 4	263	46	5,152	293	752
PAT 5	313	46	10,977	372	4,154
PAT 6	440	58	13,548	475	4,256
CONT 1	66	10	358	15	15
CONT 2	99	33	407	111	32
CONT 3	109	18	1,093	13	133
CONT 4	202	35	2,045	133	236

Time Series Datasets. We used fMRI time series data (1.5 T MR scanner) of 6 female somatoform patients and four healthy controls. Standard data preprocessing including realignment, correction for motion artifacts and normalization to standard space have been performed using SPM2[1]. In addition, to remove global effects the voxel time series have been corrected regressing out the global mean, as suggested in [85].

Vertex Labels. We labeled each node in the network by regional parcellation of the voxels into 90 brain regions using the template of Tzourio-Mazoyer *et al.* [96].

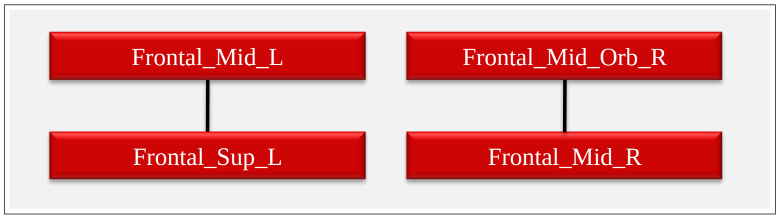
Finding Frequent Subgraphs with GREW

To allow for finding subgraphs of arbitrary size, we searched for topological subgraphs using GREW with a frequency threshold of $t = 5$. In these experiments, we searched for subgraphs with a minimum number of one edge. The total number of different subgraphs found in the ten networks is depicted in the last column of Table 6.1. Altogether we found 10,530 different subgraphs in somatoform patients and healthy controls. 10,173 different subgraphs were detected among patients, 413 within the group of healthy subjects, where some of these subgraphs were also found among the patients and vice versa.

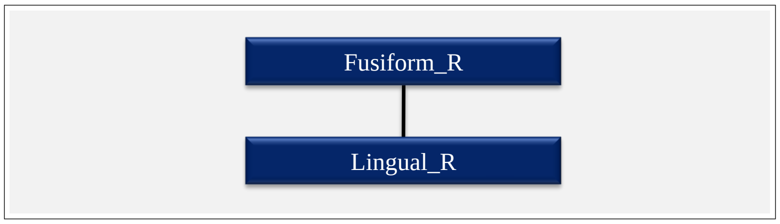
For validation we divided the subjects into three classes. Class (1) contains only the somatoform patients, class (2) consists of the controls exclusively and class (3) composes the union of class (1) and (2). Figure 6.2 shows typical representatives of each class. The two frequent subgraphs on the left occur in 57% of the patients but in no healthy subject. The middle motif arises in 50% of the healthy controls but in no patient. The upper motif on the right-hand side was found in 50% of the control group and in 14% of the patients, the lower motif in 25% of the control group and in 43% of the patient group. Most of the typical representatives of both groups comprise only a small number of vertices.

The largest subgraph (highest number of vertices and edges) of class (1) was found in subject 'Pat 5'. It consists of 28 vertices and 29 edges, five different brain compartments are involved in this motif. This motif has in total 34 instances in

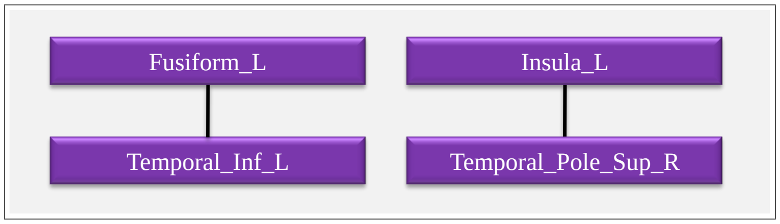
[1]`http://www.fil.ion.ucl.ac.uk/spm/`



(a) Patients *only*.



(b) Healthy controls *only*.

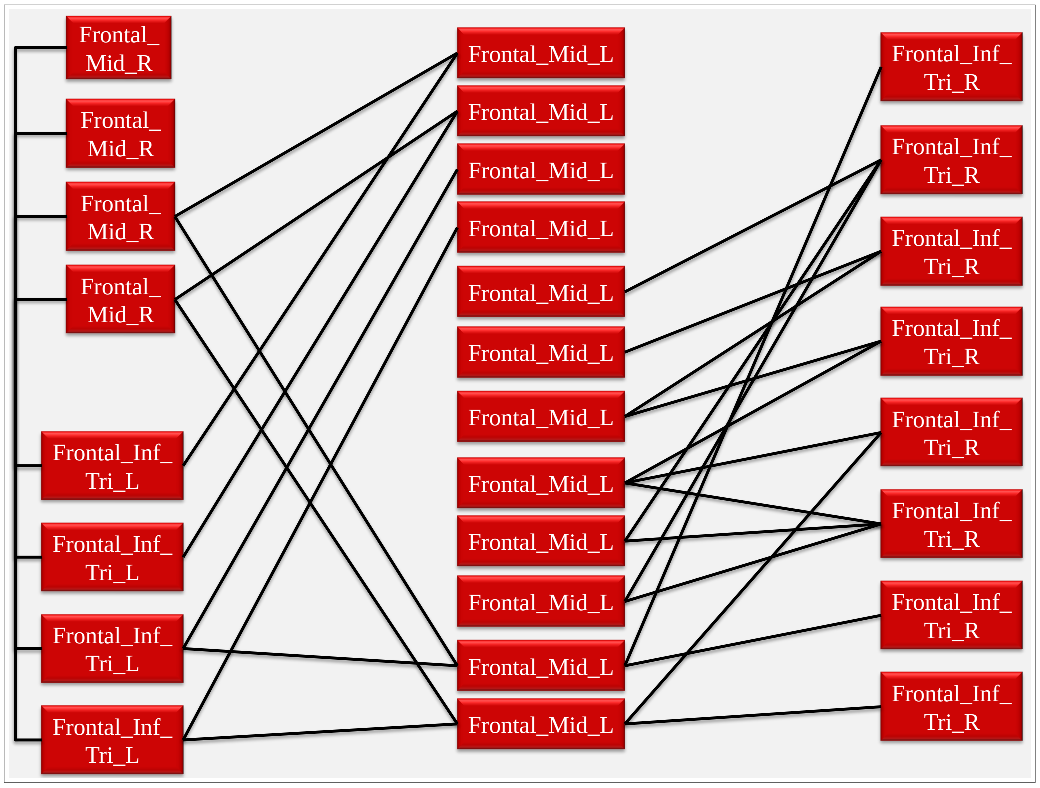


(c) Patients and Healthy controls.

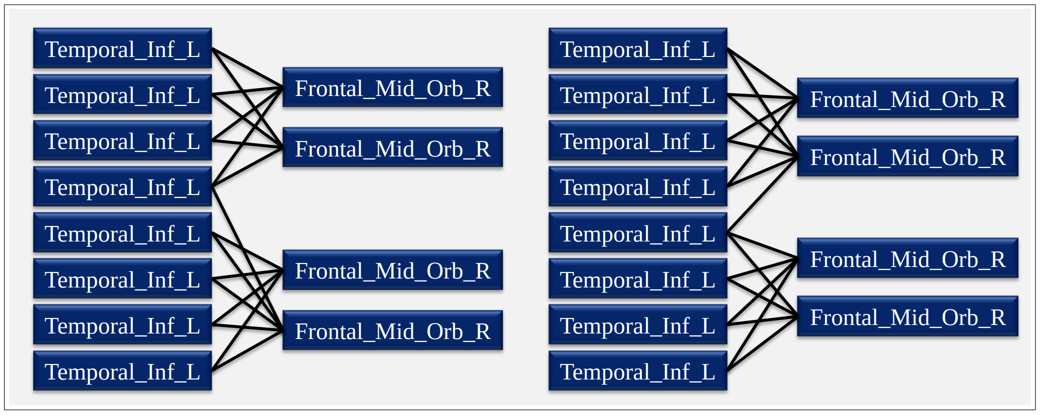
Figure 6.2: Typical representatives of subgraphs found in the groups of somatoform patients, healthy controls and the group of all subjects, respectively.

this subject. The largest subgraphs in class (2) were detected in subject 'Cont 3'. We found two subgraphs that comprise twelve nodes with two different labels and 17 edges. An example of the largest subgraphs found among the patients and the two largest subgraphs of the controls are shown in Figure 6.3. Note that no motif occurs in all subjects.

Evaluation of ROIs. We determined the frequencies of the ROIs in patients and controls. Figure 6.4(a) illustrates the results for the 15 most frequent ROIs w.r.t. the patient group, and Figure 6.4(b) for the group of healthy controls, respectively. Within the subgraphs found in the patient group, *Caudate_R* is the



(a) Patients.



(b) Healthy controls.

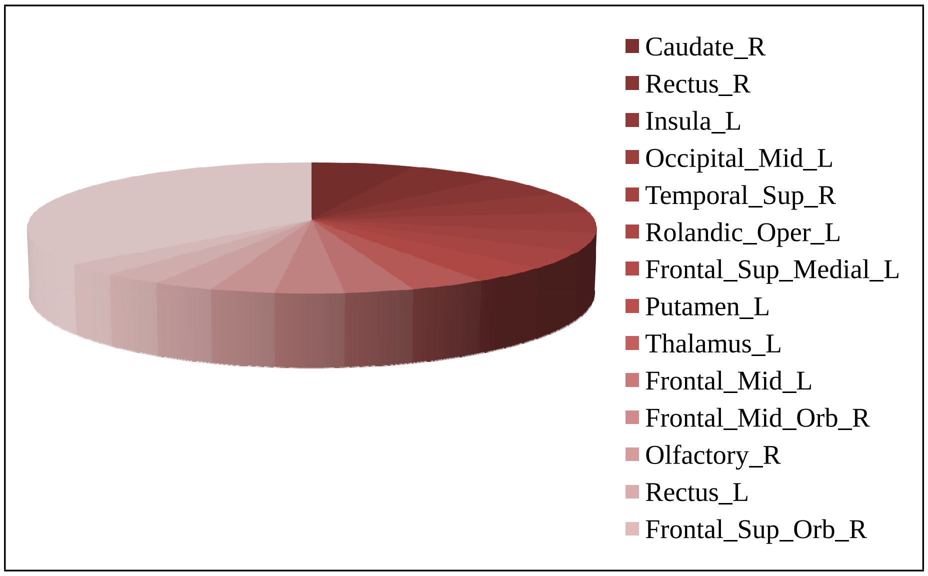
Figure 6.3: Largest subgraphs found in the groups of somatoform patients and healthy controls.

most frequent ROI ($freq_{norm} = 6.43\%$). The most frequent ROI w.r.t. the subgraphs of the control group is *Frontal_Mid_Orb_R* ($freq_{norm} = 22.04\%$). ROIs that show a small frequency are summarized by the label *Miscellaneous*. Our results are consistent with a previous study [41]. They report different activation pattern in the regions *Insula_L* (Patients: $freq_{norm} = 5.29\%$ Controls: $freq_{norm} = 1.95\%$) and *Frontal_Mid_Orb_R* (Patients: $freq_{norm} = 3.38\%$ Controls: $freq_{norm} = 22.04\%$), the key-player in the group of healthy controls. Our results of different activation of the parahippocampal cortex in patients and controls (not depicted in Figure 6.4) supports a recent study that suggests that patients with posttraumatic stress disorder showed also an altered activation pattern in the parahippocampal cortex in comparison to healthy controls when subjected to painful heat stimuli [37]. In addition, we found that patients show increased activation in *Rolandic_Oper_L*, *Caudate_R* and *Rectus_R* whereas the control group is activated in the regions *Temporal_Inf_L*, *Heschl_R* and *Lingual_R* to a higher degree. Also, the olfactory region shows alterations in the activation of patients and controls. Whereas *Olfactory_R* occurs to a much higher degree in subgraphs found in patients, subgraphs found in the networks of controls are labeled more often with *Olfactory_L*.

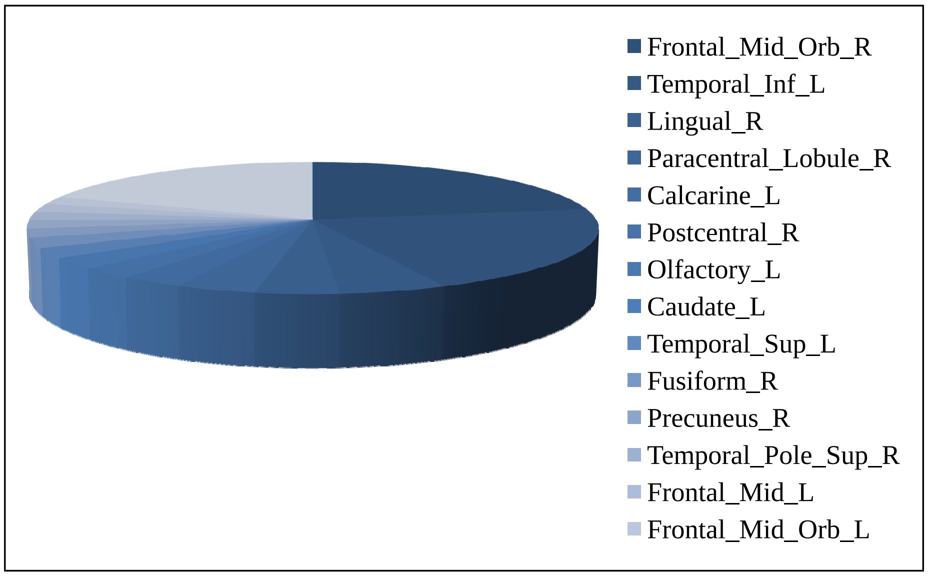
Finding Frequent Subgraphs with FANMOD

The public disposable FANMOD tool[2], implementing the exhaustive sampling algorithm by [102], is restricted to networks with a maximum number of 16 different vertex labels. Hence, we map the original network data to a modified version where only the 15 most frequent ROIs are kept. The remaining vertices are labeled by *Miscellaneous*. This labeling scheme refers to the ROI distribution among subgraphs detected by GREW as shown in Figure 6.4. As edges of the type *Miscellaneous–Miscellaneous* occur disproportional frequent within the network, and would therefore tamper with the real frequencies of the results, we ignored that kind of interaction.

[2] `http://theinf1.informatik.uni-jena.de/motifs/`



(a) Patients.



(b) Healthy controls.

Figure 6.4: Frequencies of ROIs in subgraphs of patients and controls, respectively.

Table 6.2: Subgraphs detected by FANMOD for different number of vertices compared to the number of subgraphs found by GREW, respectively. For some motif sizes, FANMOD is not applicable as the runtime effort for exhaustive enumeration grows exponentially in the size of the subgraphs.

SUBJECT	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$3 \leq k \leq 6$	# SUBGRAPHS BY GREW
PAT 1	9	13	28	42	92	4
PAT 2	42	121	288	-	451	406
PAT 3	5	6	7	8	26	4
PAT 4	22	91	374	-	487	56
PAT 5	139	509	-	-	648	988
PAT 6	431	3,292	-	-	3,723	1,861
CONT 1	6	10	16	22	54	34
CONT 2	18	32	51	78	179	56
CONT 3	9	26	-	-	35	161
CONT 4	137	565	2,399	-	3,101	472

We searched for subgraphs, consisting of up to six vertices and used a sample rate of 100,000 to estimate the number of subgraphs which is the default parameterization, as recommended by the authors. Table 6.2 illustrates the number of subgraphs that comprise $3 \leq k \leq 6$ vertices for each subject. We compared these results by applying GREW ($t = 5$) to the same modified networks. The last column of Table 6.2 demonstrates that for small subgraph sizes an exhaustive method provides significant more information about the graph structure. Note that the total number of detected subgraphs by FANMOD with a limited number of vertices exceeds the total number of subgraphs found by GREW with arbitrary size in most cases. However, an exhaustive method is only applicable for finding subgraphs that consist of a small number of vertices.

6.1.3 Conclusions

In this work, we have applied GREW, a heuristic approach for finding frequent subgraphs in one large network and the algorithm FANMOD, an exhaustive sampling method for frequent subgraph discovery. We executed both algorithms on union graph models resulting from time series data of six somatoform patients and four healthy controls. The detected subgraphs represent groups of brain compartments that covary in their activity during the process of pain stimulation. We evaluated the appearance of subgraphs for both groups. Our results let us suspect that somatoform brain disorder is caused by an additional pathogeneous activity, not by a missing physiological activity. So far we care about the topology of the network, but ignore the temporal order of these interactions. When studying network topology, it is important to bear in mind that the network models currently available are simplified models of the systems that govern cellular processes. While these processes are dynamic, the models we consider so far are all static.

In the next Section, we extend this idea to perform real *dynamic* graph mining on biological yeast network data.

6.2 Motif Discovery in Dynamic Yeast Network Data

The majority of recent subgraph mining approaches have focused on characterizing the topology of static networks. But to model real-world systems, often a temporal component has to be taken into account, as interactions between objects here usually occur for a certain period of time only. Therefore, a realistic model has to consider that edges will be inserted and/or deleted over time. The resulting data structure is called a *dynamic graph*.

These dynamic graphs occur in many real-world applications. In Biology, a wide-spread approach is to model interacting proteins as networks, where each vertex corresponds to a protein and two vertices are connected by an edge if the corresponding proteins can bind. In addition, further technologies allow biologists to measure the distribution of protein interactions at different time points. Hence, associating a time series for each protein provides interesting insights into the dynamically changing system. In social networks like Facebook, people contact each other at specific time points and form various complex relationships.

Our new approach for frequent subgraph discovery, provided in this section, performs data mining on such dynamic graphs in an on-top fashion, i.e. we search a set of frequent *static* subgraphs for frequent *dynamic* patterns. Existing subgraph mining algorithms on static graphs can be easily integrated into our framework to make them handle dynamic graphs. The search for dynamic patterns is based on the idea of suffix trees.

6.2.1 Performing Graph Mining on Dynamic Network Data

Now we provide the essential definitions from dynamic graph theory necessary to follow our argumentation, based on the definitions of classic graph theory, presented in Section 6.1.1.

Basics of Dynamic Graph Theory

First, we give the formal definition of a time series of graphs.

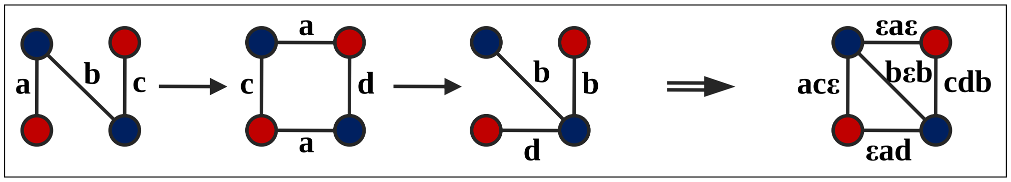


Figure 6.5: Transformation of a time series of graphs into a dynamic graph. The three graphs on the left represent a time series of graphs with edge insertions and edge deletions over time. The graph on the right is a dynamic graph that summarizes all information represented by the time series.

Definition 16 (Time Series of Graphs) *Given a sequence G_{ts} of n graphs $\{G_1, \ldots, G_n\}$ with $G_i = (V_i, E_i, L_i, l_i)$ for $1 \leq i \leq n$. We define G_{ts} to be a **time series of graphs** if $V_1 = V_i$ for all $1 \leq i \leq n$. G_i is the i-th state of G_{ts} and A_i is the adjacency matrix of the i-th state. In every state i, $l_i : V_i \cup E_i \rightarrow L_i$ assigns labels to nodes and edges.*

Note that l_i is not necessarily the same for all $1 \leq i \leq n$. This means that an edge (or even a node) might change its label in consecutive time steps of the time series. Such a time series of graphs can be transformed into a dynamic graph as follows:

Definition 17 (Dynamic Graph) *Given a time series of graphs G_{ts} of n graphs. Then the **dynamic graph** $DG(G_{ts})$ of G_{ts} is defined as $DG(G_{ts}) = (V_{DG}, E_{DG}, \ell)$, where $V_{DG} = V_i$ for all $1 \leq i \leq n$ and $E_{DG} = \cup_{i=1}^{n} E_i$. The mapping $\ell : E_{DG} \rightarrow L \cup \varepsilon$ maps each edge e in E_{DG} to a string $\ell(e)$ of length n, referred to as the* **edge existence string** *of e. Let us denote the i-th character of $\ell(e)$ as $\ell(e(i))$. $\ell(e(i)) = \varepsilon$ if e does not exist in state i of G_{ts}. If e does exist in state i of G_{ts}, then $\ell(e(i)) = l_i(e)$.*

Hence, a dynamic graph is a union graph (cf. Definition 15) over a time series of graphs in combination with the corresponding existence string. An example for such a transformation is depicted in Figure 6.5.

To extend frequent subgraph discovery to dynamic networks, we have to define two types of frequent subgraphs:

Definition 18 (Static Frequent Subgraph) *A subgraph $G_{S_{ts}}$ that has more than t embeddings in a dynamic graph $DG(G_{ts})$ is called a* ***static frequent subgraph****.*

In other terms, static frequent subgraphs in a dynamic graph are subgraphs that are topologically frequent in the summary graph. Hence, edge existence strings play no role when looking for static frequent subgraphs. The static frequent subgraphs in a dynamic graph are defined in exactly the same manner as frequent subgraphs on one single network. A dynamic pattern D is a static frequent subgraph S such that in more than u embeddings of S, edge insertions, deletions and label changes occur in the same temporal order. As we want to study cases where these insertions and deletions happen in a subsection of the complete time series only, we now have to make use of the existence strings of the edges.

Definition 19 (Common Substring) *For two strings s_1 and s_2 of length $|s_1|$ and $|s_2|$, $sub(s_1, i, j) = sub(s_2, i, j)$ means that the substrings from the i-th to the j-th character in s_1 and s_2 are identical, where $1 \leq i \leq j \leq \min(|s_1|, |s_2|)$. This identical substring is called a* ***common substring*** *of s_1 and s_2.*

Definition 20 (Common Dynamic Pattern) *Let S be a static frequent subgraph in dynamic graph DG, and let $S_1 = (V_{S_1}, E_{S_1})$ and $S_2 = (V_{S_2}, E_{S_2})$ be two embeddings of S in DG. As S_1 and S_2 are isomorphic, there must be a bijection f between E_{S_1} and E_{S_2}. Then S_1 and S_2 share a* ***common dynamic pattern*** *from position i to j if for all pairs of edges (e_1, e_2) from $E_{S_1} \times E_{S_2}$ where $e_2 = f(e_1)$ the following equality holds: $sub(\ell(e_1), i, j) = sub(\ell(e_2), i, j)$, i.e. the existence strings of corresponding edges are identical from position i to position j.*

If enough embeddings of the same static frequent subgraph share the same common dynamic pattern, then this static frequent subgraph contains a dynamic frequent subgraph. Note, that common dynamic patterns consider edge labels, whereas static frequent subgraphs ignore the labeling.

Definition 21 (Dynamic Frequent Subgraph) *Let S be a static frequent subgraph with u embeddings $\{S_1, \ldots, S_u\}$ in a dynamic graph DG. Let t be a user-defined frequency threshold. If at least t embeddings of S share the same common dynamic pattern, then the topology of S and the common dynamic pattern represent a* ***dynamic frequent subgraph****.*

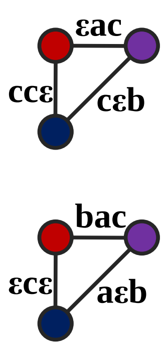
For instance, in our real-world case study presented in this section, a dynamic frequent subgraph describes a set of groups of proteins that show similar patterns of co-expression during a certain interval of a time series.

Dynamic Graph Mining

After defining frequent subgraphs in dynamic networks, we now present an efficient algorithmic solution on how to compute them. Our framework is based on the idea of suffix trees. Dynamic frequent subgraph discovery can be performed in an on-top fashion in two steps. First, we employ one of the state-of-the-art algorithms for finding frequent subgraphs in the union graph of a time series of graphs (cf. Section 2.4). Second, we search the resulting static frequent subgraphs for frequent dynamic patterns. Figure 6.6 visualizes an example for frequent subgraph discovery in a dynamic network.

Matrix Representation of Embeddings. To discover dynamic patterns we process the results of a static frequent subgraph mining algorithm (see Figure 6.6(a)). Key to our efficient scheme is to represent each embedding of a frequent static subgraph by the set of existence strings of its edges. Hence, for each of the s embeddings of a static frequent subgraph S with m edges, we obtain a set of m existence strings. We represent these sets of strings for one embedding S_i of S as an $m \times n$ matrix, $M(S_i)$, where each row corresponds to one edge, and each column to one time point in the time series (see Figure 6.6(b)).

Canonical Edge Order. To compare the matrices $M(S_1)$ and $M(S_2)$ of two different embeddings S_1 and S_2 of the same subgraph S efficiently, we have to



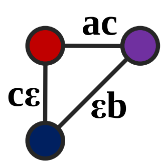
(a) Two Embeddings.

time point	1	2	3
edge 🔴—🟣	ε	a	c
edge 🟣—🔵	c	ε	b
edge 🔵—🔴	c	c	ε
concatenation	**<1:εcc>**	**<2:aεc>**	**<3:cbε>**
edge 🔴—🟣	b	a	c
edge 🟣—🔵	a	ε	b
edge 🔵—🔴	ε	c	ε
concatenation	**<1:baε>**	**<2:aεc>**	**<3:cbε>**

(b) Matrix representation.

time point	1	2	3
edge 🔴—🟣	ε	a	c
edge 🟣—🔵	c	ε	b
edge 🔵—🔴	c	c	ε
concatenation	<1:εcc>	**<2:aεc>**	**<3:cbε>**
edge 🔴—🟣	b	a	c
edge 🟣—🔵	a	ε	b
edge 🔵—🔴	ε	c	ε
concatenation	<1:baε>	**<2:aεc>**	**<3:cbε>**

(c) Common substring.



(d) Dynamic pattern.

Figure 6.6: Two embeddings (6.6(a)) of the same static frequent subgraph and their common dynamic pattern depicted as dynamic graph (6.6(d)). The two embeddings are shown with their matrix and string representation each (6.6(b)). In time step 2 and 3, edge insertions and deletions occur simultaneously in both embeddings. This is the common substring (6.6(c)) that refers to the dynamic pattern.

define a canonical ordering of edges for all embeddings of S. gSpan [107] orders the edges in each embedding of a frequent subgraph according to a Depth First Search on the vertices (cf. Section 2.4.1). In this manner, we can sort the rows of all matrices representing embeddings of the same subgraph such that corresponding edges are represented by the same row in each of these matrices.

String Representation of Embeddings. Using this canonical form of the matrices dynamic frequent subgraph discovery is equivalent to detecting identical blocks of columns. Therefore we transform each matrix into a string representation by treating each column in the matrix as one symbol which is just the concatenation of all entries in this column. We obtain s strings, each describing one embedding, which have a common substring if the embeddings show identical dynamic behavior over a certain period of time (see Figure 6.6(c)).

Frequent Common Substring Discovery. A classic result from the field of string algorithms [50] helps us to discover these common substrings extremely efficiently: detecting all longest substrings in a set of s strings is possible in linear runtime, by reading all strings exactly once by employing a suffix tree to store all substrings from a set of strings. A simple breadth-first search in the tree then provides all frequent common substrings. Obviously, each of these frequent common substrings corresponds to one dynamic pattern (see Figure 6.6(d)). To guarantee that all embeddings of a dynamic pattern start at the same time point of the time series, we include a time stamp into our matrix representation of each embedding, by adding an extra row to the matrix, which numbers the time steps from 1 to n, where n is the number of time steps in the time series. We then include these time stamps into the string representation of our matrix.

6.2.2 Dynamic Network Construction

In this section, we confirm the practical feasibility of our framework for finding frequent subgraphs in dynamic networks on the basis of a large case study on real-world biological data. We describe in detail how to construct the dynamic network models by a combination of interaction data and a time series dataset.

Protein-Protein-Interaction Datasets. We created a dynamic network by integrating protein-protein-interaction (PPI) data from yeast and a time series of yeast gene expression levels. We obtained the yeast PPI network data from the database DIP (Database of Interacting Proteins) [106, released on January 6, 2008], containing all pairs of interacting proteins identified in the yeast organism *S. cerevisiae*. This dataset consists of 4,923 proteins, which are the original set of nodes in our dynamic graph and 18,324 interactions representing edges between the nodes.

Vertex Labels. In order to assign labels to all nodes of the network, we mapped the proteins to their corresponding protein-coding open reading frames (ORFs), via information provided by the SGD (Saccharomyces Genome Database) [46] from DIP. Finally, we determined the 'molecular function' associated with this ORF in the Gene Ontology (GO) hierarchy at depth two [5]. As GO terms are organized in directed acyclic graphs, and each term can be traced to different depths, it can occur that multiple GO annotations can be assigned to one single ORF at a given depth. As current methods for frequent subgraph mining and enumeration require a unique label, we retained all ORFs with a non-ambiguous GO labeling at depth two. The twelve remaining GO functional classes were then used as node labels for the remaining proteins.

Time Series Datasets. Next we used a cell cycle time series of yeast gene expression levels by [22], available from the ExpressDB collection of yeast RNA Expression Datasets [2]. The dataset consists of 6,601 time series of 17 measurements for 6,259 ORFs, of which 6,045 ORFs can be mapped to one unique time series. 3,607 ORFS are both present in our GO-labeled yeast PPI network and in the gene expression dataset. The corresponding proteins are the final set of nodes in our dynamic graph, other nodes and their adjacent edges from the original graph were skipped. The final network comprises 3,607 nodes, twelve different classes of node labels, 7,395 edges per time step, and 17 time steps.

Edge Existence Strings. Each edge in our dynamic network was assigned an edge existence string (cf. Definition 17) of length 17 according to the following rules: We distinguish three basic categories of gene expression levels for each gene g at each time point i. We refer to its expression level at time point i as $g(i)$. Categories are assigned by comparing the median expression level of g across the time series with the $z\text{–}score_{median}$ of $g(i)$:

$$z\text{–}score_{median}(g(i)) = \frac{g(i) - median_{j \in \{1,..,n\}} g(j)}{\sqrt{\frac{1}{n} \sum (g(k) - median_{j \in \{1,..,n\}} g(j))^2}}$$

This results in three different expression levels:

- **H**igh:
  $g(i)$ is significantly higher than the median expression level of g, i.e. $z\text{–}score_{median}(g(i)) \geq 2$. That indicates that g supports important steps of the examined biological process, or regulates small subunits.

- **M**edium:
  $g(i)$ does not significantly differ from the median expression level of g, i.e. $-2 < z\text{–}score_{median}(g(i)) < 2$. That means that g does not play an important role in the examined biological process.

- **L**ow:
  $g(i)$ is significantly lower than the median expression level of g, i.e. $z\text{–}score_{median}(g(i)) \leq -2$. That argues for a repression functionality of g in the examined biological process.

For each edge e in our dynamic graph, we compared the expression profiles of its adjacent genes g_1 and g_2 to generate the edge existence string according to the following rules:

- $\ell(e(i)) =$ 'H' if $g_1(i)$ and $g_2(i)$ both show a high gene expression level.
- $\ell(e(i)) =$ 'M' if $g_1(i)$ and $g_2(i)$ both show a medium gene expression level.
- $\ell(e(i)) =$ 'L' if $g_1(i)$ and $g_2(i)$ both show a low gene expression level.

For pairs of genes g_1 and g_2 that show different gene expresson levels, we additionally introduce the character 'N'. In this manner, we generated edge existence strings from the alphabet $\Sigma = \{H, L, M, N\}$ for pairs of interacting genes in the yeast PPI network.

Enumerating all Static Frequent Subgraphs

In order to compute exact frequencies for static frequent sugraphs, we exhaustively enumerated all frequent subgraphs of our dynamic network. This step was performed using the FANMOD tool[3]. As our dynamic network is a large graph, we have to limit ourselves to subgraphs covering three or four vertices as previous studies [105], but this allows us to guarantee that we are not missing out on dynamic frequent subgraphs.

[3]`http://theinf1.informatik.uni-jena.de/motifs/`

6.2.3 Evaluation of Dynamic Frequent Subgraphs

Now we describe how to evaluate the detected motifs according significance and the evolutionary conservation rate among different species.

Significance of Dynamic Frequent Subgraphs

To assess the significance of a static frequent subgraph with frequency t, we use p-values for each static frequent subgraph according to [102]. The p-value represents the probability of this static frequent subgraph occurring at least t times in a random graph with identical degree distribution. We only retain those static frequent subgraphs whose p-value is below the significance level of $\alpha = 0.025$. We also assess the significance of the common dynamic string pattern σ of a dynamic frequent subgraph D which occurs in u out of t embeddings of the static frequent subgraph S in terms of a p-value. For this purpose, we compute the probability p of σ to occur in an embedding of S by chance. Let $M = M(i,j)_{l' \times n'}$ be the matrix representation of σ, where each row corresponds to one edge (existence string) and each column to one time step. We then define the probability of σ occurring by chance as

$$p_\sigma = \prod_{i=1}^{l'} \prod_{j=1}^{n'} p_i(M(i,j)),$$

where p_i is the background probability of the character represented by $M(i,j)$ occurring in the i-th edge of S. In other terms, the random model assumes that the existence strings of the embeddings of S were randomly generated. Under this model, the probability of σ appearing u times (denoted $|\sigma| = u$) in the t embeddings of S then follows a binomial distribution:

$$P(|\sigma| = u) = \binom{t}{u} p_\sigma^u (1 - p_\sigma)^{t-u},$$

and the p-value of the common dynamic pattern σ can be computed straightforwardly:

$$p\text{-value}(|\sigma| = u) = P(|\sigma| \geq u)$$

We deem common dynamic patterns significant, if their p-value is below 0.025. Dynamic frequent subgraphs are considered significant if their associated static frequent subgraph is significant and if its common dynamic pattern is significant as well. We only retained those dynamic frequent subgraphs that occur in at least 50% of the embeddings of the corresponding static frequent subgraph.

Evolutionary Conservation Rate: A Quality Criterion for Dynamic Frequent Subgraphs

The evolutionary conservation rate is known to be a good quality criterion in biology. Following previous biological analysis [89], we deem dynamic frequent subgraphs the more conserved, the more proteins that participate in embeddings of this subgraph have identifiable orthologs in other organisms. We analyse the conservation rate of yeast proteins by searching for orthologs (i.e. homologous sequences that were separated by a speciation event) in each of five other higher eukaryotic organisms for which information is available in the InParanoid database [84]: 644 of the 3,607 proteins in our dynamic network turned out to be conserved across all five species. Hence the probability of observing a conserved protein by uniform random sampling from the PPI network is 0.1785, of observing a set of three conserved proteins is 0.0057, and of observing a set of four conserved proteins is 0.0010.

Conservation Rate among Static and Dynamic Frequent Subgraphs

Our static frequent subgraph discovery results in 367 subgraphs with three nodes and within these 515 dynamic patterns that were both frequent and significant. We found 2,302 proteins to be members of at least one static frequent subgraph, out of which 577 are conserved across species (conservation rate 0.2507). Hence the rate of conservation among proteins that are part of static frequent subgraphs is significantly higher than among proteins that are sampled randomly from our dynamic network (Binomial distribution $B(t = 2,302; p = 0.1785)$; p-value < 0.0001). 1,787 proteins are members of at least one dynamic frequent subgraph, including 477 conserved proteins (conservation rate 0.2669). Hence the rate of conservation among proteins that are part of dynamic patterns is significantly higher than among proteins that are sampled randomly from our dynamic network (Binomial distribution $B(t = 1,787; p = 0.1785)$; p-value < 0.0001). 515 proteins are members of at least one static frequent subgraph, but of none of the dynamic patterns (100 of these are conserved, conservation rate 0.1942). While the rates of conservation among proteins from static and dynamic frequent subgraphs are not significantly different (two-sample t-test for unequal variances, $p = 0.2392$), we observed that proteins that participate in static *and* dynamic frequent subgraphs are significantly stronger conserved than those that are members of static, *but not* of dynamic static subgraphs (two-sample t-test for unequal variances, $p = 0.004$).

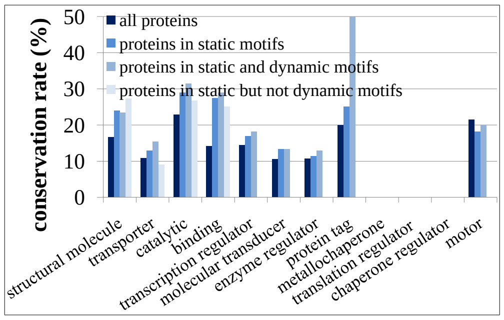
Table 6.3: Evaluation w.r.t. conservation rate of static and dynamic frequent subgraphs with three and four nodes.

	$k = 3$			$k = 4$		
	STAT.	DYN.	STAT. ONLY	STAT.	DYN.	STAT. ONLY
# FREQUENT SUBGRAPHS	367	515		3,293	2,111	
# PROTEINS	2,302	1,787	515	2,403	1,770	633
# CONSERVED PROTEINS	577	477	100	591	470	121
CONSERVATION RATE	0.2507	0.2669	0.1942	0.2459	0.2655	0.1912

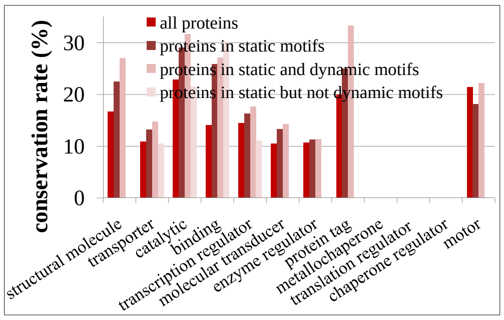
We discovered 3,293 static frequent subgraphs of size $k = 4$. Within these subgraphs we found 2,111 dynamic patterns that were both frequent and significant. 2,403 proteins are members of at least one static frequent subgraph, out of which 591 are conserved (conservation rate 0.2459). Consistent with our results on frequent subgraphs with three nodes the rate of conservation among proteins that are part of static frequent subgraphs is significantly higher than among proteins that are sampled randomly from our dynamic network (Binomial distribution $B(t = 2,403; p = 0.1785)$; p-value < 0.0001). 1,770 proteins are members of at least one dynamic pattern, including 470 conserved proteins (conservation rate 0.2655). Analogously to the frequent subgraphs with three nodes the rate of conservation among proteins that are part of dynamic patterns is significantly higher than among proteins that are sampled randomly from our dynamic network (Binomial distribution $B(t = 1,770; p = 0.1785)$; p-value < 0.0001). 633 proteins are members of at least one static frequent subgraph, but of none of the dynamic patterns (121 of these are conserved, conservation rate 0.1912). While the rates of conservation among proteins from static and dynamic frequent subgraphs with four nodes are not significantly different (two-sample t-test for unequal variances, $p = 0.1525$), we observed that proteins that participate in static *and* dynamic frequent subgraphs are significantly stronger conserved than those that are members of static, *but not* of dynamic ones (two-sample t-test for unequal variances, $p = 0.001$). Therefore the significance of the conservation rate of frequent subgraphs with four nodes is even higher than for subgraphs with three nodes. Table 6.3 summarizes the results of these experiments.

Conservation of Dynamic Frequent Subgraphs across GO Classes

We also checked whether proteins that are part of static and dynamic subgraphs show different levels of conservation depending on their molecular function as defined by GO. As can be seen from Figure 6.7, conservation rates vary widely among different functional classes, both for background conservation rate, and among dynamic and static frequent subgraphs. The figure shows the conservation rates for proteins that are participating in frequent subgraphs with three and four nodes w.r.t. twelve different classes of protein functions. For each function the general conservation rate of proteins among this class is depicted. The



(a) $k = 3$.



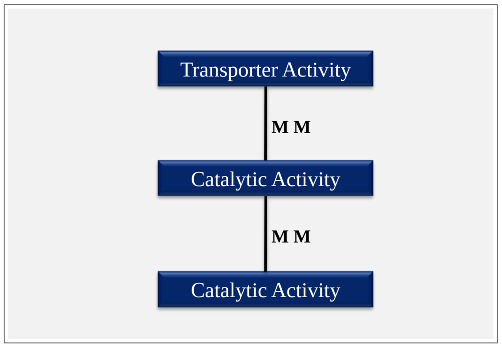
(b) $k = 4$.

Figure 6.7: Evaluation w.r.t. protein function of frequent subgraphs with three and four nodes.

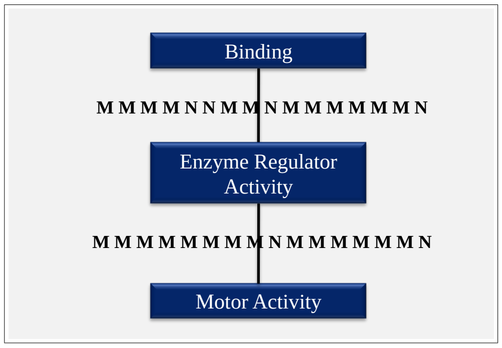
second, third and fourth bars of the histogram stand for the conservation rate of proteins involved in static frequent subgraphs, in dynamic patterns (therefore also in static frequent subgraphs) and the conservation rate of proteins that play a role in static frequent subgraphs only. It can be seen that frequent subgraphs with three and four nodes both show differences in their conservation rate depending on the protein functions. For example, proteins with structural molecule activity or proteins regulating transcription participating only in static frequent subgraphs are highly conserved in subgraphs with three nodes exclusively. Proteins that have metallochaperone activity or regulate translation or chaperones are not conserved, neither in static nor in dynamic patterns.

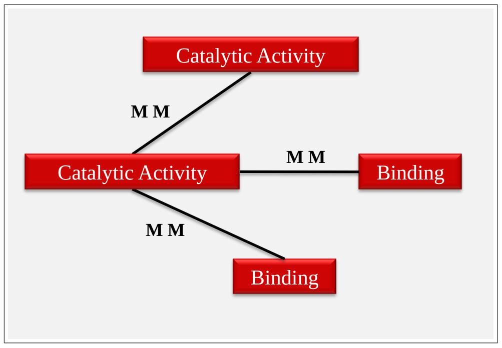
Selected Frequent Subgraphs

The most frequent and significant ($p < 0.0001$) subgraph with three nodes (cf. Figure 6.8(a)) has 4,060 embeddings. Two proteins share the same biological function *catalytic activity*, whereas the third is a transporter protein. Transporter proteins catalyze the transfer of a substance from one side of a membrane to the other. Within these 4,060 embeddings we detect 1,728 dynamic patterns comprising two time points. Figure 6.8(b) shows a significant ($p < 0.0001$) subgraph



(a) Highest number of embeddings ($k = 3$).


Binding
M M M M N N M M N M M M M M M N
Enzyme Regulator Activity
M M M M M M M M N M M M M M M N
Motor Activity
(b) Longest dynamic pattern ($k = 3$).



(c) Highest number of embeddings ($k = 4$).

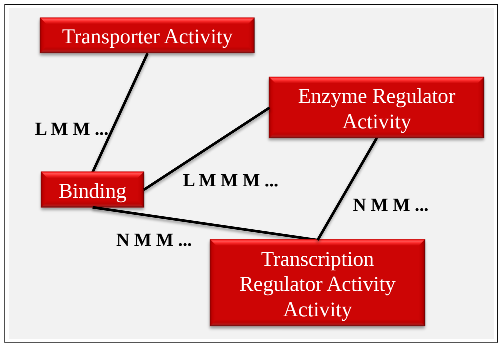

Transporter Activity
Enzyme Regulator Activity
L M M ...
Binding
L M M M ...
N M M ...
N M M ...
Transcription Regulator Activity Activity
(d) Longest dynamic pattern ($k = 4$).

Figure 6.8: Selected frequent subgraphs that consist of three or four nodes.

with three embeddings, but in two of them we detect a dynamic pattern that has identical behavior of co-expression over a period of 16 out of 17 time points. The corresponding interacting proteins are organized under *binding*, *enzyme regulator activity* and *motor activity* term of GO, respectively. This static frequent subgraph has quite few embeddings, but includes the longest dynamic pattern within static frequent subgraphs of size $k = 3$.

The most frequent subgraph of size $k = 4$ has 18,056 embeddings, illustrated in Figure 6.8(c). Two proteins show catalysis properties whereas the other two are binding proteins. 8,511 out of these embeddings have dynamic patterns comprising two time points. In Figure 6.8(d) the most significant ($p < 0.0001$) subgraph has three embeddings, whereof two have a dynamic pattern over a time period of

15 time steps. In the first time step both the binding protein and the transportation protein show very low expression levels.

6.2.4 Conclusions

We have presented a framework for frequent dynamic subgraph discovery in dynamic graphs. We have shown how to efficiently compute these patterns using suffix trees. Furthermore, we have described how to combine frequent subgraph mining algorithms with our dynamic framework and we have applied our framework on a large real-world case study to handle dynamic graphs. Dynamic graph mining is a technique that can be used in many fields of applications. Finding dynamic patterns in PPI maps promises to reveal interesting insights into biological processes.

Chapter 7

Data Mining Using Graphics Processors

In recent years, *Graphics Processing Units* (GPUs) have evolved from simple devices for the display signal preparation into powerful coprocessors supporting the CPU in various ways. Graphics applications such as realistic 3D games are computationally demanding and require a large number of complex algebraic operations for each update of the display image. Therefore, today's graphics hardware contains a large number of programmable processors which are optimized to cope with this high workload of vector, matrix, and symbolic computations in a highly parallel way. In terms of peak performance, the graphics hardware has outperformed state-of-the-art multi-core CPUs by a large margin.

We focus on exploiting the computational power of GPUs for data mining. It has been shown that many data mining algorithms, including clustering can be supported by a powerful database primitive: The *similarity join* [9]. This operator yields as result all pairs of objects in the database having a distance of less than some predefined threshold ϵ_{SJ}. To show that also the more complex basic operations of similarity search and data mining can be supported by novel parallel algorithms specially designed for the GPU, we present in Section 7.3 two algorithms for the similarity join, one being a nested block loop join, and one being an indexed loop join. Therefor specialized indexing methods are required because of the highly parallel but restricted programming environment (cf. Section 7.1). In Section 7.2, we propose such an indexing method.

Finally, to demonstrate that highly complex data mining tasks can be efficiently implemented using novel parallel algorithms, we propose parallel versions of two widespread clustering algorithms in Sections 7.4 and 7.5. The first algorithm CUDA-DClust combines the high computational power and the multiple advantages provided by the density-based clustering algorithm DBSCAN [31]. This approach is highly parallel and exploits thus the high number of simple SIMD (Single Instruction Multiple Data) processors of today's graphics hardware. The parallelization which is required for GPUs differs considerably from previous parallel algorithms which have focused on shared-nothing parallel architectures prevalent in server farms. For even further acceleration, we also propose the algorithm CUDA-DClust*, the indexed variant of CUDA-DClust. In addition, we introduce a parallel version of k-means clustering [72] which follows an algorithmic paradigm that is very different from density-based clustering. We demonstrate the superiority of our approaches over the corresponding sequential algorithms on CPU.

All algorithms for the GPU have been implemented using NVIDIA's technology *Compute Unified Device Architecture* (CUDA) [1]. Vendors of graphics hardware have recently anticipated the trend towards general purpose computing on GPU and developed libraries, pre-compilers and application programming interfaces to support GP-GPU applications. CUDA offers a programming interface for the C programming language in which both the *host program* as well as the *kernel functions* are assembled in a single program [1]. The host program is the main program, executed on the CPU. In contrast, the so-called *kernel functions* are executed in a massively parallel fashion on the (hundreds of) processors in the GPU. An analogous technique is also offered by ATI using the brand names Close-to-Metal, Stream SDK, and Brook-GP.

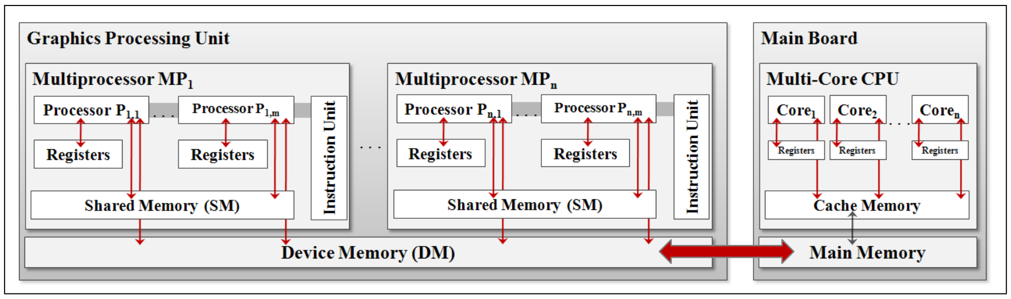


Figure 7.1: Architecture of a GPU.

7.1 The GPU Architecture

From the hardware perspective, a GPU consists of a number of multiprocessors, each of which consists of a set of simple processors which operate in a SIMD fashion, i.e. all processors of one multiprocessor execute in a synchronized way the same arithmetic or logic operation at the same time, potentially operating on different data. For instance, a GPU of the newest generation GT200 (e.g. on the graphics card GeForce GTX280) has 30 multiprocessors, each consisting of eight SIMD-processors, summarizing to a total amount of 240 processors inside one GPU. The computational power sums up to a peak performance of 933 GFLOP/s.

7.1.1 The Memory Model

Apart from some memory units with special purpose in the context of graphics processing (e.g. texture memory), we have three important types of memory, as visualized in Figure 7.1. The *shared memory* (SM) is a memory unit with fast access (at the speed of register access, i.e. no delay). The shared memory is shared among all processors of a multiprocessor. It can be used for local variables but also to exchange information between threads on different processors of the same multiprocessor. It cannot be used for information which is shared among threads on different multiprocessors. The shared memory is fast but very limited in capacity (16 KBytes per multiprocessor).

The second kind of memory is the so-called *device memory* (DM), which is the actual video RAM of the graphics card (also used for frame buffers etc.). The device memory is physically located on the graphics card (but not inside the GPU), is significantly larger than shared memory (typically up to some hundreds of MBytes), but also significantly slower. In particular, memory accesses to device memory cause a typical latency delay of 400-600 clock cycles (on G200-GPU, corresponding to 300-500ns). The bandwidth for transferring data between device memory and GPU (141.7 GB/s on G200) is higher than that of CPU and main memory (about 10 GB/s on current CPUs). The device memory can be used to share information between threads on different multiprocessors. If some threads schedule memory accesses from contiguous addresses, these accesses can be coalesced, i.e. taken together to improve the access speed. A typical cooperation pattern for device memory and shared memory is to copy the required information from device memory to shared memory simultaneously from different threads (if possible, considering coalesced accesses), then to let each thread compute the result on shared memory, and finally, to copy the result back to device memory.

The third kind of memory considered here is the *main memory* which is not part of the graphics card. The GPU has no access to the address space of the CPU. The CPU can only write to or read from device memory using specialized API functions. In this case, the data packets have to be transferred via the Front Side Bus and the PCI-Express Bus. The bandwidth of these bus systems is strictly limited, and therefore, these special transfer operations are considerably more expensive than direct accesses of the GPU to device memory or direct accesses of the CPU to main memory.

7.1.2 The Programming Model

The basis of the programming model of GPUs are lightweight processes which are easy to create and to synchronize, called *threads*. In contrast to CPU processes, the generation and termination of GPU threads as well as context switches between different threads do not cause any considerable overhead either. In typical applications, thousands or even millions of threads are created, for instance one thread per pixel in gaming applications. It is recommended to create much more threads than the number of available SIMD-processors because context switches are also used to hide the latency delay of memory accesses: Particularly an access to the device memory may cause a latency delay of 400-600 clock cycles, and during that time, a multiprocessor may continue its work with other threads.

The CUDA programming library [1] contains API functions to create a large number of threads on GPU, each of which executes a function called *kernel function*. These functions (executed in parallel on the GPU) as well as the *host program* (executed sequentially on the CPU) are defined in an extended syntax of the C programming language. The kernel functions are restricted in functionality (e.g. no recursion). On GPUs the threads do not even have an individual instruction pointer, but this is rather shared by several threads. For this purpose, threads are grouped into so-called *warps* (typically 32 threads per warp). One warp is processed simultaneously on the eight processors of a single multiprocessor (SIMD) using 4-fold pipelining (totalling in 32 threads executed fully synchronously). If not all threads in a warp follow the same execution path, they are executed in a serialized way. The number of SIMD-processors per multiprocessor as well as the concept of 4-fold pipelining is constant on all current CUDA-capable GPUs. Multiple warps are grouped into *thread groups* (TG). It is recommended [1] to use multiples of 64 threads per thread group. The different warps in a thread group (as well as different warps of different thread groups) are executed independently. The threads in one thread group use the same shared memory and may thus communicate and share data via the shared memory. The threads in one group can be synchronized, i.e. all threads wait until all warps of the same group have reached that point of execution. The latency delay of the device memory can be hidden by scheduling other warps of the same or a different thread group whenever one warp waits for an access to device memory. To allow switching between warps of different thread groups on a multiprocessor, it is recommended that each thread uses only a small fraction of the shared memory and registers of the multiprocessor [1].

7.1.3 Atomic Operations

In order to synchronize parallel processes and to ensure the correctness of parallel algorithms, CUDA offers *atomic operations* such as increment, decrement, or exchange (to name just those out of the large collection, which will be needed by our algorithms). Most of the atomic operations work on integer data types in device memory. However, the latest CUDA versions even allow atomic operations in shared memory. If, for instance, some parallel processes share a list as a common resource with concurrent reading and writing from/to the list, it may be necessary to (atomically) increment a counter for the number of list entries (which is in most cases also used as the pointer to the first free list element). Here, atomicity implies the following two requirements:

If two or more threads increment the list counter, then (1) the value counter after all concurrent increments must be equivalent to the value before plus the number of concurrent increment operations. And, (2), each of the concurrent threads must obtain a separate result of the increment operation which indicates the index of the empty list element to which the thread can write its information. Hence, most atomic operations return a result after their execution.

For instance, the operation `atomicInc` has two parameters, the address of the counter to be incremented, and an optional threshold value which must not be exceeded by the operation. The operation works as follows: The counter value at the address is read, and incremented (provided that the threshold is not exceeded). The operation `atomicDec` works in the inverse way. It also commands two parameters, the address of the counter to be decremented, and an threshold value that must not be went below. `atomicDec` reads the counter value at the address, and decrements that value if it is unequal to zero and larger than the given threshold. Finally, the *old* value of the counter (before incrementing/decrementing) is returned to the kernel method which invoked `atomicInc`/`atomicDec`. If two or more threads (of the same or different thread groups) call some atomic operations simultaneously, their result is that of an arbitrary sequentialization of the concurrent operations.

The operation `atomicCas` works in an analogous way. It performs a Compare-and-Swap operation. It has three parameters, an address, a compare value and a swap value. If the value at the address equals the compare value, the value at the address is replaced by the swap value. In every case, the old value at the address (before swapping) is returned to the invoking kernel method.

7.2 An Index Structure for the GPU

Many data mining algorithms for problems like classification, regression, clustering, and outlier detection use similarity queries as a building block. In many cases, these similarity queries even represent the largest part of the computational effort of the data mining tasks. Hence, efficiency is of high importance here. Similarity queries are defined as follows: Given is a database $DS = \{x_1, \cdots x_n\} \subseteq \mathbb{R}^d$ of a number n of vectors from a d-dimensional space, and a query object $q \in \mathbb{R}^d$. We distinguish between two different kinds of similarity queries, the range queries and the nearest neighbor-queries:

Definition 22 (Range Query) *Let $\epsilon_{SJ} \in \mathbb{R}_0^+$ be a threshold value. The result of the **range query** is the set of the following objects:*

$$N_{\epsilon_{SJ}}(q) = \{x \in DS : \quad ||x - q|| \leq \epsilon_{SJ}\},$$

where $||x - q||$ is an arbitrary distance function between two feature vectors x and q, e.g. the Euclidean distance.

Definition 23 (Nearest Neighbor Query) *The result of a **nearest neighbor query** is the set:*

$$NN(q) = \{x \in DS : \quad \forall x' \in DS : \quad ||x - q|| \leq ||x' - q||\}.$$

Definition 23 can also be generalized for the case of the k-nearest neighbor query $(NN_k(q))$, where a number k of nearest neighbors of the query object q is re-

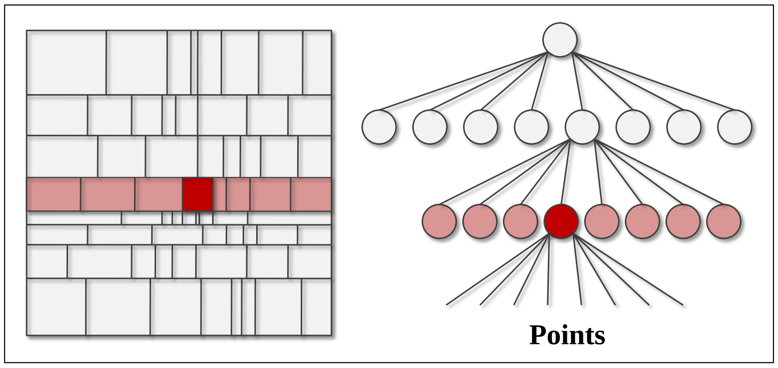


Figure 7.2: Index structure for GPU.

trieved.

The performance of similarity queries can be significantly improved if a multi-dimensional index structure supports the similarity search. Our index structure needs to be traversed in parallel for many search objects using the kernel function. Since kernel functions do not allow any recursion, and as they need to have small storage overhead by local variables etc., the index structure must be kept very simple as well. To achieve a good compromise between simplicity and selectivity of the index, we propose a data partitioning method with a constant number of directory levels. The first level partitions the dataset DS according to the first dimension of the data space, the second level according to the second dimension, and so on. Therefore, before starting the actual data mining method, some transformation technique should be applied which guarantees a high selectivity in the first dimensions (e.g. Principal Component Analysis, Fast Fourier Transform, Discrete Wavelet Transform, etc.). Figure 7.2 shows a simple, 2-dimensional example of a 2-level directory (plus the root node which is considered as level-0), similar to [57, 66]. The fanout of each node is eight. In our experiments in Sections 7.3.3 and 7.4.3, we used a 3-level directory with fanout 16.

Before starting the actual data mining task, our simple index structure must be constructed in a bottom-up way by fractionated sorting of the data: First, the dataset is sorted according to the first dimension, and partitioned into the specified

number of quantile partitions. Then, each of the partitions is sorted individually according to the second dimension, and so on. The boundaries are stored using simple arrays which can be easily accessed in the subsequent kernel functions. In principle, this index construction can already be done on GPU, because efficient sorting methods for GPU have been proposed [38]. Since bottom up index construction is typically not very costly compared to the data mining algorithm, our method performs this preprocessing step on CPU.
When transferring the dataset from the main memory into the device memory in the initialization step of the data mining method, our new method has additionally to transfer the directory (i.e. the arrays in which the coordinates of the page boundaries are stored). Compared to the complete dataset, the directory is always small.

The most important change in the kernel functions in our data mining methods regards the determination of the ϵ_{SJ}-neighborhood of some given seed object q, which is done by exploiting SIMD-parallelism inside a multiprocessor. In the non-indexed version, this is done by a set of threads (inside a thread group) each of which iterates over a different part of the (complete) dataset. In the indexed version, one of the threads iterates in a set of nested loops (one loop for each level of the directory) over those nodes of the index structure which represent regions of the data space which are intersected by the neighborhood-sphere of $N_{\epsilon_{SJ}}(q)$. In the innermost loop, we have one set of points (corresponding to a data page of the index structure) which is processed by exploiting the SIMD-parallelism, like in the non-indexed version.

7.3 Performing the Similarity Join Algorithm on GPU

The *similarity join* is a basic operation of a database system designed for similarity search and data mining on feature vectors. In such applications, we are given a database DS of objects which are associated with a vector from a multidimensional space, the feature space. The similarity join determines pairs of objects which are similar to each other. The most widespread form is the ϵ_{SJ}-join which determines those pairs from $DS \times DS$ which have a Euclidean distance of no more than a user-defined radius ϵ_{SJ}:

Definition 24 (Similarity Join) *Let $DS \subseteq \mathbb{R}^d$ be a set of feature vectors of a d-dimensional vector space and $\epsilon_{SJ} \in \mathbb{R}_0^+$ be a threshold. Then the **similarity join** is the following set of pairs:*

$$\textit{SimJoin}(DS, \epsilon_{SJ}) = \{(x, x') \in (DS \times DS) : \quad ||x - x'|| \leq \epsilon_{SJ}\}$$

If x and x' are elements of the same set, the join is a *similarity self-join*. Most algorithms including the method proposed in this thesis can also be generalized to the more general case of non-self-joins in a straightforward way. Algorithms for a similarity join with nearest neighbor predicates have also been proposed.

The similarity join is a powerful building block for similarity search and data mining. It has been shown that important data mining methods such as clustering and classification can be based on the similarity join. Using a similarity join instead of single similarity queries can accelerate data mining algorithms by a high factor [9].

7.3.1 Similarity Join Without Index Support

The baseline technique to process any join operation with an arbitrary join predicate is the nested loop join (NLJ) which performs two nested loops, each enumerating all points of the dataset. For each pair of points, the distance is calculated and compared to ϵ_{SJ}. The pseudocode of the sequential version of NLJ is given in Algorithm 2.

It is easily possible to parallelize the NLJ, e.g. by creating an individual thread for each iteration of the outer loop. The kernel function then contains the inner loop, the distance calculation and the comparison. During the complete run of the kernel function, the current point of the outer loop is constant, and we call this point the *query point* q of the thread, because the thread operates like a similarity

Algorithm 2 Sequential algorithm for the nested loop join.

```
algorithm sequentialNLJ(dataset $DS$)
for all $q \in DS$ do
  for all $x \in DS$ do
    if $dist(x, q) \leq \epsilon_{SJ}$ then
      report $(x, q)$ as a result pair or do some further processing on $(x, q)$
    end if
  end for
end for
```

query, in which all database points with a distance of no more than ϵ_{SJ} from q are searched. The query point q is always held in a register of the processor.

Our GPU allows a truly parallel execution of a number m of incarnations of the outer loop, where m is the total number of Arithmetic Logic Units (ALU) of all multiprocessors (i.e. the warp size 32 times the number of multiprocessors). Moreover, all the different warps are processed in a quasi-parallel fashion, which allows to operate on one warp of threads (which is ready-to-run) while another warp is blocked due to the latency delay of a device memory access of one of its threads.

The threads are grouped into thread group, which share the shared memory. In our case, the shared memory is particularly used to physically store for each thread group the current point x of the inner loop. Therefore, a kernel function first copies the current point x from the device memory into the shared memory, and then determines the distance of x to the query point q. The threads of the same warp are running perfectly simultaneously, i.e. if these threads are copying the same point from device memory to shared memory, this needs to be done only once (but all threads of the warp have to wait until this relatively costly copy operation is performed). However, a thread group may (and should) consist of multiple warps. To ensure that the copy operation is only performed once per thread group, it is necessary to synchronize the threads of the thread group before and after the copy operation using the API function `synchronize()`. This API function blocks all threads in the same thread group until all other threads (of other warps) have reached the same point of execution. The pseudocode for this algorithm is presented in Algorithm 3.

Algorithm 3 Parallel algorithm for the nested loop join on GPU.

```
algorithm GPUsimpleNLJ(dataset $DS$)  // host program executed on CPU
deviceMem float $DS'[][] := DS[][]$;          // allocate memory in DM $DS$
threads := n;                                 // number of points in $DS$
tpg := 64;                                    // threads per group
startThreads(simpleNLJKernel, threads, tpg); // one thread per point
waitForThreadsToFinish();


kernel simpleNLJKernel(int threadID)
register float $q[] := DS'$[threadID]$[]$;        // copy q from DM into the register
                                              // and use it as query point q
                                              // index is determined by threadID
for $i := 0$ to $n - 1$ do
  synchronizeThreadGroup();
  shared float $x[] := DS'[i][]$;                 // copy x from DM to SM
  synchronizeThreadGroup();   // all threads of TG can work with x
  if $dist(x, q) \leq \epsilon_{SJ}$ then
    report $(x, q)$ as a result pair using synchronized writing
    or do some further processing on $(x, q)$ directly in kernel
  end if
end for
```

7.3.2 An Indexed Parallel Similarity Join Algorithm on GPU

If the dataset does not fit into device memory, a simple partitioning strategy can be applied. It must be ensured that the potential join partners of an object are within the same partition as the object itself. Therefore, overlapping partitions of size $2 \times \epsilon_{SJ}$ can be created.

The performance of the NLJ can be significantly improved if an index structure is available as proposed in Section 7.2. On sequential processing architectures, the indexed NLJ leaves the outer loop unchanged. The inner loop is replaced by an index-based search retrieving candidates that may be join partners of the current object of the outer loop. The effort of finding these candidates and refining them is often orders of magnitude smaller compared to the non-indexed NLJ. When parallelizing the indexed NLJ for the GPU, we follow the same paradigm as in the last section, to create an individual thread for each point of the outer loop. It is beneficial for the performance, if points having a small distance to each other are collected in the same warp and thread group, because for those points, similar paths in the index structure are relevant.

After index construction, we have not only a directory in which the points are organized in a way that facilitates search. Moreover, the points are now clustered in the array, i.e. points which have neighboring addresses are also likely to be close together in the data space (at least when projecting on the first few dimensions). Both effects are exploited by our join algorithm displayed in Algorithm 4.

Instead of performing an outer loop like in a sequential indexed NLJ, our algorithm now generates a large number of threads: One thread for each iteration of the outer loop (i.e. for each query point q). Since the points in the array are clustered, the corresponding query points are close to each other, and the join partners of all query points in a thread group are likely to reside in the same branches of the index as well. Our kernel method now iterates over three loops, each loop for one index level, and determines for each partition if the point is inside the partition or, at least no more distant to its boundary than ϵ_{SJ}. The corresponding subnode is accessed if the corresponding partition is able to contain join partners of the

Algorithm 4 Algorithm for the similarity join on GPU with index support.

```
algorithm GPUIndexedJoin(dataset $DS$)
deviceMem index idx := indexAndSort($DS$); // changes ordering of points
threads := n;
tpg := 64;
groups := threads/tpg;
for $i := 1$ to $groups$ do
    deviceMem float blockbounds[$i$][] := calcBlockBounds($DS$, index);
end for
deviceMem float $DS'[][] := DS[][]$;
startThreads(indexedJoinKernel, threads, tpg); // one per point
waitForThreadsToFinish();


kernel indexedJoinKernel(int threadID, int blockID)
register float $q[] := DS'$[threadID][];          // copy q from DM into register
shared float myblockbounds[] := blockbounds[blockID][];
for $x_i := 0$ to $indexsize.x$ do
    if indexPageIntersectsBoundsD1(idx, myblockbounds, $x_i$) then
        for $y_i := 0$ to $indexsize.y$ do
            if indexPageIntersectsBoundsD2(idx, myblockbounds, $x_i$, $y_i$)
            then
                for $z_i := 0$ to $indexsize.z$ do
                    if indexPageIntersectsBoundsD3(idx, myblockbounds,
                    $x_i$, $y_i$, $z_i$) then
                        for $w := 0$ to $indexPageSize$ do
                            synchronizeThreadGroup();
                            shared float $p[] :=$
                            getPointFromIndexPage(idx,$DS', x_i, y_i, z_i, w$);
                            synchronizeThreadGroup();
                            if $dist(p, q) \leq \epsilon_{SJ}$ then
                                report $(p, q)$ as a result pair using synchronized writing
                            end if
                        end for
                    end if
                end for
            end if
        end for
    end if
end for
```

current point of the thread. When considering the warps which operate in a fully synchronized way, a node is accessed, whenever at least one of the query points of the warps is close enough to (or inside) the corresponding partition. For both methods, indexed and non-indexed nested loop join on GPU, we need to address the question how the resulting pairs are processed. Often, for example to support density-based clustering (cf. Section 7.4), it is sufficient to return a counter with the number of join partners. If the application requires to report the pairs themselves, this is easily possible by a buffer in device memory which can be copied to CPU after the termination of all kernel threads. The result pairs must be written into this buffer in a synchronized way to avoid that two threads write simultaneously to the same buffer area. The CUDA API provides atomic operations (such as atomic increment of a buffer pointer) to guarantee this kind of synchronized writing. Buffer overflows are also handled by our similarity join methods. If the buffer is full, all threads terminate and the work is resumed after the buffer is emptied by CPU.

7.3.3 Experimental Evaluation

We performed various experiments on synthetic datasets. The implementation for all variants is written in C and all experiments are performed on a workstation with Intel Core 2 Duo CPU E4500 2.2 GHz and 2 GB RAM which is supplied with a Gainward NVIDIA GeForce GTX280 GPU (240 SIMD-processors) with 1GB GDDR3 SDRAM. The performance of the similarity join on the GPU, is validated by the comparison of four different variants for executing the similarity join:

1. NLJ on CPU
2. NLJ on CPU with index support (as described in Section 7.2)
3. NLJ on GPU
4. NLJ on GPU with index support (as described in Section 7.2)

For each version we determine the speedup factor by the ratio of CPU runtime and GPU runtime. For this purpose we generated three 8-dimensional synthetic datasets of various sizes (up to ten million (m) points) with different data distributions, as summarized in Table 7.1. Dataset DS_1 contains uniformly distributed data. DS_2 consists of five Gaussian clusters which are randomly distributed in feature space. Similar to DS_2, DS_3 is also composed of five Gaussian clusters, but the clusters are linearly correlated. The threshold ϵ_{SJ} was selected to obtain a join result where each point was combined with one or two join partners on average.

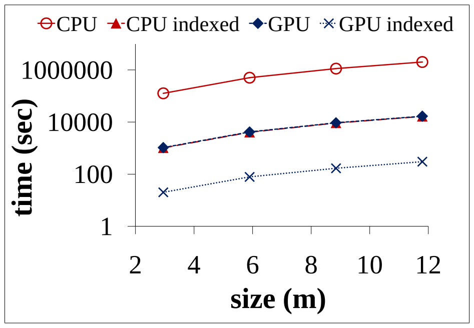
Table 7.1: Synthetic datasets used for the evaluation of the similarity join on GPU.

	SIZE	DATA DISTRIBUTION
DS_1	3m - 10m points	uniform distribution
DS_2	250k - 1m points	normal distribution, Gaussian clusters
DS_3	250k - 1m points	normal distribution, Gaussian clusters

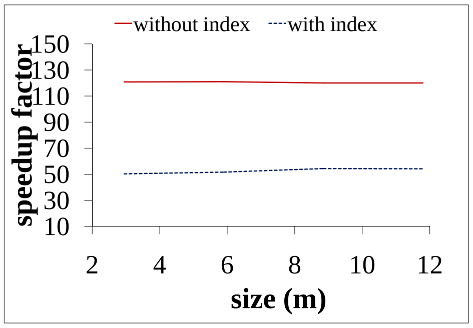
Evaluation of the Dataset Sizes

Figures 7.3(a) to 7.3(f) display the runtime in seconds and the corresponding speedup factors of NLJ on CPU with/without index support and NLJ on the GPU with/without index support in logarithmic scale for all three datasets DS_1, DS_2 and DS_3. The time needed for data transfer from CPU to GPU and back as well as the (negligible) index construction time has been included. The tests on dataset DS_1 were performed with $\epsilon_{SJ} = 0.125$, and $\epsilon_{SJ} = 0.588$ on DS_2 and DS_3 respectively.

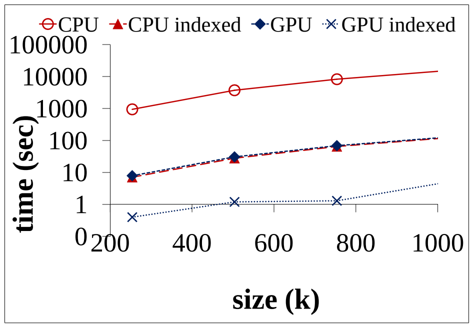
NLJ on the GPU with index support performs best in all experiments, independent of the data distribution or size of the dataset. Note that, due to massive parallelization, NLJ on the GPU without index support outperforms CPU without index by a large factor. For example on DS_3 (1m points of normal distributed data with Gaussian clusters) we achieve a speedup factor of approximately 120, shown in Figure 7.3(f). The GPU algorithm with index support outperforms the



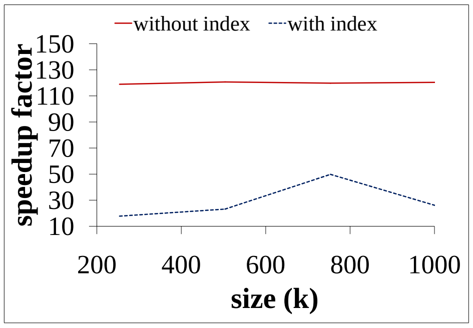
(a) Runtime on dataset DS_1.



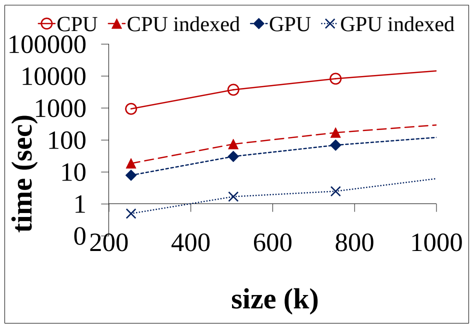
(b) Speedup on dataset DS_1.



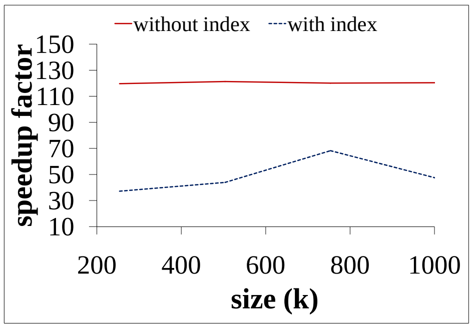
(c) Runtime on dataset DS_2.



(d) Speedup on dataset DS_2.



(e) Runtime on dataset DS_3.



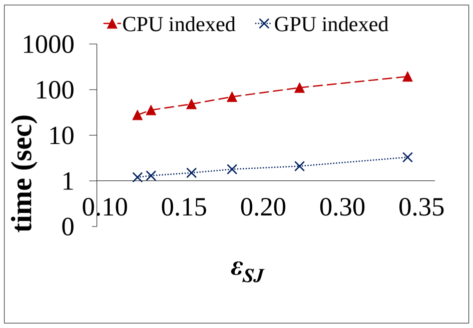
(f) Speedup on dataset DS_3.

Figure 7.3: Evaluation of the NLJ on CPU and GPU with and without index support w.r.t. the size of different datasets.

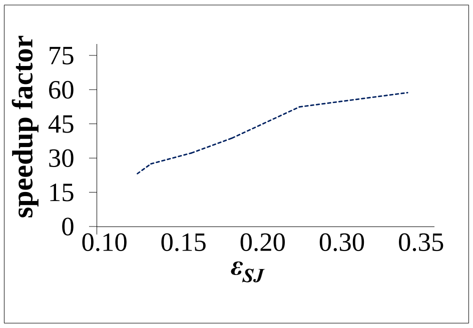
corresponding CPU algorithm (with index) by a factor of 25 on dataset DS_2 (cf. Figure 7.3(d)). Remark that for example the overall improvement of the indexed GPU algorithm on dataset DS_2 over the non-indexed CPU version is more than 6,000. Hence, these results demonstrate the potential of boosting performance of database operations with designing specialized index structures and algorithms for the GPU.

Evaluation of the Parameter ϵ_{SJ}

In these experiments we test the impact of the parameter ϵ_{SJ} on the performance of NLJ on the GPU with index support and use the indexed implementation of NLJ on CPU as benchmark. All tests are performed on dataset DS_2 with a fixed size of 500k data points. The parameter ϵ_{SJ} is evaluated in a range from 0.125 to 0.333.
Figure 7.4(a) shows that the runtime of NLJ on the GPU with index support increases for larger ϵ_{SJ} values. However, the GPU version outperforms the CPU implementation by a large factor (cf. Figure 7.4(a)), that is proportional to the value of ϵ_{SJ}. In this evaluation the speedup ranges from 20 for $\epsilon_{SJ} 0.125$ to almost 60 for $\epsilon_{SJ} = 0.333$.



(a) Runtime on dataset DS_2.



(b) Speedup on dataset DS_2.

Figure 7.4: Evaluation of the NLJ on CPU and GPU with and without index support w.r.t. ϵ_{SJ}.

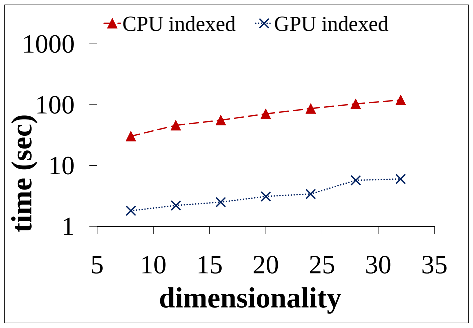
Evaluation of the Dimensionality

These experiments provide an evaluation w.r.t. the dimensionality of the data. As in the experiments for the evaluation of the parameter ϵ_{SJ}, we use again the indexed implementations both on CPU and GPU and perform all tests on dataset DS_2 with a fixed number of 500k data objects. The dimensionality is evaluated in a range from eight to 32. In addition, we performed these experiments with two different settings, namely $\epsilon_{SJ} = 0.588$ and $\epsilon_{SJ} = 1.429$.

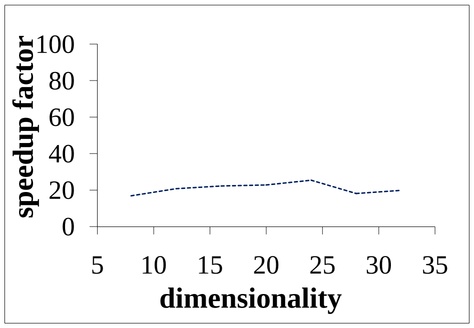
Figure 7.5 illustrates that NLJ on the GPU outperforms the benchmark method on CPU by factors of about 20 for $\epsilon_{SJ} = 0.588$ to approximately 70 for $\epsilon_{SJ} = 1.429$. This order of magnitude is relatively independent of the data dimensionality. As in our implementation the dimensionality is already known at compile time, optimization techniques of the compiler have an impact on the performance of the CPU version as can be seen especially in Figure 7.5(c). However the dimensionality also affects the implementation on the GPU, because higher dimensional data come along with a higher demand of shared memory. This overhead affects the number of threads that can be executed in parallel on the GPU.

7.3.4 Conclusions

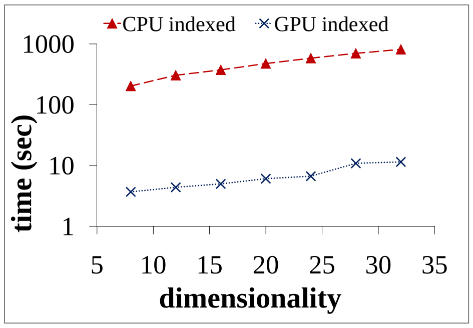
In this section, we have addressed the question, how similarity join algorithms can be efficiently executed on the GPU. We have proposed an index structure which is particularly suited to be held in the Device Memory of the GPU and which can be accessed by kernel functions of the GPU in an efficient way. Since graphics processors correspond to a highly parallel architecture consisting of up to some hundreds of single programmable processors, we have also shown how similarity join algorithms can be parallelized. We have proposed two efficient similarity join algorithms, one based on the nested loop join and the other based on the indexed nested loop join. Both algorithms are particularly dedicated for GPU processing and outperform the sequential algorithms by a large factor.



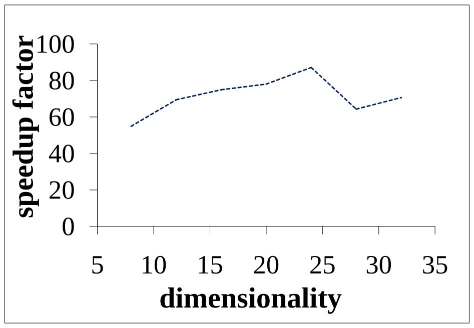
(a) Runtime on dataset DS_2 ($\epsilon_{SJ} = 0.588$).



(b) Speedup on dataset DS_2 ($\epsilon_{SJ} = 0.588$).



(c) Runtime on dataset DS_2 $\epsilon_{SJ} = 1.429$.



(d) Speedup on dataset DS_2 $\epsilon_{SJ} = 1.429$.

Figure 7.5: Evaluation of the NLJ on CPU and GPU with and without index support w.r.t. the dimensionality.

7.4 Density-based Clustering Using GPU

In this section, we describe the algorithm CUDA-DClust – a parallel variant of density-based clustering for the execution on GPUs. First, we introduce the classic definitions of density-based clustering, based on the famous algorithm DBSCAN [31]. And afterwards, we describe our ideas behind CUDA-DClust in detail.

7.4.1 Foundations of Density-based Clustering

The idea of density-based clustering is that clusters are areas of high point density, separated by areas of significantly lower point density that can be formalized using two parameters, called $\epsilon \in \mathbb{R}^+$ and *MinPts* $\in \mathbb{N}^+$. The central notion is the *core object*. A data object P is called a core object of a cluster, if at least *MinPts* objects (including P itself) are in its ϵ-neighborhood denoted by $N_\epsilon(P)$, which corresponds to a sphere of radius ϵ. Formally:

Definition 25 (Core Object) *Let DS be a set of n objects from $\mathbb{R}^d$, $\epsilon \in \mathbb{R}^+$ and MinPts $\in \mathbb{N}^+$. An object $P \in DS$ is a **core object**, iff*

$$|N_\epsilon(P)| \geq \textit{MinPts}, \textit{ where } N_\epsilon(P) = \{Q \in DS : ||P - Q|| \leq \epsilon\}.$$

Two objects may be assigned to a common cluster. In density-based clustering this is formalized by the notions *direct density reachability* and *density connectedness*.

Definition 26 (Direct Density Reachability) *Let $P, Q \in DS$. Q is **directly density reachable** from P (in symbols: $P \lhd Q$) iff*

1. *P is a core object in DS, and*
2. *$Q \in N_\epsilon(P)$.*

If P and Q are both core objects, then $P \lhd Q$ is equivalent with $P \rhd Q$. The density connectedness is the transitive and symmetric closure of the direct density reachability:

Definition 27 (Density Connectedness) *Two objects P and Q are* ***density connected*** *(in symbols: $P \bowtie Q$) iff there is a sequence of core objects $(P_1, ..., P_m)$ of arbitrary length m such that*

$$P \rhd P_1 \rhd ... \lhd P_m \lhd Q.$$

In density-based clustering, a cluster is defined as a maximal set of density connected objects:

Definition 28 (Density-based Cluster) *A subset $c \subseteq DS$ is a* ***density-based cluster*** *iff the following two conditions hold:*

1. *Density connectedness: $\forall P, Q \in C : P \bowtie Q$.*
2. *Maximality: $\forall P \in C, \forall Q \in DS \setminus C : \neg P \bowtie Q$.*

The algorithm DBSCAN [31] implements the cluster notion of Definition 28 using a data structure called *seed list SL* containing a set of seed objects for cluster expansion. More precisely, the algorithm proceeds as follows:

1. Mark all objects as *unprocessed*.
2. Consider arbitrary unprocessed object $P \in DS$.
3. If P is core object, assign new cluster-ID C, and do step (4) for all elements $Q \in N_\epsilon(P)$, which do not yet have a cluster-ID:
4. (a) mark element Q with cluster-ID C and
   (b) insert object Q into seed list SL.
5. While SL not $\emptyset$ repeat step (6) for all elements $P' \in S$:
6. If P' is core object, do step (7) for all elements $Q \in N_\epsilon(P')$, which do not yet have any cluster-ID:

7. (a) mark element Q with cluster-ID C and
   (b) insert object Q into seed list SL.

8. If DS still contains unprocessed objects, do step (2).

Since every object of the database is considered only once in Step (2) or (6) exclusively, we have a complexity, which is n times the complexity of $N_\epsilon(P)$ (which is linear in the number of objects n if there is no index structure, and sublinear or even $O(\log(n))$ in the presence of a multi-dimensional index structure).

7.4.2 CUDA-DClust

Now we extend the concept of density-based clustering for special parallel environment provided by the GPU.

Chains: The Main Idea of Parallelization

Since in density-based clustering there is a complex relationship between the data objects and an unknown number of clusters, where two objects share a common cluster whenever they are (directly or indirectly) connected by an unknown number of core objects, it is not possible to define a straightforward parallelization e.g. by assigning an individual thread to each data object, cluster, or dimension. Instead, our approach CUDA-DClust is based on a new concept especially introduced to allow for massive parallelism in density-based clustering, called *chain*. A chain is a set of data objects belonging to a common density-based cluster, formally:

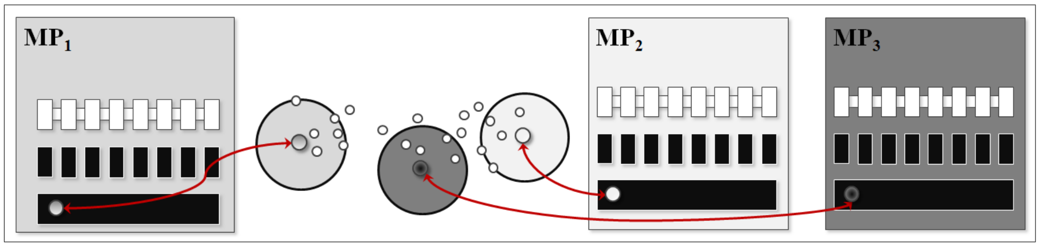
Definition 29 (Density-based Chain) *A subset $C \subseteq DS$ is a **density-based chain** iff the following condition holds:*

$$\forall P, Q \in C : P \bowtie Q.$$

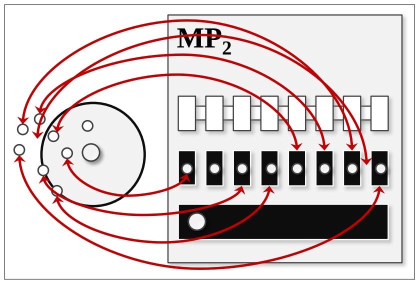
Note that a cluster may be composed of several chains but each chain belongs to one single cluster only. In contrast to Definition 28, we do not require maximality. Using the concept of chains, we basically perform the task of cluster expansion, i.e. the transitive closure computation of the relation of direct density reachability. Thus, a chain can be considered as a tentative cluster with a tentative cluster-ID (which we call *chain-ID*). The idea is, instead of sequentially performing *one* single cluster expansion like, for instance, in DBSCAN, we start many different cluster expansions at the same time via different chains from different starting points. Of course CUDA-DClust has to register every *collision* between two or more chains carefully. A collision means that CUDA-DClust has to notice that two chains actually participate in the same cluster. A detailed description of our collision handling is presented in the following.

We now introduce a concept for book-keeping all information that is essential for collision handling, called *collision matrix*. A collision matrix CM is an upper triangular boolean matrix of size $(p \times p)$, where p is the number of chains that are expanded simultaneously. If a collision of chains i and j is detected, we set $CM_{i,j} :=$ **true** if $i < j$ and $CM_{j,i} :=$ **true** otherwise. Since CM is small, we later run a sequential algorithm to determine transitive collisions. Finally, a valid cluster-ID is assigned to each chain and is distributed to the points in a parallel fashion on GPU. Figure 7.6(a) illustrates parallelism based on the concept of chains. Here, three density-based chains are processed in different thread groups and may, therefore, be computed in a parallel fashion on different MPs of the GPU. Each MP is responsible for the expansion of one chain. The seed point to be expanded is loaded into the shared memory in order to have it efficiently accessible for every thread in this thread group.

Besides the approach of chains, we introduce a second, but equally important idea of parallelization. Whenever a point S is considered for determining its core object property and for marking its neighbors as potential new seed points, all points in $N_\epsilon(S)$ must be determined. Therefor, a number of threads is generated in order to process many potential neighbors P_i of S simultaneously. Note that,



(a) Parallelism at the level of multiprocessors (chains).



(b) Parallelism exemplarily of SIMD-processor MP_2.

Figure 7.6: Two different ideas of parallelization.

as already mentioned in Section 7.1, the generation of a high number of threads in CUDA does not cause any considerable overhead. In contrast, it is of high importance to generate a high number of potentially parallel threads in order to keep the processors in a GPU busy, even in the frequent case of delays due to memory latency. Figure 7.6(b) shows the intra-MP parallelism, exemplarily for MP$_2$ of the example illustrated in Figure 7.6(a). All possible mating partners Q_i of the seed point, which have to be tested whether or not they are in $N_\epsilon(P)$ are processed by different SIMD-processors in MP$_2$. Hence, the coordinates are loaded in the corresponding registers of the SIMD-processors, and each of them computes one distance function between P and Q_i simultaneously. Summarizing, we achieve a high degree of intra-MP and inter-MP parallelism by having one thread group for each in parallel executed chain and one thread for each potential partner point of the current seed point of a chain.

In CUDA, multiples of 32 threads must be grouped into thread groups. As indicated in Section 7.1, threads of the same group are potentially executed in a fully synchronous way in the same MP, may exchange information through the shared memory, and all threads of the same thread group can be synchronized when all threads wait until all other threads of the same thread group have reached the same point of execution. Therefore, it is obviously beneficial to collect all threads belonging to the same chain into one thread group.

The Cluster Expansion Kernel

The cluster expansion kernel is the main kernel method of CUDA-DClust. It performs the actual work, i.e. the determination of the core point property and the transitive expansion of the clusters. We generate a high number of threads, all executing the cluster expansion kernel. Some other kernel methods for supportive and cleaning tasks will be described in short later.

Consider the pseudocode presented in Algorithm 5a. Each thread group, generated by the cluster expansion kernel starts with the ID of the seed point P that is going to be processed. Its main tasks are, determine the neighbor-points, assert the core object property of P, mark the neighbors as chain members, and record potential collisions in CM. A thread group starts by reading the coordinates of P into the shared memory, and determining the minimum bounding rectangle MBR$_\epsilon(P)$ of $N_\epsilon(P)$ allowing d-fold parallelism, where d is the dimension of the data space. Then the threads are synchronized and a loop encounters all points $Q \in$ PointSet from DS and considers them as potential neighbors in $N_\epsilon(P)$. This is done by all threads in the thread group allowing a maximum degree of intra-group parallelism. Therefore, this loop starts with the point with index *thread-ID*, and in each iteration, exactly the number of threads per thread group is added to the index of the considered point Q. The further processing of Q by the procedure `processObject` depends on the core object property of P as clusters are expanded from core objects only. However, at an early stage it may be unclear if P is a core object. In any case, the distance δ between P and Q is determined. If

Algorithm 5a The cluster expansion kernel.

```
// some global information available on DM:
float pointSet [$n$][$d$];
int pointState [$n$];          // initialized with UNPROCESSED
                               // it will contain the ChainID/ClusterID of the object
                               // but also special values like NOISE
float $\epsilon$;
int minPts;
int threadGroupSize, numChains, maxSeedLength;
int seedList [numChains][maxSeedLength];
int seedLength [numChains];
boolean collisionMatrix[numChains][numChains];     // initialized with FALSE


kernel clusterExpansionKernel(int threadGroupId, int threadID)
chainID ≡ threadGroupID;
mySeedList [] ≡ seedList [chainID][];
mySeedLength ≡ seedLength [chainID];
shared int neighborBuffer [minPts];
shared int neighborCount = 0;

mySeedLength := mySeedLength − 1;
shared int seedPointID := mySeedList[mySeedLength];
shared float $P$[] := pointSet [seedPointID][];        // copy from DM
synchronizeThreads();
for $i := threadID$ to $n - 1$ $step\ threadGroupSize$ do
  processObject($i$);
end for
synchronizeThreads();
if neighborCount $\geq$ minPts then
  pointState[seedPointID] := chainID;
  for $i := 0$ to $minPts - 1$ do
    markAsCandidate(neighborBuffer[$i$]);  // reconsider neighbors
                                           // in quarantine buffer
  end for
else
  pointState[seedPointID] := NOISE;
end if
```

Algorithm 5b Procedure used by the cluster expansion kernel.

```
procedure processObject(int $i$)
register float $Q$ [] := pointSet[$i$][];
register float $\delta$ :=Distance($P, Q$);
if $\delta \leq \epsilon$ then
  register int $h$ = atomicInc(neighborCount);
  if $h \geq$ minPts then
    markAsCandidate($i$);
  else
    neighborBuffer[$h$] $:= i$;  // $P$ not yet confirmed as core object,
                                 // put Q in quarantine buffer
  end if
end if
```

Algorithm 5c Procedure used by the cluster expansion kernel.

```
procedure markAsCandidate(int $i$)
register int oldState = atomicCAS(pointState[$i$],UNPROCESSED,chainID);
if $oldState = UNPROCESSED$ then
  register int $h$ := atomicInc(mySeedLength, maxSeedLength);
  if $h < maxSeedLength$ then
    seedlist[$h$] $:= i$;
  end if
else
  if $oldState \neq NOISE \wedge oldState \neq chainID$ then
    if $oldState < i$ then
      collisionMatrix[oldState][$i$] := true;        // collision!
    else
      collisionMatrix[$i$][oldState] := true;
    end if
  end if
end if
```

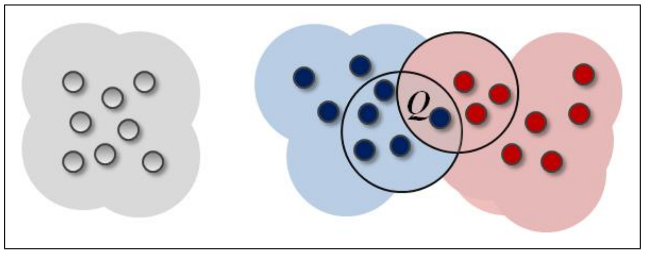


Figure 7.7: A collision of two chains.

δ exceeds ϵ nothing needs to be done. Otherwise, we have to distinguish: If P is already confirmed to be a core object, then we can immediately mark Q as a chain member, but have to perform further operations, described in the next paragraph. Otherwise, we increment (cf. Section 7.1.3) the counter *neighborCount*, which is individual to P atomically and take Q into a temporary list called *neighborBuffer* with at most $MinPts$ entries. We can say, that Q is taken under *quarantine* until the core status of P is really known. Whenever we mark an object as a chain member, we have to perform the following actions: Check if the object is already a member of the same or a different chain. In the first case, nothing needs to be done. Otherwise we have a *collision* of two chains. That means, we have to note that the two chain-IDs actually belong to the same cluster-ID. Figure 7.7 shows a typical situation of such a scenario. Note that three situations are possible: First, the point Q was originally marked by the blue chain in the middle and later the red chain recognized the collision, or vice versa. The third option is, that the blue and the red chain tried to mark the collision object simultaneously. To cope with that case, we need atomic operations again. To label the neighbor points we use `atomicCAS` (compare-and-swap, cf. Section 7.1.3). This operation checks if a certain value is stored at the corresponding address. If the object had the status *UNPROCESSED* then `atomicCAS` replaces the old label by the current chain-ID and CUDA-DClust tries to include Q in the seed list of our chain. Otherwise, the label remains and the old value is returned to the kernel. Hence, we can decide unambiguously whether there was a collision or not.

Management of the Seed List

The seed list SL is the central data structure of density-based clustering. Its purpose is to provide a waiting queue for those objects, which are directly density reachable from any object that has already been confirmed to be a core object of the current cluster, also named *candidates*. The core object property of the candidates in the queue will be checked later, and a confirmed core object P will be replaced by those objects, which are directly density reachable from P. SL is a very simple data structure: An unpriorized queue for a set of objects with the only operations of storing and fetching an arbitrary object. In sequential density-based clustering algorithms such as DBSCAN or OPTICS, only one seed list must be maintained. In contrast, CUDA-DClust simultaneously performs cluster expansions of a moderately high number of chains, and, therefore, multiple seed lists must be managed, which potentially causes space problems. The index number of the actual seed list that is used by a thread group can be determined from its TG-ID (which simultaneously serves as Chain-ID). To facilitate the readability of the code, at the beginning of the kernel ClusterExpansion, these simplifications are introduced by equivalence associations using the symbol $\equiv$. These are only meant as shortcuts, and no copying of data is needed.

For the insertion of objects into SL the operation `atomicInc` (for increment) has to be used because some candidates may be simultaneously inserted by different threads of the same thread group. The space in each seed list in the device memory is limited by a parameter (*maxSeedLength*). In our experiments, we used 1,024 points per seed list. In case of a list overflow, new points are only marked by the current chain-ID but not inserted into the seed list. We use two different counters to keep track of the number of discarded candidates. Using a special kernel method, called *SeedRefill Kernel* we can search for candidates that have been discarded.

Load Balancing

Since a number of seeds is expanded simultaneously and a number of chains is processed in parallel, it naturally happens that one of the chains is completed before the others. In this case, we start a new chain from an unprocessed object. If no such object is available, we split a chain, i.e. one of the objects from SL of a randomly selected chain is removed and we start a new chain with that candidate. Thereby, an almost constant number of threads working in the system is guaranteed, all with approximately the same workload.

The Main Program for CPU

Apart from initialization, the main program consist merely of a loop starting three different kernel methods on the GPU, until no more unprocessed data objects exist any more:

1. Transfer the dataset from main memory to device memory;
2. Create *NumChains* new chains from randomly selected points of DS;
3. StartKernel (ClusterExpansion);
4. Transfer the states of the chains from device memory to main memory;
5. If necessary, StartKernel (NewSeeds);
6. If necessary, StartKernel (SeedRefill);
7. If unprocessed objects exist, continue with step (3);
8. Transfer the result (cluster-IDs) from device memory to main memory.

CUDA-DClust* - The Indexed Version of CUDA-DClust

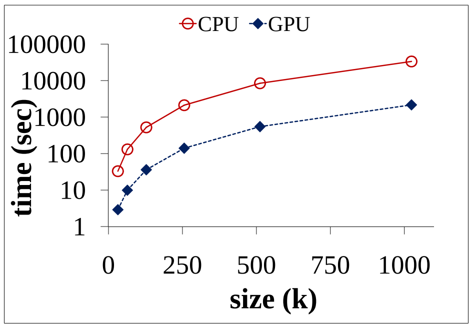
The performance of CUDA-DClust can be significantly improved by the multi-dimensional index structure presented in Section 7.2, supporting the search of points in the ϵ-neighborhood of some object $P \in N_\epsilon(P)$. To receive the indexed version of CUDA-DClust named CUDA-DClust*, our index structure must be constructed in a bottom-up way by fractionated sorting of the data DS before starting the actual clustering method. When transferring DS from the main memory into the device memory in the initialization step of CUDA-DClust, CUDA-DClust* has to transfer the directory additionally. Compared to the size of DS, the directory is always small. The most important change in our cluster expansion kernel regards the determination of $N_\epsilon(P)$ of some given seed object P, which is done by exploiting SIMD-parallelism inside a MP. In CUDA-DClust, this was done by a set of threads inside a thread group, each of which iterated over a different part of DS. In CUDA-DClust*, one of the threads iterates in a set of nested loops over those nodes of the index structure, which represent regions of DS that intersect $N_\epsilon(P)$. In the innermost loop, we have one set of points corresponding to a data page of the index structure, which is processed by exploiting the SIMD-parallelism, like in CUDA-DClust.

7.4.3 Experimental Evaluation

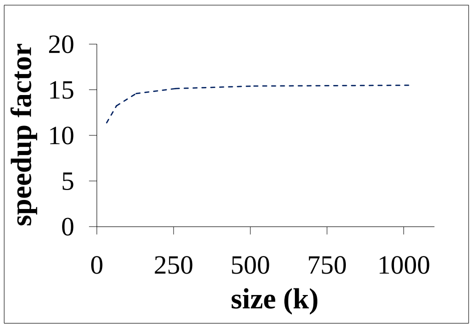
To evaluate the performance of density-based clustering on GPU, we execute different experiments. First, we evaluate CUDA-DClust vs. DBSCAN without index support and CUDA-DClust* vs. DBSCAN with index support w.r.t. the size of the datasets. Second, we analyze the impact of the parameters $MinPts$ and ϵ. Finally, we perform an evaluation w.r.t. the data dimensionality. All test data are synthetic feature vectors containing 20 randomly generated Gaussian clusters. Unless otherwise mentioned, the algorithms are parameterized with $\epsilon = 0.05$ and $MinPts = 4$. These parameters are used in an analogous way like the correspondent parameters of the sequential DBSCAN algorithm. The time needed for index construction is not included in these experiments, but evaluated separately. A detailed presentation of the complete runtime profile of CUDA-DClust is given in the rear part of this Section. The implementation for all variants is written in C++ and all experiments are performed on a workstation with Intel Core 2 Duo CPU E4500 2.2 GHz and 2 GB RAM, which is supplied with a Gainward NVIDIA GeForce GTX280 GPU (240 SIMD-processors) with 1GB GDDR3 SDRAM. As a benchmark we apply a single-threaded implementation of DBSCAN on the CPU, that uses the same index structure as CUDA-DClust for all experiments with index support.

CUDA-DClust vs. DBSCAN

Figure 7.8 displays the runtime in seconds of CUDA-DClust compared to the DBSCAN implementation on the CPU without index support for various data sizes in logarithmic scale and the corresponding speedup factor. Due to massive parallelization, CUDA-DClust outperforms CPU without index by a large factor that is even growing with the number of points n. The speedup ranges from ten for 30k points up to 15 for 1m points. Note that the calculation on GPU takes 40 minutes, compared to nine hours on CPU.



(a) Runtime.

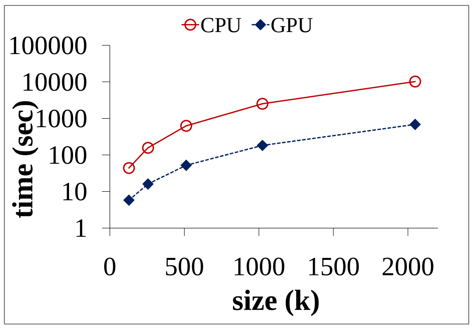


(b) Speedup.

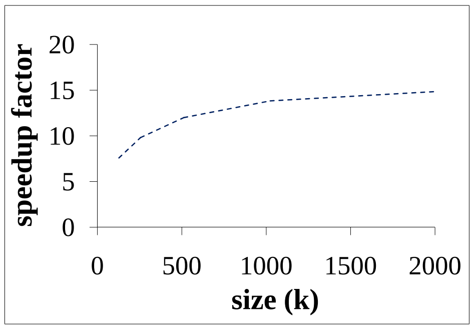
Figure 7.8: Evaluation of density-based clustering on GPU without index support w.r.t. the size.

CUDA-DClust* vs. Indexed DBSCAN

The following experiments are performed with index support as introduced in Section 7.2. The runtime of CUDA-DClust* and the indexed version of DBSCAN on CPU and speedup are presented in Figure 7.9. CUDA-DClust* outperforms the benchmark again by a large factor, that is proportional to the size of the dataset. In this evaluations the speedup ranges from 3.5 for 30k points to almost 15 for 2m points. A guaranteed speedup factor of at least ten is obtained by datasets consisting of more than 250k points.



(a) Runtime.



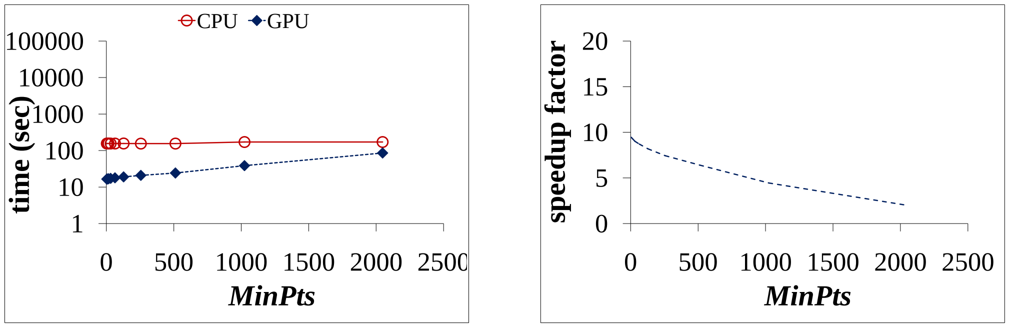
(b) Speedup.

Figure 7.9: Evaluation of density-based clustering on GPU with index support w.r.t. the size.

Impact of the Parameter MinPts

In these experiments we test the impact of the parameter $MinPts$ on the performance of CUDA-DClust* against the indexed version of DBSCAN on CPU. We use a synthetic dataset with a fixed number of about 260k 8-dimensional points and choose $\epsilon = 0.05$. The parameter $MinPts$ is evaluated in a range from 4 to 2,048. Figure 7.10 shows that the runtime of CUDA-DClust* increases for larger $MinPts$ values. For $4 \leq MinPts \leq 512$ the speedup factor remains relatively stable between ten and five. Then the speedup decreases significantly. However

CUDA-DClust* outperforms the implementation on CPU even for large $MinPts$-values by a factor of two. The speedup decline can be explained by a closer look at the implementation details of CUDA-DClust* (cf. Section 7.4.2). It can be seen that the first $MinPts - 1$ found neighbors have to be buffered, because they can not be marked before it is guaranteed that the seed point is also a core point. As all threads that are executed in parallel have to share the limited shared memory, less threads can be executed in parallel if a high number of points have to be buffered in addition.

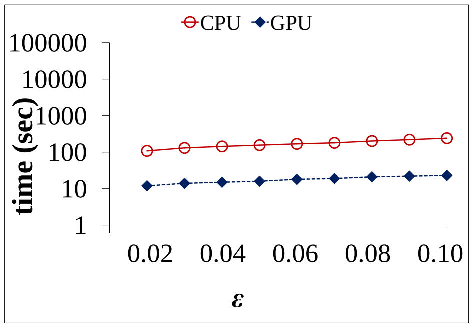


(a) Runtime. (b) Speedup.

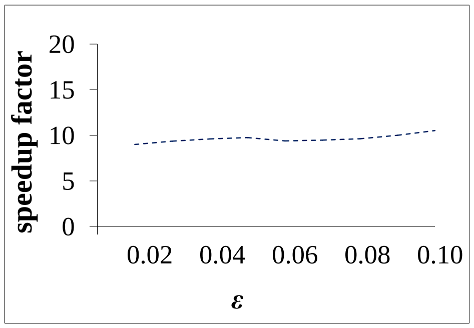
Figure 7.10: Evaluation of CUDA-DClust* w.r.t. $MinPts$.

Impact of the Parameter ϵ

We also evaluated the impact of the second DBSCAN parameter ϵ on CUDA-DClust* using a synthetic dataset consisting of 260k 8-dimensional points and fixed $MinPts = 4$. We test the impact of ϵ for values that range from 0.02 to 0.10 and present the results in Figure 7.11. Evidently, the impact of ϵ is negligible, as the range of the corresponding speedup factor is almost stable between nine and ten.



(a) Runtime.



(b) Speedup.

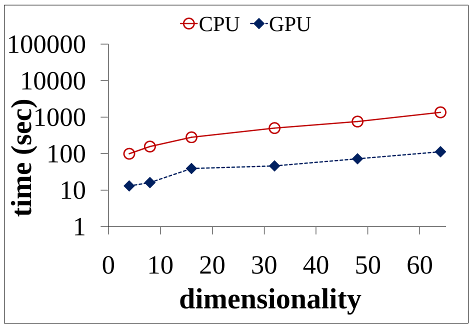
Figure 7.11: Evaluation of CUDA-DClust* w.r.t. ϵ.

Evaluation of the Dimensionality

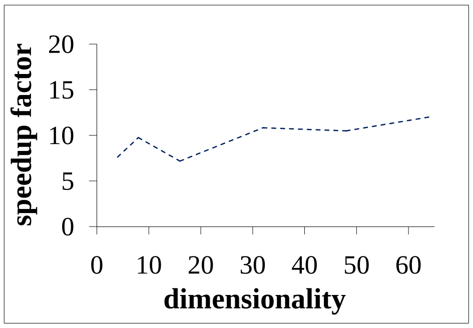
Here, we provide an evaluation w.r.t. the dimensionality of the data, ranging from 4 to 64. Again, we use our algorithm CUDA-DClust* and perform all tests on the synthetic dataset consisting of 260k 8-dimensional points clustered with default parameter-settings for ϵ and $MinPts$. Figure 7.12 illustrates that CUDA-DClust* outperforms the benchmark by factors of about seven to twelve depending on the dimensionality of the data. In our DBSCAN implementation the dimensionality is already known at compile time. Hence, optimization techniques of the compiler have an impact on the performance of the CPU version. Obviously, also the size of the data structures for the seed points in the GPU implementation is affected by the dimensionality and hence influences the performance of CUDA-DClust*, as the data structures are stored in the shared memory. This overhead affects the number of threads that can be executed in parallel on the GPU.

Evaluation of the Index Construction

All experiments do not include the time that is needed for constructing the index structure (described in Section 7.2) so far. Table 7.2 displays the time needed to construct the index, the runtime of CUDA-DClust and relates these two terms to



(a) Runtime.



(b) Speedup.

Figure 7.12: Evaluation of CUDA-DClust* w.r.t. the dimensionality.

each other for various sizes and dimensions of synthetic datasets. All experiments indicate that the time for building the index structure is negligible and independent of the parameter-setting by a fraction of lower than 5%. Further analysis demonstrate that larger datasets show an even smaller value of lower than 2%.

Table 7.2: Evaluation of the Index Construction for CUDA-DClust*.

n	DIM	INDEX	GPU	INDEX / GPU
65k	8	0.1 s	2.2 s	4.5 %
130k	8	0.2 s	5.8 s	3.4 %
260k	8	0.4 s	16.5 s	2.4 %
260k	16	0.6 s	38.6 s	1.6 %
260k	32	1.0 s	45.9 s	2.2 %
260k	64	1.8 s	111.6 s	1.6 %
520k	8	0.8 s	52.1 s	1.5 %
1m	8	1.7 s	182.4 s	0.9 %
2m	8	3.6 s	681.8 s	0.5 %

Runtime Profiling

In this section, we present a detailed runtime profile of CUDA-DClust* concerning different parts of the algorithm that is evaluated on two different synthetic datasets. The corresponding charts are depicted in Figure 7.13. The left diagram illustrates the distribution of the execution time of the program parts cluster expansion kernel, new seeds kernel, collision matrix and seed refill kernel on a dataset consisting of 520k 8-dimensional points and the right diagram for 4m points, respectively. Thus the bulk of performance is executed by the cluster expansion kernel. The auxiliary program parts and computations on CPU only consume a small fraction of the complete runtime as desired. Remark that the CPU is not reserved during the execution of the cluster expansion kernel, that requires the main part of the computation. Hence, the CPU can process multiple other jobs while the algorithm is executed by the cluster expansion kernel. In all experiments the time needed to divide the clustering process and the time to integrate the result for both algorithms CUDA-DClust and CUDA-DClust* is negligible.

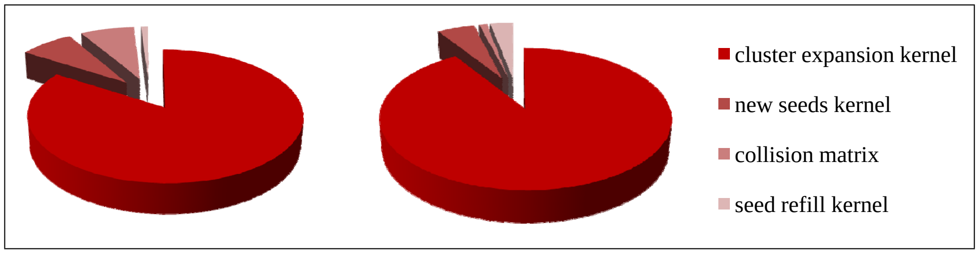


Figure 7.13: Runtime profile of CUDA-DClust* for two different datasets.

7.4.4 Conclusions

We proposed CUDA-DClust, a novel algorithm for very efficient density-based clustering supported by the computing power of the GPU. This architecture allows for extreme parallelization at very low cost. CUDA-DClust combines several concepts to exploit the special characteristics of the GPU for clustering, as parallel cluster expansion supported by the concept of chains, parallel nearest neighbor search that can even be accelerated by the use of a hierarchical index structure, and effective load balancing among the microprocessors. Our experiments demonstrate that CUDA-DClust outperforms the algorithm DBSCAN on CPU by an order of magnitude and yields an equally accurate clustering. The impressive results can even be increased by the use of an appropriate index structure, which yields to our second proposed algorithm CUDA-DClust*.

7.5 Partitioning Clustering on GPU

Finally, we describe the idea of performing a second paradigm of clustering, namely partitioning k-means clustering, on GPU.

7.5.1 The Sequential Procedure of k-means Clustering

A well-established partitioning clustering method is the k-means clustering algorithm [72]. k-means requires a metric distance function in vector space. In addition, the user has to specify the number of desired clusters k as an input parameter. Usually k-means starts with an arbitrary partitioning of the objects into k clusters. After this initialization, the algorithm iteratively performs the following two steps until convergence: (1) Update centers: For each cluster, compute the mean vector of its assigned objects. (2). Re-assign objects: Assign each object to its closest center. The algorithm converges as soon as no object changes its cluster assignment during two subsequent iterations.

Figure 7.14 illustrates an example run of k-means for $k = 3$ clusters. Figure 7.14(a) shows the situation after random initialization. In the next step, every data point is associated with the closest cluster center (cf. Figure 7.14(b)). The resulting partitions represent the Voronoi cells generated by the centers. In the following step of the algorithm, the center of each of the k clusters is updated, as shown in Figure 7.14(d). Finally, assignment and update steps are repeated until convergence.

In most cases, fast convergence can be observed. The optimization function of k-means is well defined. The algorithm minimizes the sum of squared distances of the objects to their cluster centers. However, k-means is only guaranteed to converge towards a local minimum of the objective function. The quality of the result strongly depends on the initialization. Finding that clustering with k clusters minimizing the objective function actually is a NP-hard problem, for details see e.g. [75]. In practice, it is therefore recommended to run the algorithm several times with different random initializations and keep the best result. For large datasets, however, often only a very limited number of trials is feasible. Parallelizing k-means in GPU allows for a more comprehensive exploration of the search space of all potential clusterings and thus provides the potential to obtain a good and reliable clustering even for very large datasets.

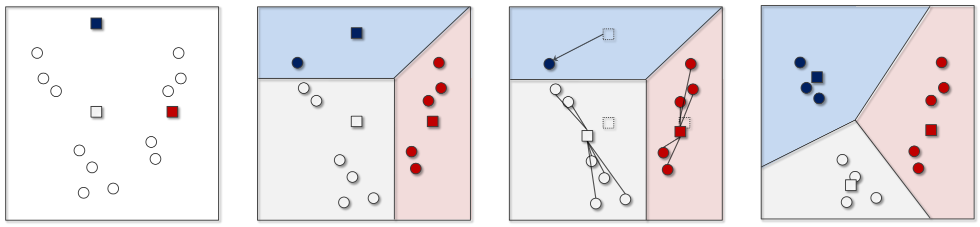

(a) Initialization. (b) Assignment. (c) Recalculation. (d) Termination.

Figure 7.14: Sequential partitioning clustering by the k-means algorithm.

7.5.2 CUDA-k-Means

In k-means, most computing power is spent in step (2) of the algorithm, i.e. reassignment which involves distance computation and comparison. The number of distance computations and comparisons in k-means is $O(k \cdot i \cdot n)$, where i denotes the number of iterations and n is the number of data points.

The CUDA-k-MeansKernel

In k-means, the cluster assignment of each data point is determined by comparing the distances between the point and each cluster center. This work is performed in parallel by the CUDA-k-meansKernel. The idea is, instead of (sequentially) performing cluster assignment of one single data point, many different cluster assignments are started simultaneously. In detail, one single thread per data point is generated, all executing the CUDA-k-meansKernel. Every thread which is generated from the CUDA-k-meansKernel (cf. in Algorithm 6) starts with the ID of a data point x that is going to be processed. Its main tasks are, to determine the distance to the next center and the ID of the corresponding cluster.

A thread starts by reading the coordinates of the data point x into the register. The distance of x to its closest center is initialized by ∞ and the assigned cluster is therefore set to `null`. Then a loop encounters all $c_1, c_2, \ldots, c_k$ centers and considers them as potential clusters for x. This is done by all threads in the same thread group allowing a maximum degree of intra-group parallelism. Finally, the cluster whose center has the minimum distance to the data point x is reported together with the corresponding distance value using synchronized writing.

The Main Program for CPU

Apart from initialization and data transfer from main memory to device memory, the main program consists of a loop starting the CUDA-k-meansKernel on GPU until the clustering converges. After the parallel operations are completed by all threads of the group, the following steps are executed in each cycle of the loop:

1. Copy distance of processed point x to the nearest center from device into main memory.
2. Copy cluster, x is assigned to, from device memory into main memory.
3. Update centers.
4. Copy updated centers to device memory.

Algorithm 6 Parallel algorithm for k-means on GPU.

```
algorithm CudaKMeans(dataset $DS$, int $k$) // host program executed on CPU
deviceMem float $DS'[][] := DS[][]$;            // allocate memory in DM for $DS$
threads := n;                                   // number of points in $DS$
tpg := 64;                                      // threads per group
deviceMem float $Centroids[][]$ := initCentroids(); // allocate memory in DM
                                                    // for the initial centroids
double $actCosts := \infty$;                        // initial costs of clustering


repeat
  $prevCost := actCost$;
  startThreads(CudaKMeansKernel, threads, tpg); // one thread per point
  waitForThreadsToFinish();
  float $minDist := minDistances[threadID]$; // copy distance to nearest
                                             // centroid from DM into MM
  float $cluster := clusters[threadID]$;     // copy assigned cluster
                                             // from DM into MM
  double $actCosts :=$ calculateCosts(); // update costs of clustering
  deviceMem float $Centroids[][] :=$ calculateCentroids();
                                         // copy updated centroids to DM
until $|actCost - prevCost| < threshold$

kernel CudaKMeansKernel(int $threadID$)
register float $x[] := DS'[threadID][]$;   // copy x from DM into register
float $minDist := \infty$;                  // distance of $x$ to the next centroid
int $cluster :=$ null;                      // ID of the next centroid (cluster)
for $i := 1$ to $k$ do
  register float $c[] := Centroids[i][]$   // copy actual centroid
                                           // from DM into register
  double $dist :=$ distance($x$,$c$);
  if $dist < minDist$ then
    $minDist := dist$;
    $cluster := i$;
  end if
end for
report($minDist$, $cluster$);           // report assigned cluster and distance
                                        // using synchronized writing
```

7.5.3 Experimental Evaluation

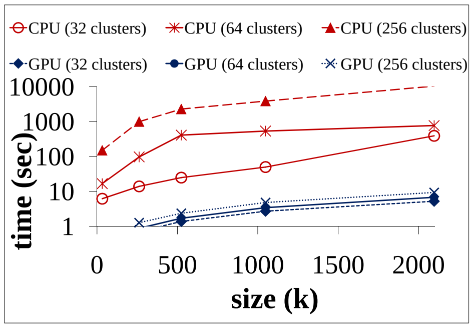
To analyze the efficiency of k-means clustering on GPU, we present experiments w.r.t. different dataset sizes, number of clusters and dimensionality of the data. As benchmark we apply a single-threaded implementation of k-means on CPU to determine the speedup of the implementation of k-means on GPU. As the number of iterations may vary in each run of the experiments, all results are normalized by a number of 50 iterations both on GPU and CPU implementation of k-means. All experiments are performed on synthetic datasets as described in detail in each of the following settings.

Evaluation of the Dataset Sizes

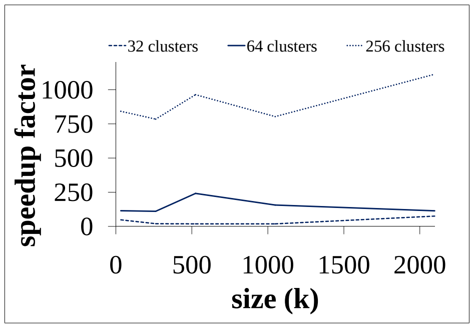
For these experiments we created 8-dimensional synthetic datasets of different sizes, ranging from 32k to 2m data points. The datasets consist of different numbers of random clusters, generated like DS_1, specified in Table 7.1. Figure 7.15(a) displays the runtime in seconds in logarithmic scale and the corresponding speedup factors of CUDA-k-means and the benchmark method on CPU for different number of data points. The time needed for data transfer from CPU to GPU and back has been included. The corresponding speedup factors are given in Figure 7.15(b). Once again, these experiments support the evidence that the performance of data mining approaches on GPU outperform classic CPU versions by significant factors. Whereas a speedup of approximately 10 to 100 can be achieved for relatively small number of clusters (data objects), we obtain a speedup of about 1000 for 256 clusters, that is even increasing with the number of data objects.

Evaluation of the Number of Clusters

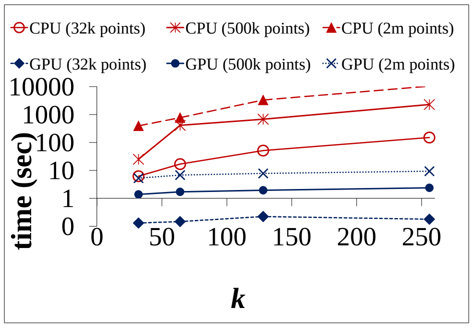
We performed several experiments to validate CUDA-k-means w.r.t. the number of clusters k. Figure 7.15(c) shows the runtime in seconds of CUDA-k-means compared with the implementation of k-means on CPU on 8-dimensional syn-



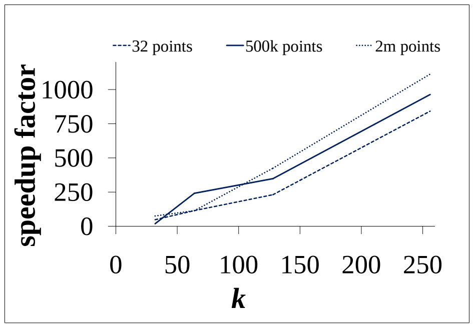
(a) Runtime for different dataset sizes.



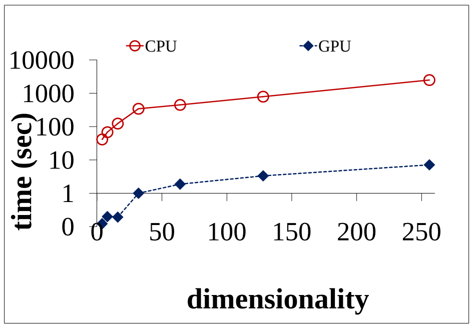
(b) Speedup for different dataset sizes.



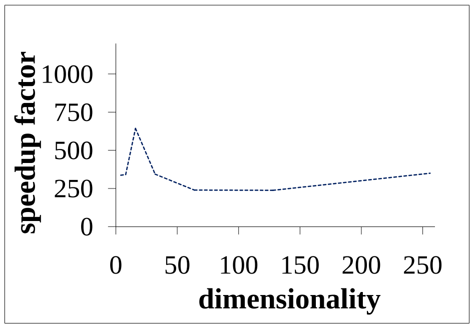
(c) Runtime for different number of clusters.



(d) Speedup for different number of clusters.



(e) Runtime for different dimensionalities.



(f) Speedup for different dimensionalities.

Figure 7.15: Evaluation of k-means clustering on CPU and GPU w.r.t. the size of different datasets, the number of clusters, and different dimensionalities.

thetic datasets that contain different number of clusters, ranging from 32 to 256, again together with the corresponding speedup factors in Figure 7.15(d).

The experimental evaluation of k on a dataset that consists of 32k points results in a maximum performance benefit of more than 800 compared to the benchmark implementation. For 2m points the speedup ranges from nearly 100 up to even more than 1,000 for a dataset that comprises 256 clusters. In this case the calculation on the GPU takes approximately five seconds, compared to almost three hours on CPU. Therefore, we determine that due to massive parallelization, CUDA-k-means outperforms the CPU implementation by large factors, that are even growing with k and the number of data objects n.

Evaluation of the Dimensionality

These experiments provide an evaluation w.r.t. the dimensionality of the data. We perform all tests on synthetic data consisting of 16k data objects. The dimensionality of the test datasets vary in a range from 4 to 256. Figure 7.15(f) illustrates that CUDA-k-means outperforms the benchmark method k-means on CPU by factors of 230 for 128-dimensional data to almost 500 for 8-dimensional data. On GPU and CPU, the dimensionality affects possible compiler optimization techniques, like loop unrolling as already shown in the experiments for the evaluation of the similarity join on the GPU (cf. Section 7.3.3).

7.5.4 Conclusions

We demonstrated how Graphics processing Units (GPU) can effectively support highly complex data mining tasks. In particular, we focused on the iterative algorithm k-means and proposed algorithms illustrating how to effectively support clustering on GPU. Our proposed algorithm is accustomed to the special environment of the GPU which is most importantly characterized by extreme parallelism at low cost. Our experimental evaluation emphasizes the potential of the GPU for high-performance data mining.

Chapter 8

Conclusions

The rapidly increasing amount of data stored in databases requires effective and efficient data mining methods to gain new information contained in the collected data. Clustering is one of the primary data mining tasks and aims at detecting subgroups of similar data objects. This thesis contributes to the fields of mining structured datatypes and proposes opportunities for boosting the data mining process by the intelligent adoption of Graphics Processing Units (GPUs). In Section 8.1, a detailed summary of our contributions is given. This thesis is concluded by some potential ideas for future work in Section 8.2.

8.1 Summary

Many data contain some kind of hierarchical relationship. That means that data mining approaches can reveal the information present in this data w.r.t. different levels. Hierarchical clustering has therefore become very popular in various scientific domains, such as molecular biology, medicine or economy. In this thesis, we contributed two novel hierarchical clustering algorithms – ITCH (Information-theoretic Cluster Hierarchies) and GACH (Genetic Algorithm for Finding Cluster Hierarchies). ITCH is a hierarchical information-theoretic clustering method that has the following advantages: It uses a Gaussian probability density function

(PDF) as representative for each cluster. This is returned as the result of the clustering method but also consequently used as an optimization criterion during the algorithm. Furthermore, each node in the hierarchy represents a meaningful cluster. This is achieved by an adaptation of the Minimum Description Length (MDL) principle for the hierarchical case. Outliers are flexibly handled by assigning them to higher levels of the cluster hierarchy. Outliers which are completely dissimilar to all objects in the database may be assigned to the root of the hierarchy; other points which are outliers to a lesser degree and just do not fit to any very specific cluster at a very detailed view may be assigned to intermediate layers of the hierarchy. Finally, ITCH is effective, efficient and fully automatic, i.e. difficult and critical parameter settings are avoided. The second hierarchical clustering algorithm, presented in this thesis, is called GACH. GACH is a genetic-based algorithm which combines the benefits of genetic algorithms, information theory and model-based clustering being an efficient and accurate hierarchical clustering technique. It uses a MDL-based fitness function. Hence, it can be easily applied to real world applications without requiring any background knowledge by the user about the data, like e.g. the real number of clusters. Due to the fact that GACH integrates an EM-like strategy, the content of all clusters is described by an intuitive description in form of a PDF. As ITCH, GACH assigns outliers to appropriate inner nodes of the hierarchy, depending on their degree of outlierness. Our experimental evaluation demonstrate that GACH outperforms a multitude of other clustering approaches in terms of accuracy. In fact, we have obtained better results concerning different quality measures.

Besides hierarchical data mining, the analysis of data that are composed of attributes concerning different domains has turned out to be among the top 10 challenging problems in data mining. This thesis contributes a novel integrative clustering algorithm that addresses the question how to find a good partitioning of data that have both numerical and categorical attributes. Most of the existing clustering approaches require prior knowledge on the data that specifies the number of clusters or defines the weighting of the numerical and categorical information.

Here, we gave a solution to avoid difficult parameter settings in integrative clustering which was guided by the information-theoretic idea of data compression. For this purpose, we defined $iMDL$, an alternative for the classic MDL principle. The data partitioning can be used for efficient coding, and the achievable compression rate thus defines an objective function for the integrative clustering problem. On top of the idea of data compression we have proposed INTEGRATE, an information-theoretic integrative clustering method which is a k-means based approach for mixed type data that fulfills the following requirements. First, based on the MDL principle, INTEGRATE allows for a natural weighting of numerical and categorical information. Second, there is no need for difficult parameter settings which makes INTEGRATE fully automatic. Third, being the result of an optimization process maximizing the achievable data compression, INTEGRATE uses the numerical and categorical information most effectively. And finally, it shows high efficiency and is therefore scalable to large datasets.

Performing data mining on complex data that can not be easily described by feature vectors was another challenging problem during this work. However, inspired by the success of the skyline operator for multi-criteria decision making, we demonstrated that the skyline of a dataset is a very useful representation capturing the most interesting characteristics. And therefore, data mining on skylines is very promising for knowledge discovery. With SkyDist, a similarity measure for comparing different skylines, we proposed the first approach to data mining on skyline objects. We presented a baseline Monte-Carlo Sampling approach, called MCSkyDist that approximates the distance and an exact sweep-line based computation, named SLSkyDist. We showed, that SkyDist can easily be integrated into many data mining techniques. Real world case studies on clustering skylines that represent interesting offers of an automotive market or that describe the performance profile of basketball players, demonstrated that data mining on skylines, enabled by SkyDist, yields interesting novel knowledge.

Finally, we presented different contributions to the field of graph mining. Among the aforementioned data structures, graphs or networks are the most general ones, as almost every relationship can be modeled by graphs. In particular, we worked on a large real case study on brain networks of patients with somatoform pain disorder. The brain network models were constructed out of fMRI scan time series (a special form of neuroimaging) of six somatoform patients and four healthy controls. Each edge inside the network represents a significant degree of co-activation between different brain regions. On these network models, we have applied the heuristic frequent subgraph algorithm GREW and FANMOD, an exhaustive sampling method for frequent subgraph discovery. The resulting patterns represent groups of brain compartments that covary in their activity during the process of pain stimulation. We evaluated the appearance of frequent subgraphs for the group of patients and the healthy controls, respectively. Our results let us suspect that somatoform pain disorder is caused by an additional pathogeneous activity, not by a missing physiological activity. So far we have cared about the topology of the network, but ignore the temporal order of these interactions. To take this aspect into account, we defined an approach for dynamic graph mining and applied it on biological yeast networks. To achieve real dynamic networks, we co-integrated gene expression and protein interaction data from the organism *S.cerevisiae*. The information about the presence of an edge at a particular time step is modeled by so-called existence strings. We have defined an efficient algorithm for frequent dynamic subgraph discovery in an on-top fashion, meaning that we search a set of frequent static subgraphs for dynamic patterns. To provide the enumeration of static patterns, existing subgraph mining algorithms can be easily integrated into our framework. The dynamic patterns are efficiently computed using the idea of suffix trees. The resulting patterns represent groups of proteins that occur frequently within a protein interaction network and in which pairs of proteins are co-regulated in the same temporal order.

In the second part of this thesis, we contributed to the efficiency aspect of data mining algorithms. Many computations are extremely time consuming, especially in the presence of large databases. We presented solutions by the help of graphics processors that can effectively support highly complex data mining tasks. In particular, we demonstrated how to boost the clustering process and presented ideas for accelerating the similarity search. We selected two well-known clustering algorithms, the density-based algorithm DBSCAN and the iterative partitioning algorithm k-means and proposed approaches illustrating how to effectively support clustering on GPU. Our proposed algorithms are accustomed to the special environment of the GPU which is most importantly characterized by extreme parallelism at low cost. In addition, we proposed a parallel version of the similarity join and an index structure for efficient similarity search. Going beyond the primary scope of this idea, these building blocks are applicable to support a wide range of data mining tasks, including outlier detection, association rule mining and classification. Our extensive experimental evaluation emphasizes the potential of the GPU for high-performance data mining.

8.2 Outlook

At the end of this thesis, the potentials of the proposed methods for future research are emphasized.

- While much work has been done in identifying cluster structures that are described by Gaussian Mixture Models, the field of detecting structures that form relationships of PDFs of arbitrary type is almost unexplored. Extending the coding scheme for Gaussian PDFs to mixture models of the Exponential Power Distribution type would therefore be a next step.

- Our algorithms ITCH and GACH for information-theoretic mining of cluster hierarchies are based on a hierarchical extension of the MDL principle. For simplicity the coding scheme is restricted to the diagonal covariance matrix. In order to apply our algorithms on arbitrary oriented clusters, we plan to integrate the coding of the complete covariance matrix w.r.t. hierarchical cluster relationships.

- For GACH, we used the heuristic approach of genetic algorithms. Another promising probabilistic heuristic concept would be simulated annealing. In each step of the simulated annealing process, the current solution for the hierarchical clustering problem is replaced by a random solution. The new solution may then be accepted with a probability that depends both on the difference between the corresponding $hMDL$ value and also on a global parameter T (called the temperature), that is gradually decreased during the process.

- Another interesting idea would be to amplify the field of integrative data mining. While our proposed algorithm INTEGRATE is designed to handle data that consists of numerical and categorical attributes, one might also be interested in incorporating data types like e.g. graph-structures. Using a variety of different information for the data mining process yields to a maximum of knowledge extraction. For example, a clustering method that combines categorical and graph-structured data can be used highly effective for mining social network data.

- For high efficient data mining, future research could be guided in the following directions: Besides similarity search and clustering there is a high number of data mining processes that are often hard to apply on large real-world data. Therefore, it would be beneficial to develop further algorithms to support more specialized data mining tasks on GPU, including for example subspace and correlation clustering and medical image processing.

List of Figures

List of Tables

Bibliography

[1] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[2] J. Aach, W. Rindone, and G. M. Church. Systematic Management and Analysis of Yeast Gene Expression Data. *Genome Res.*, 10:431–445, Feb 2000.

[3] A. Ahmad and L. Dey. A K-mean Clustering Algorithm for Mixed Numeric and Categorical Data. *Data Knowl. Eng.*, 63(2):503–527, 2007.

[4] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In *SIGMOD Conference*, pages 49–60, 1999.

[5] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene Ontology: Tool for the Unification of Biology. The Gene Ontology Consortium. *Nat Genet.*, 25:25–29, 2000.

[6] T. Bäck, editor. *Proceedings of the 7th International Conference on Genetic Algorithms, East Lansing, MI, USA, July 19-23, 1997*. Morgan Kaufmann, 1997.

[7] H. M. Berman, J. D. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Research*, 28(1):235–242, 2000.

[8] D. J. Bernstein, T.-R. Chen, C.-M. Cheng, T. Lange, and B.-Y. Yang. ECM on Graphics Cards. In *EUROCRYPT*, pages 483–501, 2009.

[9] C. Böhm, B. Braunmüller, M. M. Breunig, and H.-P. Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pages 298–305, 2000.

[10] C. Böhm, C. Faloutsos, J.-Y. Pan, and C. Plant. Robust Information-theoretic Clustering. In *KDD*, pages 65–75, 2006.

[11] C. Böhm, C. Faloutsos, and C. Plant. Outlier-robust Clustering using Independent Components. In *SIGMOD Conference*, pages 185–198, 2008.

[12] C. Böhm, F. Fiedler, A. Oswald, C. Plant, B. Wackersreuther, and P. Wackersreuther. ITCH: Information-Theoretic Cluster Hierarchies. In *ECML/PKDD (1)*, pages 151–167, 2010.

[13] C. Böhm, S. Goebl, A. Oswald, C. Plant, M. Plavinski, and B. Wackersreuther. Integrative Parameter-Free Clustering of Data with Mixed Type Attributes. In *PAKDD (1)*, pages 38–47, 2010.

[14] C. Böhm, R. Noll, C. Plant, and B. Wackersreuther. Density-based Clustering using Graphics Processors. In *CIKM*, pages 661–670, 2009.

[15] C. Böhm, R. Noll, C. Plant, B. Wackersreuther, and A. Zherdin. Data Mining Using Graphics Processing Units. *T. Large-Scale Data- and Knowledge-Centered Systems*, 1:63–90, 2009.

[16] C. Böhm, R. Noll, C. Plant, and A. Zherdin. Indexsupported Similarity Join on Graphics Processors. In *BTW*, pages 57–66, 2009.

[17] C. Böhm, A. Oswald, C. Plant, M. Plavinski, and B. Wackersreuther. SkyDist: Data Mining on Skyline Objects. In *PAKDD (1)*, pages 461–470, 2010.

[18] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther. Pattern Mining in Frequent Dynamic Subgraphs. In *ICDM*, pages 818–822, 2006.

[19] S. Börzsönyi, D. Kossmann, and K. Stocker. The Skyline Operator. In *ICDE*, pages 421–430, 2001.

[20] F. Cao, A. K. H. Tung, and A. Zhou. Scalable Clustering Using Graphics Processors. In *WAIM*, pages 372–384, 2006.

[21] B. C. Catanzaro, N. Sundaram, and K. Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *ICML*, pages 104–111, 2008.

[22] R. Cho, M. Campbell, E. Winzeler, L. Steinmetz, A. Conway, L. Wodicka, T. Wolfsberg, A. Gabrielian, D. Landsman, D. Lockhart, and R. Davis. A Genome-wide Transcriptional Analysis of the Mitotic Cell Cycle. *Mol. Cell*, 2:65–73, Jul 1998.

[23] D. J. Cook and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. *JAIR*, 1:231–255, 1994.

[24] E. H. Davidson, J. P. Rast, P. Oliveri, A. Ransick, C. Calestani, C.-H. Yuh, T. Minokawa, G. Amore, V. Hinman, C. Arenas-Mena, O. Otim, C. T. Brown, C. B. Livi, P. Y. Lee, R. Revilla, A. G. Rust, Z. Pan, M. J. Schilstra, P. J. C. Clarke, M. I. Arnone, L. Rowen, R. A. Cameron, D. R. McClay, L. Hood, and H. Bolouri. A Genomic Regulatory Network for Development. *Science*, 295(5560):1669–1678, 2002.

[25] D. Defays. An Efficient Algorithm for a Complete Link Method. *Comput. J.*, 20(4):364–366, 1977.

[26] A. Demiriz, K. P. Bennett, and M. J. Embrechts. Semi-supervised Clustering using Genetic Algorithms. *Artificial neural networks in engineering*, pages 809–814, 1999.

[27] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. In *J Roy Stat Soc*, number 39, pages 1–31, 1977.

[28] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[29] P. K. Desikan and J. Srivastava. Mining Temporally Changing Web Usage Graphs. In *WebKDD*, pages 1–17, 2004.

[30] B. Dom. An Information-theoretic External Cluster-Validity Measure. In *UAI*, pages 137–145, 2002.

[31] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, pages 226–231, 1996.

[32] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. Knowledge Discovery and Data Mining: Towards a Unifying Framework. In *KDD*, pages 82–88, 1996.

[33] Y. Feng and G. Hamerly. PG-means: Learning the Number of Clusters in Data. In *NIPS*, pages 393–400, 2006.

[34] J. R. Filho, C. Alippi, and P. Treleaven. Genetic Algorithm Programming Environments. *IEEE Computer*, 27:28–43, 1994.

[35] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[36] J. Gasteiger and T. Engel. *Chemoinformatics: A Textbook*. Wiley-VCH, 2003.

[37] E. Geuze, H. Westenberg, A. Jochims, C. de Kloet, M. Bohus, E. Vermetten, and C. Schmahl. Altered Pain Processing in Veterans with Posttraumatic Stress Disorder. *Arch. Gen. Psychiatry*, 64:76–85, Jan 2007.

[38] N. K. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management. In *SIGMOD Conference*, pages 325–336, 2006.

[39] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast Computation of Database Operations using Graphics Processors. In *SIGMOD Conference*, pages 215–226, 2004.

[40] P. Grünwald. A Tutorial Introduction to the Minimum Description Length Principle. *CoRR*, math.ST/0406077, 2004.

[41] H. Gündel, M. Valet, C. Sorg, D. Huber, C. Zimmer, T. Sprenger, and T. R. Tölle. Altered Cerebral Response to Noxious Heat Stimulation in Patients with Somatoform Pain Disorder. *Pain*, 137(2):413–421, July 2008.

[42] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explorations*, 11(1):10–18, 2009.

[43] G. Hamerly and C. Elkan. Learning the K in K-means. In *NIPS*, 2003.

[44] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.

[45] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[46] E. L. Hong, R. Balakrishnan, Q. Dong, K. R. Christie, J. Park, G. Binkley, M. C. Costanzo, S. S. Dwight, S. R. Engel, D. G. Fisk, J. E. Hirschman, B. C. Hitz, C. J. Krieger, M. S. Livstone, S. R. Miyasato, R. S. Nash, R. Oughtred, M. S. Skrzypek, S. Weng, E. D. Wong, K. K. Zhu, K. Dolinski, D. Botstein, and J. M. Cherry. Gene Ontology Annotations at SGD: New Data Sources and Annotation Methods. *Nucleic Acids Research*, 36(Database-Issue):577–581, 2008.

[47] Z. Huang. Clustering Large Data Sets with Mixed Numeric and Categorical Values. In *In The First Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 21–34, 1997.

[48] Z. Huang. Extensions to the k-Means Algorithm for Clustering Large Data Sets with Categorical Values. *Data Min. Knowl. Discov.*, 2(3):283–304, 1998.

[49] Z. Huang and M. K. Ng. A Note on K-modes Clustering. *J. Classification*, 20(2):257–261, 2003.

[50] L. C. K. Hui. Color Set Size Problem with Application to String Matching. In *CPM*, pages 230–243, 1992.

[51] D. H. Huson and D. Bryant. Application of Phylogenetic Networks in Evolutionary Studies. *Mol Biol Evol*, 23(2):254–267, 2006.

[52] A. Inokuchi, T. Washio, and H. Motoda. Complete Mining of Frequent Patterns from Graphs: Mining Graph Data. *Machine Learning*, 50(3):321–354, 2003.

[53] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[54] M. Kanehisa, S. Goto, S. Kawashima, Y. Okuno, and M. Hattori. The KEGG Resource for Deciphering the Genome. *Nucleic Acids Research*, 32(Database-Issue):277–280, 2004.

[55] N. Kashtan, S. Itzkovitz, R. Milo, and U. Alon. Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs.

[56] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.

[57] M. Kitsuregawa, L. Harada, and M. Takagi. Join Strategies on KD-Tree Indexed Relations. In *ICDE*, pages 85–93, 1989.

[58] D. Kossmann, F. Ramsak, and S. Rost. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *VLDB*, pages 275–286, 2002.

[59] H.-P. Kriegel, K. M. Borgwardt, P. Kröger, A. Pryakhin, M. Schubert, and A. Zimek. Future Trends in Data Mining. *Data Min. Knowl. Discov.*, 15(1):87–97, 2007.

[60] K. Krishna and M. N. Murty. Genetic K-means Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(3):433–439, 1999.

[61] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. In *ICDM*, pages 313–320, 2001.

[62] M. Kuramochi and G. Karypis. Finding Frequent Patterns in a Large Sparse Graph. In *SDM*, 2004.

[63] M. Kuramochi and G. Karypis. GREW-A Scalable Frequent Subgraph Discovery Algorithm. In *ICDM*, pages 439–442, 2004.

[64] M. Lahiri and T. Y. Berger-Wolf. Structure Prediction in Temporal Networks using Frequent Subgraphs. In *CIDM*, pages 35–42, 2007.

[65] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *KDD*, pages 177–187, 2005.

[66] S. T. Leutenegger, J. M. Edgington, and M. A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, 1997.

[67] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*, pages 1111–1120, 2008.

[68] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the Sky: Efficient Skyline Computation over Sliding Windows. In *ICDE*, pages 502–513, 2005.

[69] W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. Molecular Dynamics Simulations on Commodity GPUs with CUDA. In *HiPC*, pages 185–196, 2007.

[70] L. A. N. Lorena and J. C. Furtado. Constructive Genetic Algorithm for Clustering Problems. *Evolutionary Computation*, 9(3):309–328, 2001.

[71] N. Ma, J. Guan, and Y. Zhao. Bringing PageRank to the Citation Analysis. *Inf. Process. Manage.*, 44(2):800–810, 2008.

[72] J. B. Macqueen. Some Methods of Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[73] S. Manavski and G. Valle. CUDA Compatible GPU Cards as Efficient Hardware Accelerators for Smith-Waterman Sequence Alignment. *BMC Bioinformatics*, 9(S-2), 2008.

[74] U. Maulik and S. Bandyopadhyay. Genetic Algorithm-based Clustering Technique. *Pattern Recognition*, 33(9):1455–1465, 2000.

[75] M. Meila. The Uniqueness of a Good Optimum for K-means. In *ICML*, pages 625–632, 2006.

[76] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs (3rd ed.)*. Springer, 1996.

[77] X. V. Nguyen, J. Epps, and J. Bailey. Information Theoretic Measures for Clusterings Comparison: Is a Correction for Chance Necessary? In *ICML*, page 135, 2009.

[78] A. Oswald and B. Wackersreuther. Motif Discovery in Brain Networks of Patients with Somatoform Pain Disorder. In *DEXA Workshops*, pages 328–332, 2009.

[79] S. K. Pal, D. Bhandari, and M. K. Kundu. Genetic Algorithms for Optimal Image Enhancement. *Pattern Recogn. Lett.*, 15(3):261–271, 1994.

[80] S. Pantazi, Y. Kagolovsky, and J. R. Moehr. Cluster Analysis of Wisconsin Breast Cancer Dataset Using Self-Organizing Maps, 2002.

[81] D. Pelleg and A. W. Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *ICML*, pages 727–734, 2000.

[82] F. Pernkopf and D. Bouchaffra. Genetic-based EM Algorithm for Learning Gaussian Mixture Models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1344–1348, 2005.

[83] G. Piatetsky-Shapiro, R. Grossman, C. Djeraba, R. Feldman, L. Getoor, and M. J. Zaki. Is there a Grand Challenge or X-prize for Data Mining? In *KDD*, pages 954–956, 2006.

[84] M. Remm, C. E. V. Storm, and E. L. L. Sonnhammer. Automatic Clustering of Orthologs and In-paralogs from Pairwise Species Comparisons. *J. Mol. Biol.*, 314:1041–1052, Oct 2001.

[85] G. E. Sarty. *Computing Brain Activity Maps from fMRI Time-Series Images*. Cambridge University Press, 2007.

[86] P. Scheunders. A Genetic c-Means Clustering Algorithm Applied to Color Image Quantization. *Pattern Recognition*, 30(6):859–866, 1997.

[87] S. A. A. Shalom, M. Dash, and M. Tue. Efficient K-Means Clustering Using Accelerated Graphics Processors. In *DaWaK*, pages 166–175, 2008.

[88] S. A. A. Shalom, M. Dash, and M. Tue. An Approach for Fast Hierarchical Agglomerative Clustering Using Graphics Processors with CUDA. In *PAKDD (2)*, pages 35–42, 2010.

[89] S. S. Shen-Orr, R. Milo, S. Mangan, and U. Alon. Network Motifs in the Transcriptional Regulation Network of Escherichia Coli. *Nat Genet.*, 31(1):64–68, May 2002.

[90] T. P. Speed. Review of 'Stochastic Complexity in Statistical Inquiry' (Rissanen, J.; 1989). *IEEE Transactions on Information Theory*, 37(6):1739–, 1991.

[91] S. Still and W. Bialek. How Many Clusters? An Information-theoretic Perspective. *Neural Computation*, 16(12):2483–2506, 2004.

[92] A. Szalay and J. Gray. 2020 Computing: Science in an Exponential World. *Nature*, 440:413–414, 2006.

[93] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient Progressive Skyline Computation. In *VLDB*, pages 301–310, 2001.

[94] A. Tasora, D. Negrut, and M. Anitescu. Large-scale Parallel Multi-body Dynamics with Frictional Contact on the Graphical Processing Unit. *Proc. of Inst. Mech. Eng. Journal of Multi-body Dynamics*, 222(4):315–326.

[95] N. Tishby, F. C. Pereira, and W. Bialek. The Information Bottleneck Method. *CoRR*, physics/0004057, 2000.

[96] N. Tzourio-Mazoyer, B. Landeau, D. Papathanassiou, F. Crivello, O. Etard, N. Delcroix, B. Mazoyer, and M. Joliot. Automated Anatomical Labeling of Activations in SPM using a Macroscopic Anatomical Parcellation of

the MNI MRI Single-subject Brain. *NeuroImage*, 1(15):273–289, January 2002.

[97] N. Vasconcelos and A. Lippman. Learning Mixture Hierarchies. In *NIPS*, pages 606–612, 1998.

[98] E. M. Voorhees. Implementing Agglomerative Hierarchic Clustering Algorithms for Use in Document Retrieval. *Inf. Process. Manage.*, 22(6):465–476, 1986.

[99] B. Wackersreuther, A. Oswald, P. Wackersreuther, J. Shao, and K. M. Borgwardt. Graph Mining on Brain Co-Activation Networks. *Database Technology for Life Sciences and Medicine, World Scientific*, 2010.

[100] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm, and K. M. Borgwardt. Frequent Subgraph Discovery in Dynamic Networks. In *MLG*, 2010.

[101] S. Wasserman and K. Faust. *Social Network Analysis. Methods and Applications*. Cambridge University Press, 1994.

[102] S. Wernicke. Efficient Detection of Network Motifs. *IEEE/ACM Trans. Comput. Biology Bioinform.*, 3(4):347–359, 2006.

[103] S. Wessely, C. Nimnuan, and M. Sharpe. Functional Somatic Syndromes: One or Many? *Lancet*, 354:936–939, September 1999.

[104] L. D. Whitley, T. Starkweather, and C. Bogart. Genetic Algorithms and Neural Networks: Optimizing Connections and Connectivity. *Parallel Computing*, 14(3):347–361, 1990.

[105] S. Wuchty, Z. N. Oltvai, and A. L. Barabasi. Evolutionary Conservation of Motif Constituents in the Yeast Protein Interaction Network. *Nat Genet.*, 35(2):176–179, Oct 2003.

[106] I. Xenarios, D. W. Rice, L. Salwínski, M. K. Baron, E. M. Marcotte, and D. Eisenberg. DIP: the Database of Interacting Proteins. *Nucleic Acids Research*, 28(1):289–291, 2000.

[107] X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*, pages 721–724, 2002.

[108] Q. Yang and X. Wu. 10 Challenging Problems in Data Mining Research. *International Journal of Information Technology and Decision Making*, 5(4):597–604, 2006.

[109] X. Zhu, R. Jin, Y. Breitbart, and G. Agrawal. MMIS07, 08: Mining Multiple Information Sources Workshop Report. *SIGKDD Explorations*, 10(2):61–65, 2008.