TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Informatik
Lehrstuhl für Bioinformatik

Machine Learning of Timed Automata

Jana A. Schmidt

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. R. Westermann
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. B. Rost
	2. Univ.-Prof. Dr. St. Kramer Johannes Gutenberg Universität Mainz

Die Dissertation wurde am 27.05.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.11.2013 angenommen.

Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17.05.2013

Abstract

This dissertation investigates the applicability of automata in the domain of process mining. Process mining is a part of machine learning that aims to describe, discover and to predict dynamic systems. It can be used especially in the domain of biology and medicine where many processes are still not fully understood. The main problem here is that the processes are very complex and that many factors interact with each other. Usually, one tries to model these factors by variables that influence each other. Therefore, a kind of dependency structure of the variables within a specific model is optimized to best reflect the measured data. In contrast, this work aims at modeling the variables' values change without explicitly assuming the inter-dependencies. This aim is achieved by a new type of model, which is based on automata. Automata are finite state models that are normally used to express formal grammars, to model and to detect languages that rely on discrete events. However, we expand the power of automata, as a first step towards modeling dynamic multivariate processes. Therefore, one part was to find a method (PRTA), which automatically identifies states and transitions of the given process. This is necessary because usually, the definition of states, i.e., the current characteristics of the system, are not known or have not been described before. To this end, the states of an automaton are annotated with so-called profiles, which describe the current stage of the system. The main idea for the automatic identification of the profiles is to extract frequent patterns of the process variables' characteristics. Moreover, the time component of the process should also be captured, which is done by the implementation of clock guards that define in which time frame a change in the states may take place. A subsequent problem is the scalability of the approach for large data sets. Therefore, a method (SPRTA) that uses online maximum frequent pattern based clustering is presented. To include existing background knowledge, a new type of constraint was developed and also implemented in the proposed approach (CSPRTA). Such constraints are based on the characteristics of the process and basically define the properties of the final states. Lastly, an improvement towards a genuine online induction method is presented (OPRTA). It may also detect concept drift in the underlying system. All of these approaches are evaluated with respect to scalability, accuracy, and, if appropriate, predictability. We hope that this kind of model may find some applications in biological and medical process mining.

Zusammenfassung

Diese Dissertation beschäftigt sich mit der Frage, ob Automaten für das Lernen von dynamischen Prozessen anwendbar sind. Der Bereich des sogenannten 'Process Mining' als Teil des maschinellen Lernens versucht diese Prozesse zu beschreiben, zu entdecken und auch deren Verlauf vorherzusagen. Besonders in der Biologie und Medizin kann das Anwendung finden, da dort viele Prozesse noch nicht ausreichend verstanden werden. Die Hauptursache dafür liegt darin, dass diese Prozesse sehr komplex sind und viele Faktoren miteinander interagieren. Bis jetzt versuchte man hauptsächlich, diese Faktoren mittels voneinander abhängigen Variablen und einer Abhängigkeitsstruktur in bestimmten Modellen nachzubilden, so dass die gemessenen Daten möglichst optimal reproduziert werden können. Im Gegensatz dazu versucht diese Arbeit die Änderung der Variablenausprägungen zu modellieren, ohne direkt Abhängigkeiten zwischen den Variablen anzunehmen. Dazu wurde ein neues Modell basierend auf Automaten entwickelt. Automaten sind endliche Zustandsmaschinen, die hauptsächlich darin Anwendung finden, formale Grammatiken zu beschreiben und Sprachen zu modellieren bzw. zu entdecken, die auf diskreten Ereignissen beruhen. Trotzdem haben wir in dieser Arbeit, Automaten dahingehend erweitert, dynamische multivariate Prozesse abbilden zu können (PRTA). Dazu war es zuerst nötig, eine Methode zu definieren, die aus den gegebenen Daten automatisch Zustände des Modells induziert, da solche Zustände, d.h. die derzeitigen Eigenschaften des Systems, u.U. nicht bekannt sind oder noch nicht konsolidiert wurden. Dafür werden die Zustände im Automaten mit sogenannten Profilen annotiert, die den derzeitigen Zustand des Systems beschreiben. Die zugrunde liegende Idee bei der Identifikation der Zustände und ihrer Profile ist, häufige Muster in den Eigenschaften des Prozesses zu entdecken. Daneben soll auch die zeitliche Komponente in das Modell einbezogen werden, was durch 'clock guards' - einem Zeitintervall für mögliche Übergänge - erreicht wird. Als nächster Schritt wurde diese Methodik auch für größere Datensätze skalierbar gemacht (SPRTA), wozu ein neues Clusteringverfahren basierend of maximalen häufigen Itemsets vorgestellt wird. Weiterhin sollte es ermöglicht werden, Hintergrundwissen in das Modell einfließen zu lassen (CSPRTA), wofür eine neue Art von Bedingungen entwickelt und ins Modell eingeschlossen wurde. Diese Bedingungen beschreiben die finalen Eigenschaften der Zustände. Im letzten Teil der Arbeit, wurde der PRTA auf inkrementelle und prinzipiell unendliche Datenströme angepasst (OPRTA), so dass auch eine Veränderung des Konzeptes im unterliegenden Prozess entdeckt werden kann. All diese Ansätze wurden hinsichtlich der Skalierbarkeit, Genauigkeit und ggf. Vorhersagekraft untersucht, so dass wir hoffen, dass diese Modell in einigen biologischen oder medizinischen Fragestellungen Anwendung findet.

Acknowledgements

This dissertation would not have been possible without the guidance and help of a set of individuals who spent a lot of time to assist me when I needed help, to inspire me when there was a lack of focus, to discuss and to evaluate several ideas, to push me when I was lazy, to share my concerns when there was too much work and to believe in a happy ending. Therefore, first of all I want to thank my parents Dr. Ursula Schmidt and Dr. Wolf-Dieter Schmidt for their continuing support and strong beliefs in their daughter. They made a happy study time and a very challenging PhD period possible and always stood by my side reassuring myself.
Beside my parents, I also want to thank my remaining family and friends, who supported me by sharing their similar experiences in their lives and PhD-times and continuously pushed me forward.
However, I will not forget to thank all of the students, who did some very interesting projects with me during their Diploma, Master or Bachelor thesis, for long discussions, critical questions and the immense work they managed: Constanze Schmitt for sharing the first impressions in process mining and continuing our discussions as a later colleague, Elisabeth Braendle and Sonja Ansorge for the implementation of some crazy ideas, Huang Xiao, Christian Mertes and Goukun Zang for the gain of knowledge in totally different domains. Besides, a thank is also to be contributed to all of my colleagues and especially Marianne Mueller, who supervised my Diploma thesis and encouraged me to follow a scientific career.
For many years Andreas Hapfelmeier has supported, discussed, reviewed, and sometimes even rescued my way through the scientific and industrial jungle. Especially, in the last (midnight) hours before a deadline he always was a friend, whom no thank can appreciate.
Last but not least I want to thank my supervisor Prof. Dr. Stefan Kramer for the motivation and inspiration to dig into Data Mining and Machine Learning in the domain of process mining. I think that the knowledge that I gained during this time will help and guide me all my life, from which probably the most important thing is that he encouraged me to follow such a new field of research.

Contents

Chapter 1

Introduction

1.1 Motivation

Recent years have shown a surge of interest in the evaluation and analysis of temporal data in many areas of science and industry. In the field of medicine, for instance, the evaluation of temporal data can help to understand the stages [79] and the progression of diseases. Another example is given by the domain of molecular biology, where time labeled data may provide insights into cellular processes. This work was essentially motivated by these two real world problems. For understanding the progression of diseases in a population, the challenge is to determine how the health status of each individual of the population will evolve in the future, and more general, to find a model for the overall disease progression of the entire population. A model that describes and identifies the patterns of diseases may be valuable for general practitioners, because it enables them to adjust their therapy, control therapy guidelines and quantify side effects. Moreover, the inclusion of many measurements leads to a model that is not biased by selection effects as is common in traditional medical studies. One challenge of the model is that the prediction time point is unknown in the training phase. Consider a physician that treats a patient. He wants to receive a short term prediction of the health status of his patient to optimize the therapy or to consult a specialist. However, another physician that is involved in a study may be interested in a long term prediction of the disease progression, to decide whether or not the patient should be enrolled in a new drug therapy trial. Therefore, a model that incorporates the prediction for a continuous time frame is desired. The problem of disease progression is additionally complex, because a set of variables (diseases) has to be predicted depending on their current configuration. Thus, there is a multivariate dependency of the current variables, which may (partly) influence each other. The same is true for the second real world application: the gene expression profiles of cells. Here again, the genes' expression values are considered as variables

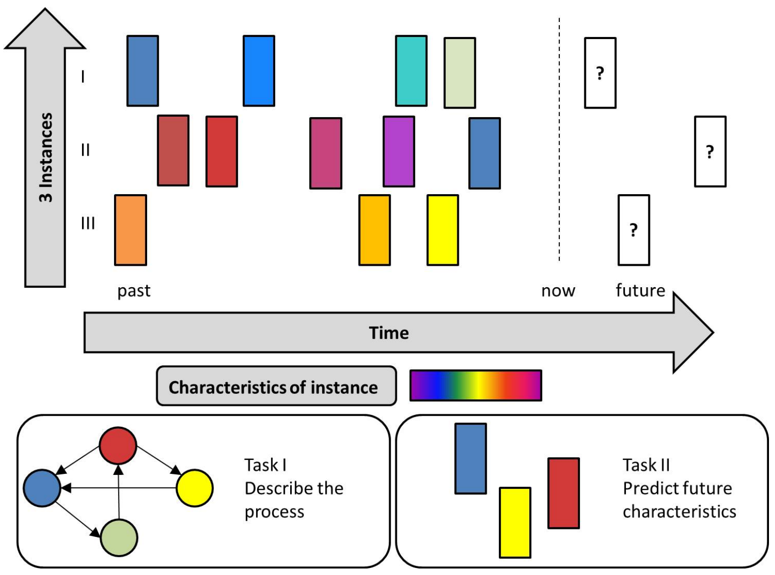


Figure 1.1: Illustration of the general problem setting

that influence each other. By learning from known gene expression patterns, it is possible to identify the influence of gene configurations on future gene expressions. Both problem settings require a model that shows the influence of variables (here genes and diseases) on each other, without assuming a specific dependency function apriori. Typically, the data provided for this purpose includes the description of stages (or states) as well as their temporal relation.

Figure 1.1 illustrates the problem setting: process modeling and prediction for the provided data set type. The input consists of a set of instances (here three rows of rectangles) having specific properties (indicated by the color). In the first real world problem, the instances reflect the individuals of a population, in the second setting, individual cells of a cell colony. Their properties are the diseases they have, or in the second setting, the genes that are currently expressed. As time passes, these characteristics can change. For example, there may be diseases that are cured and others that arise or deteriorate. Likewise, there are genes that are currently expressed and that also influence the expression level of other genes in the future. This is illustrated by the change in color of each instance from left to right. Note, that the instances are observed at different points of time. Moreover, the measurement of the instances is not necessarily at a fixed frequency. This leads to another model requirement: Since the data set contains measurements at individual time points for each instance, the model must be able to generalize from different observation time frames.

In the lower part of the figure, the two main problem settings are illustrated. First, a general, easy-to-understand model shall be elaborated, so that an

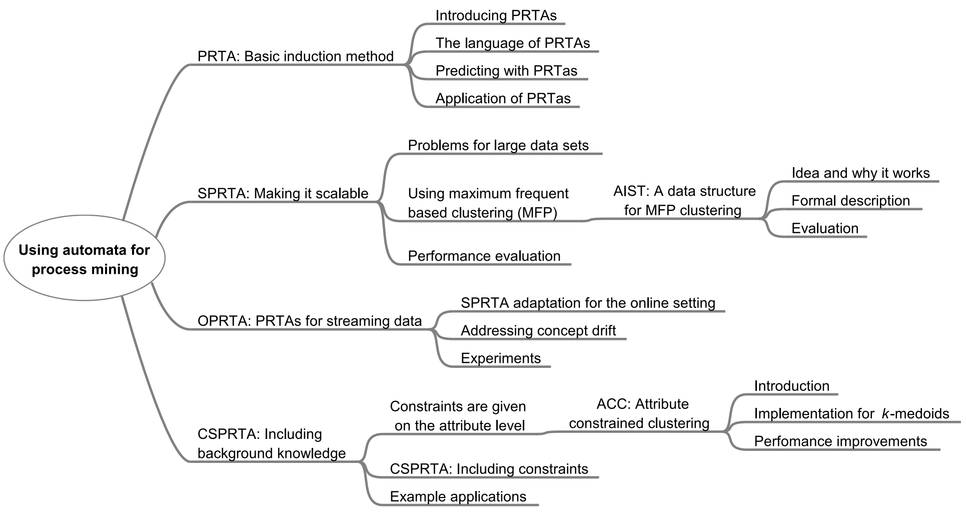

Using automata for process mining
PRTA: Basic induction method
Introducing PRTAs
The language of PRTAs
Predicting with PRTas
Application of PRTas
SPRTA: Making it scalable
Problems for large data sets
Using maximum frequent based clustering (MFP)
AIST: A data structure for MFP clustering
Idea and why it works
Formal description
Evaluation
Performance evaluation
OPRTA: PRTAs for streaming data
SPRTA adaptation for the online setting
Addressing concept drift
Experiments
CSPRTA: Including background knowledge
Constraints are given on the attribute level
ACC: Attribute constrained clustering
Introduction
Implementation for k-medoids
Perfomance improvements
CSPRTA: Including constraints
Example applications

Figure 1.2: Mindmap of this thesis

educated user can inspect the patterns that are found in the process that alters the instances. This is depicted in the lower left hand side of the figure. The other problem is to predict the characteristics of the instances in the future (lower right hand side). Therefore, the induced model should be used. One possible way of representing such complex temporal phenomena is by timed automata [4], which are finite state models explicitly modeling time. They can be linked to domain concepts and help to reason about real time processes. Until now, experts construct such models by hand, which can be time-consuming and error-prone. The situation is even more complicated if the states of the process are described by multiple attributes. In fact, in such a setting, even the definition of a state is unclear. To deal with the problem of an automatic extraction of meaningful, expressive and temporally ordered states, we propose a new algorithm based on finding real-time automata. Formally, the observations consist of a multi-dimensional attribute vector and a corresponding time point, denoting when the state was observed. The implicit modeling of time, e.g. by an untimed model like Hidden Markov Models (HMMs) would result in a combinatorial explosion of states. The same is true for modeling multiple state characteristics. This problem is solved by adding profiles to states that represent all their events and the states' characteristics. The annotation of states makes the problem feasible and additionally makes the resulting model easier to understand.

1.2 Organization of the work

This work addresses several aspects of automata learning. Figure 1.2 shows a high-level mind map of the organization of this thesis. There are four

main projects: (1) the description of the basic model: probabilistic real time automata (*PRTA*), (2) its adaptation to large, sparse data sets, (3) its extension to data streams with concept drift and (4) the incorporation of background knowledge.
Chapter 2 reviews related work and puts this dissertation in the context of automata. As the described problem setting is closely related to graphical causal models, this chapter gives a short overview of the main approaches of this field. This includes Bayesian networks, Markov Random Fields and Factor Graphs. Although, these approaches model the dependency of specific variables, a short explanation shows why they were not adopted in this work. Second, the domain of process mining that already addresses the inclusion of several instances in time will be discussed. Here, we focus on Hidden Markov Models and Petri Nets as they are the most commonly used models. Third, an extended overview of the domain of automata detection will be given. This also includes grammars and formal languages, but mainly focuses on the basis of this work: (timed) automata. Different induction algorithms are presented as well as applications. Finally, a summary reviews the necessary requirements for the algorithms that may tackle the problem setting and indicates how this may be tackled.
Chapter 3 summarizes all data sets that are used for the evaluation of the algorithms, which include synthetic and real world data sets. Mostly, the synthetic data sets are taken to address complexity and runtime issues, while the real world data sets (covering disease and gene expression data) show how the resulting model can be interpreted. This helps to point out the benefits and shortcomings of the approaches. Additionally, we present quality measures that enable the comparison of the different types of algorithms. This includes runtime, similarity to an original automaton and interpretability.
Chapter 4 introduces the basic model: probabilistic real time automata (*PRTA*). The problem setting and the type of automata is formally defined. In the following chapter a description of how the basic function, the word-acceptance problem, of an automaton is solved. Section 4.1.2 describes the induction algorithm of PRTAs and provides details about the used prefix tree acceptor (PTA), the underlying clustering, and the merge-function. The formal description of a PRTA concludes a presentation of how PRTAs can be used to predict future events. A subsequent experimental section evaluates the applicability of the proposed PRTA.
Chapter 5 makes the PRTA scalable for large sparse data sets using an online clustering algorithm. The approach is then referred to as scalable PRTA (*SPRTA*). An appropriate clustering method is discussed along with the necessary mathematical functions. The basic idea is to find clusters that share as many frequent patterns as possible. The patterns are found in the instances that belong to a cluster. If large patterns are shared between the instances, they are very similar and form distinguishable profiles of the

states. Experiments again show the applicability of this approach.
Chapter 6 describes the basic data structure that is needed for the SPRTA algorithm: the augmented itemset tree ($AIST$). It is used to efficiently mine maximal frequent itemsets in an online setting, i.e. instances are observed one by one and each instance is only observed once. The AIST data structure is evaluated under different points of view and its efficiency is analyzed for various types of data sets. The main conclusion is that it is most applicable for sparse data sets consisting of large frequent patterns.
Chapter 7 shows how background knowledge can be incorporated into the process of automata induction (constrained SPRTA *CSPRTA*). The idea behind that is that for, e.g. medical data sets there are already some expectations or even focus groups that a physician wants to explore. Such expectations can be easily described by using attribute constraints, i.e. constraints that define the properties of the result. For example, consider an automaton having a cluster with persons suffering from diabetes. Two different types of constraints *must-link* and *must-link-exclusive* are defined and implemented into the induction algorithm. Some experiments show which types of constraints can be defined and used for this domain.
The following Chapter 8 presents the basics for such a constrained induction of automata. Here, the two constraints *must-link* and *must-link-exclusive* are defined and evaluated in the k-medoids algorithm, which serves as a proof of concept. The constraints are then applied on synthetic and real-world data sets, which show how the performance of a clustering algorithm can be improved if some background knowledge is taken into account.
Chapter 9 presents the last extension of the PRTA, an approach that adapts it to a genuine online setting (online PRTA *OPRTA* method). To do so, the main idea of the SPRTA-setting is adjusted to avoid the construction of the PTA. Additionally, the method is adjusted to include concept drift. Concept drift means that the underlying data generating process changes over time. For the two examples, this may be the case when a new therapy is used that may cure a disease or its side effects, or if a new environmental factor stresses the cell so that other gene expression patterns are observed. In either case, this would change the underlying automaton structure so that the OPRTA-algorithm also has to adopt to such shifts.
Chapter 10 summarizes the presented work, discusses the introduced algorithms and experiments and gives an outlook on future work.
Finally, the contribution of this thesis can be summarized as follows:

- We present a new model to capture multivariate processes.
- We make this approach scalable and also provide an online induction method.
- Therefore, we introduce a new algorithm for online maximum itemset mining.

- Finally, we propose a new type of constraint to include background knowledge for the proposed methods.

For each approach, a separate chapter formally defines the problem setting, the induction method and gives the results of the experiments on synthetic and real world data sets.

Chapter 2

Related Work

Today, temporal data are becoming available in many domains, where only little is known about the processes generating the data. Example domains include the internet (click paths), medicine (disease progressions) and biology (life cycles of organisms, biochemical pathways). In this thesis, a process is assumed, described by states (or events) that are labeled with a time stamp and a multi-attribute vector holding M variables. The problem is to identify a model that fully represents the data and additionally forms a hypothesis about the underlying process. Additionally, it should be possible to infer predictions. A potential solution to the problem comes from the field of graphical models. They give a compact representation of the processes and are easy to understand and interpret by the user. Although there exist learning algorithms for graphical models, they are often still created by an expert (e.g. HMMs), which is unsuitable for an automatic and unbiased model creation.

The general task is to infer a probabilistic model, i.e. a model that provides a probability for the observation of a given instance, from data set S. The data set consists of several sequences: $S = s_1, \dots, s_n$, where a sequence consists of various symbols $s_i = \langle s_{i1}, \dots, s_{ik} \rangle$, $s_{ik} \in \Sigma$, and Σ is an alphabet. However, in the presented real world problems an element of Σ is not a single symbol but an attribute vector. Therefore, each of the presented approaches is also evaluated whether it is also capable of modeling a multivariate (time) sequence. A multivariate sequence s_i does not only consist of single symbols but of a sequence of feature vectors having m attributes $s_{ik} = (a_1, \dots, a_m)$. As a first step towards such complex time-series descriptions this work focusses on binary feature vectors: $a_l \in \{0, 1\}$. Moreover, the multivariate sequences that are considered in this work are labeled with time stamps, i.e. s_i is a sequence of symbol-time pairs $s_i = \langle (s_{i1}, t_1), \dots, (s_{in}, t_n) \rangle$.

One problem that inspired this work is based in medical data mining: A population of individuals is monitored over time. At individual-specific time-points, each person gives feedback about its health-status, which is a binary feature vector that indicates whether a specific disease is present. Usually, this monitoring is done at a physician who writes down all diseases that a person suffers from. So, if a specific disease is not coded, the person is not considered to have it. Such visits may occur at different time steps, depending on the person's characteristics, so that the time intervals between the 'measurements' may also vary. However, there will be a timepoint, where someone wants to know in advance how this population will progress. This could be a health care provider to plan capacities and utilization or health care sponsors for budget planning. Then, a forecast of the population's health status has to be done, using an adequate model. To show different approaches to this problem setting, graphical causal models and approaches from process mining with their benefits and shortcomings are presented. Subsequently, the approach of using automata is introduced and the current limitations are described in this section. Last, we give a short description of how the benefits of all presented models could be combined into an automaton.

2.1 Graphical causal models

Graphical models are used to model a complex system by describing the topology of its components. That means that the dependencies of the components are described, i.e. which component influences another. A next aim is to validate assumptions about the graph components and of course to induce algorithms that make use of the topology in an efficient way. Besides, such a model should be translatable into a different form and moreover, should be understandable by other persons. In general, graphical models are depicted by different types of entities: nodes and edges. Nodes represent random variables (that may also be hidden) and edges reflect the dependencies between them. Hidden variables are a special type of the nodes and are frequently used to model noise or unknown characteristics of the system. More formally, graphical causal models are (un)directed (acyclic) graphs on a set of random variables (RV). As these models should be as simple as possible, RVs are grouped if they express similar dependencies. This makes it much easier for the user to identify which variables are independent from another under specific circumstances.
To learn such a model from the training data one commonly expresses the conditional distributions or potentials as parameterized functions. By a smart choice of such a parameterized function, the computation of the model parameters is much easier although the resulting type of model then is restricted. Usually a graphical model is constructed with prior knowledge about the model parameters. Each parameter that is known or believed to contribute to the process behavior is included in the model. This is done by considering each parameter as a random variable and then modeling the

dependencies between these random variables. Each parameter can either be an observable variables $v = (v_1, \dots, v_T)$ or a hidden variable $(h^{(t)})$ with parameter θ, where T reflects the number of instances. Then, training data is used to find the best setting of parameters. The best setting is the one that maximizes the probability of the training data:

$$P(h, v) = P(\theta) \prod_{t=1}^{T} P(h^{(t)}, v^{(t)} | \theta)$$

This formalization can also be used in a time-related problem setting. Then the parameters can be considered as random variables at specific timepoints that change over time:

$$P(\theta^{(t)} | \theta^{(t-1)})$$

There, the current value of a RV depends on the values of another set of RVs. Using uniform priors often makes the computation of the remaining dependencies much easier by using only the likelihood. Conjugate priors also offer this advantage but moreover, allow for the inclusion of stronger prior knowledge. After having specified the parameters and variables of such a system, algorithms are needed that calculate the specific values of the parameters to find the best model. Often, correct algorithms are intractable in the computational costs and therefore, heuristics and approximations (like the expectation maximization algorithm) are applied. One application where such models are used is the detection of image segments [33] and gene network inference [19]. In the following, several different types of graphical networks will be presented. They all model the relationship between variables in a (slightly) different way. After a short review of these methods, each of them is examined whether it is possible to model time-series of multivariate event structures.

2.1.1 Causal networks

Causal networks (CNs) are a class of models that are based on directed acyclic graphs (DAGs) that model the dependency between variables. In the following, Bayesian networks, Markov Random Fields and Factor Graphs are described to illustrate how variables can be modeled. For all of these types of models, one essential task is structure learning. Structure learning is necessary, if the true underlying causalities, i.e. which variable affects another, is not known. Of course, such a model should be as small as possible but should include all conditional dependencies of the given variables. The IC-algorithm [68] achieves that by first identifying all pairs of independent variables (a, b) for which an undirected graph is created that has an edge between a and b. Then a third variable c is tested whether it is dependent on the previous ones and if so, is added to the graph with v-like structure. Several additional constraints have to hold, and many improvements have

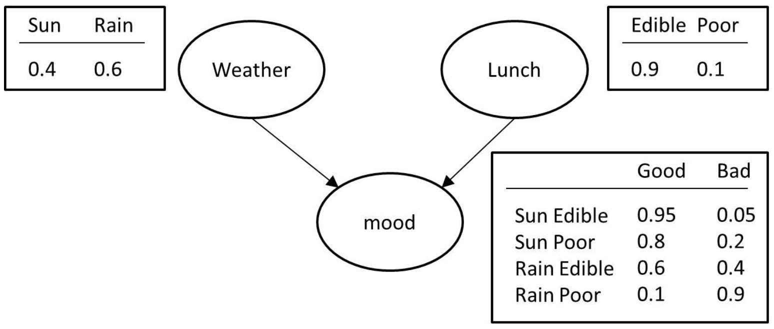


Figure 2.1: Illustration of a Bayesian network that models the dependency of someone's mood on the current weather and the lunch quality.

been proposed like the FD2CN algorithm [105]. It first extracts conditional independencies and then combines them via the chain and Markov boundary rule to infer a CN. This approach is able to infer relationships among variables and therefore the structure of the model. However, it is not applicable to multi-attribute events and noisy data, which is one of the key requirements to handle the addressed problem setting. Moreover, it is hard to reason about the underlying process that created the data set and also to evaluate the time courses of the events.

2.1.2 Bayesian Networks

Bayesian networks (BN) are a type of graphical causal models. They consist of nodes (which represent random variables) and transitions between the nodes and form a directed acyclic graph. The transitions represent the dependencies of the variables. Additionally, each node (or RV) is annotated with a conditional probability function from its parents $P(RV|P_a(RV))$, where P_a denotes the set of RV's parents. The distribution of these dependencies can in fact be arbitrary, but in general Gaussian distributions are used. Figure 2.1 shows an example of a Bayesian network (circles represent the variables and arcs their dependencies). The current mood of a person depends on the current weather and the lunch quality. Next to each node, a dependency table is given that shows the (conditional) probabilities for each variable value. Using these parameters it is possible to estimate the current mood of a person. Moreover, BNs can express the probability for a data set – its joint probability distribution which is the product of all probabilities that can occur in the model

$$P(BN) = \prod_{i=1}^{N} P(RV_i|P_a(RV_i)).$$

For easy problems (like the one given in Figure 2.1), Bayes' rule can be applied to fit the parameters (dependencies) between the data variables.

However, in more complex systems this may lead to an intractable number of variable combinations and thus to an enormeous runtime of the inference algorithm. To overcome such problems, one solution groups the variables accounting for their dependency relations and then exactly fits the parameters for this much smaller variable set. Second, an adequate splitting or even the elimation of variables can lead to more simple models. This can be applied if some variable can be ruled out by background knowledge to have an effect on the overall process. Third, approximations can be used to estimate the parameters. Most commonly, maximum likelihood estimation is used for incomplete probability models. In contrast, a maximum-a-posteriori estimation is used for complete BNs. To estimate the local maxima of the likelihood (or a-posteriori) functions, standard optimzation algorithms like gradient descent algorithms are used. In the case that there are missing values in the data set, the EM-algorithm (expectation-maximization) can also be applied to the induction of BNs. However, all these methods are based on a given underlying RV-dependency structure. If this structure is not known it has also to be inferred. Usually, this is then done in a greedy fashion. Elements (nodes and transitions) are added/deleted to the network, if a network specific score increases [34]. Then, this change is fixed in the network structure. Such a score can for instance be related to the minimum description length (MDL) principle, which leads to the smallest model producing the least errors. BNs have been applied exentensively to model causal relationships. One example is the domain of genetic inference [19], where the interaction of genes shall be explored. However, BNs still model single variables and have not yet been applied to modeling several discrete variables in parallel, i.e. the dependency of a set of variables is modeled in one node. Therefore, BNs remain unsuitable for the given problem setting.

2.1.3 Markov Random Fields

Markov Random Fields (MRF) also represent a set of random variables, but as a main difference from the previously introduced models, it is based on an *undirected* graph. Thus it may also be cyclic. There are three additional properties that make an undirected graph a MRF:

1. Any two non-adjacent variables are conditionally independent given all other variables.
2. A variable is conditionally independent of all other variables given its neighbours.
3. Any two subsets of variables are conditionally independent given a separating subset.

The basic idea behind MRF is to model a set of variables, where the variables are influenced by each ‘neighbours’. Moreover, there is an external ‘field’ that also influences the behaviour of the variables [51]. This model was first introduced by Ising, a german physicist, who wanted to model magnetism of different metals. However, this idea could be very nicely applied to other problems, especially it could be transformed into a temporal relationship, because it can be shown that the Gibbs measure and the Markov chain measure are the same[51] under specific conditions. To apply MRF for other domains, it was applied on a graph, including its parameterization. Therefore, MRF that can be factorized according to cliques of the graph are used, because these probability distributions are much easier to establish. Along with the random variables a *potential function* (or *clique potential*) for each maximal clique is given $g_k(RV_{C_k})$, where C_k are the elements of the kth maximal clique. Of course, such an MRF can also model the joint distribution. In this case, the joint distribution is expressed by the product of all potential functions divided by a normalization constant Z which gives the partition function

$$P(MRF) = \frac{1}{Z} \prod_{k=1}^{K} g_k(RV_{C_k}),$$

where

$$Z = \sum_{RV_1,\dots,RV_N} (\prod_{k=1}^{K} g_k(RV_{C_k})).$$

Note that one essential difference to Bayesian networks is that in BNs the product of conditional probabilities is automatically normalized ($Z = 1$). Using the partition function Z for many concepts from statistical mechanics, such as entropy is very straight forward in MRFs because it gives an intuitive understanding of the process. Moreover, if one wants to examine the effects of additional variables (like driving forces in statistical mechanics), i.e., how the system reacts on a pertubation, this is also done with the Z-function. To infer the conditional contribution the same estimation as for BNs can be done. Given one set of variables RV' for which the distribution has to be inferred one takes another set RV'' and sums over all possible assignments $u \notin RV', RV''$. One application example for MRFs is the domain of genome wide association studies [57]. The key idea is to include the effect of linkage disequilibrium for single nucleotide polymorphisms (SNPs). A graph is created that holds an edge for each pair of SNPs that are considered to be linked. If not, their occurence is considered independent. Then, a random binary variable is introduced for each SNP that shows whether a SNP is associated with a specific disease. Then, if SNP are in linkage disequilibrium their values are encouraged to have the same values. Li et al. [57] applied this model to a neuroblastoma data set containing 1032 cases and 2043 controls. From approximately 32.000 genes 5 could be identified to have a high correlation with neuroblastoma and linkage disequilibrium.

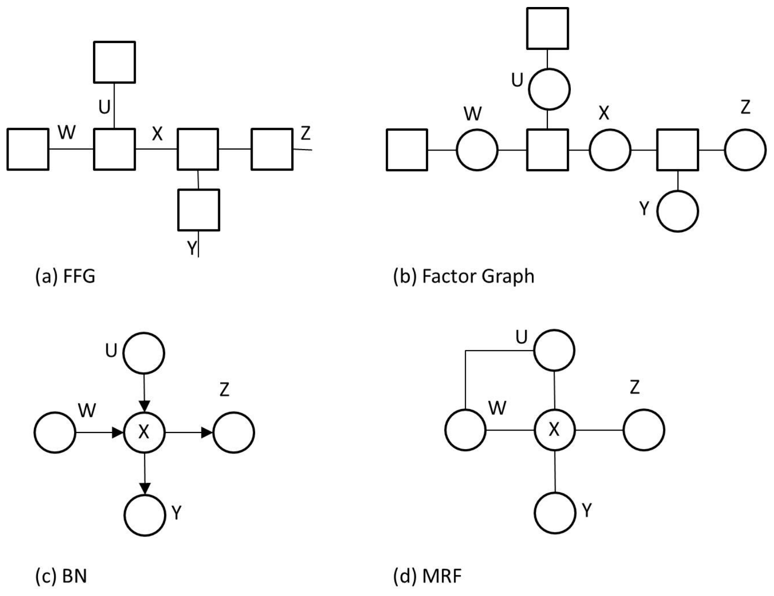


Figure 2.2: Illustration of the presented types of graphical models [60] for the factorization $p(u, w, x, y, z) = p(u)p(w)p(x|u, w)p(y|x)p(z|x)$.

2.1.4 Factor Graphs

Factor graphs (FG) subsume Bayesian Networks and Markov Random fields [33]. This also implies that BNs and MRFs can be translated into FGs. A FG is a bipartite graph for a set of RVs and a set of nodes that correspond to functions, where the product of all factors is the desired global function, while the factorized functions (nodes) correspond to local functions. However, there is also an alternative presentation of factor graphs by Forney (Forney-style factor graphs [*FFG*]) that use half-edges instead of node-functions. Figure 2.2 illustrates a FFG and moreover, shows how such a FFG can be expressed as a MRF or a BN. Essentially, a factor graph can be formed by applying the following rules [60]:

- There is a unique node for every factor.
- There is a unique (half-) edge for every variable.
- The node representing a factor g is connected with the edge (or half edge) representing a variable x iff g is a function of x.

The joint probability is equal to that of MRFs. The expression of functions by the nodes is very useful in the field of message passing algorithms (like belief propagation) and has many applications in the field of coding or signal processing like Kalman-filtering, which produces a model by stepwise measurements and parameter estimations.

To sum up graphical models, one important fact is that they are a very powerful tool to model variables and their dependencies. However, to the large number of variable combinations they may be inappropriate to model the influence of a set of variables on a set of other variables. Moreover, time constraints or a generalization of observations cannot be incorporated into such an approach.

2.2 Process Mining

Process mining tackles the identification of processes within a system with no external information. This also includes the identification of the elements of the process and their temporal relation. For example, it may be unknown how a process is actually conducted while another task may be to validate a predefined workflow. However, the main application is to find out the underlying mechanisms of a specific process. Examples for such a problem are given by electronic medical records (EMRs[1]) that arise in hospitals or the transaction logs of an enterprise resource planning system. Using such logs an analyst can identify the underlying sequences of frequently occurring tasks and derive optimizations or compare it to a predefined rule set to detect errors or abnormalities.
For the process induction of simple logs, of the shelve tools already exist[2] that include a variety of algorithms, like Petri nets and Hidden Markov Models (HMM). The tools can be applied to process logs consisting of simple, predefined events. They assume that it is possible to record events, i.e. they need a sequence of totally ordered events as input, which means that the events occur one after another. Such event sequences are collected in so-called event logs, which then serve as data basis for the process mining algorithms. Depending on the kind of process the user has to decide which type of model he expects. As one example, parallelism (when events can occur in parallel) can only be detected by Petri nets, while switch or redo-tasks can also be modeled by HMMS. In contrast, Petri nets usually do not offer probability distributions of the events/sequences that can occur or the frequency of path-choices. In the following, two approaches – HMMs and Petri nets – for the induction of processes or sequence patterns are presented.

2.2.1 Hidden Markov Models

Hidden Markov Models (HMMs) are a special case of graphical models. Algorithms for graphical models were extended to first order HMMs (e.g., for the MAP and inference problem [72]). First order HMMs consist of

[1] http://www.practicefusion.com/
[2] http://www.promtools.org/prom6/

hidden state variables (H) and observable variables (O), where the hidden variables define the underlying model. Hidden variables are connected by directed edges. Each hidden variable has exactly one predecessor and an associated observable variable, which is only dependent on the precedent variables (Markov property). An HMM can be defined by a triple $\lambda = (A, B, \pi)$. A is the matrix of state transition probabilities, B the observation symbol probability and π the initial state distribution. Such a model is very closely related to automata (cf. Section 2.3.2 as it can also provide estimations of the probability of sequences s_i (also denoted as words). Such a probability can be calculated using HMMs by Equation (2.1) when the set of possible state sequences that may produce a word $Q*$ is known.

$$P(s_i|\lambda) = \sum_{v \in Q*} P(s_i, v|\lambda) = \sum_{v \in Q*} P(s_i|v, \lambda)P(v|M) \tag{2.1}$$

The fitting of the parameters of an HMMs (A, B, π) is achieved by maximum likelihood estimation using the Forward-Backward algorithm [6] (also known as the Baum-Welch algorithm) that converges to a local maximum. Given the observation sequence S and the model parameters λ, $P(S|\lambda)$ is to be maximized. This can also be viewed as choosing the hypothesis or model $\hat{A}$ of all hypotheses $\mathcal{A}$ that maximizes $P(S|\lambda)$ [30].

$$\hat{\lambda}_{MAP} = \arg\max_{\hat{\lambda} \in \mathcal{L}} P(\hat{\lambda}|S) = \arg\max_{\hat{\lambda} in \mathcal{L}} P(S|\lambda)P(\hat{\lambda}). \tag{2.2}$$

Essentially this type of formulation arises from the Bayes framework that tries to optimize the tradeoff between the sample likelihood and the prior probability. If all models are considered as equally likely then this framework chooses the model with the maximal likelihood, which is depicted in Equation

$$\hat{\lambda}_{ML} = \arg\max_{\lambda} P(\widehat{S|\lambda}) = \arg\max_{\lambda} \prod_{i=1}^{m} P(s_i|\lambda)P(\hat{\lambda}). \tag{2.3}$$

Interestingly, maximum aposteriori learning is the same as learning under the minimum description length principle because Equation 2.2 can be reformulated as

$$\hat{\lambda}_{MAP} = \arg\min_{\hat{\lambda} \in \mathcal{L}} -log_2 P(S|\hat{\lambda}) - log_2(\hat{\lambda}). \tag{2.4}$$

Both terms show the encoding of the errors given the data set and the encoding of the model.
Although there exist quite a few solving schemes for the parameter fitting for HMMs, the model learning task is a more complex problem. Additionally, there exist four further tasks: Evaluating, predicting, smoothing and

decoding an HMM. Evaluation identifies the probability $P(h|S,t)$ of a hidden state h at a time point t dependent on an observed sequence S. This can be calculated with the Forward algorithm. The second task, prediction, can also be solved by the Forward algorithm. Here, the probability $P(h|S)$ of being in state h at time point $t+\delta$ (a time point in the future) is calculated, given t, δ and the emission sequence S. In contrast, smoothing calculates the probability of being in state h at a earlier point in time $t-\delta$ and is achieved by the Backward algorithm. The last task, decoding, calculates the most likely hidden state sequence that produces a given emission sequence S at the time point t. The Viterbi algorithm was devised for this task. Both algorithms, the Baum-Welch and the Viterbi algorithm can also be applied in the domain of automata (when standard-sequences are modeled), especially in the case of probabilistic deterministic finite automata. There, the problem setting is even simpler as there exists only one path for each word so that the computational complexity is reduced. In order to model time-labeled states of a process, where the states are described by a multi-dimensional attribute vector, a fully connected ergodic HMM could be used. Due to the fact that standard HMMs model only one-dimensional variables, there are two possibilities of what a state emits in this problem setting. Either each state emits exactly one variable (symbol) or each state holds a probability distribution over all variables O. Actually, the first case is a special case of the second case, where each emission probability is set to zero except one, which is the one of the desired symbol.

In the first mentioned model (one symbol per state), the structure of the HMM is defined as the set of all possible $2^{|O|}$ states that are fully connected. Each state models one subset of variables. The resulting emission matrix B is easy to determine: only one entry of the matrix is set to one, because each state emits exactly one of the $2^{|O|}$ combinations of variables. The determination of the transition matrix is computationally more expensive because there are $|O|^2$ connections within the model. Therefore, A will be large and there is no guarantee that each connection is represented in the data.

The second HMM design for the given problem is to use only a certain number of states which may emit all events. Here, all parameters have to be estimated from the data, especially the emission probability of each variable in each state. Thus A is quite small and $B \approx |M| * |O|$, but the number of states has to be defined by an expert. If the designer follows inadequate hypotheses, the modeling will never be appropriate. However, in both cases the complexity of the parameter estimation is mainly dependent on the number of variables $|O|$. In the domain of multidimensional events, as it is the case in our problem setting, the number of events (variables) is proportional to the number of possible combinations of each variable dimension. Even in the easiest case, where each event is described by d binary variables (so each event is of dimensionality d), the number of possible states is equal to 2^d. This quickly becomes infeasible with growing dimensionality [56] and makes multidimensional problems very hard for standard HMMs. To handle the dimensionality problem, DT-HMMs [62] were proposed. However, they are still not capable of modeling variables with more than two dimensions. So, other approaches to the two main drawbacks (multidimensionality and structure learning) of HMMs were presented. To handle multi-dimensional input and output, *Multi-Output HMMs* [10] were proposed. However, to train them, a hand-made structure has to be given a priori. For every problem, a hand-tweaked model is assumed, which strongly depends on the ideas and intuitions of an expert. Nevertheless, there are tools like MoCaPy, which can be used to model Multi-Output HMMs and to fit models to given observation sequences. To learn the structure of HMMs, algorithms have been studied extensively, but rely on one-dimensional sequences [86] only. So, they are not appropriate for the problem setting that was outlined in Section 1.1 in their current form.

2.2.2 Dynamic Bayesian Nets

Dynamic Bayesian Nets (DBN) are a generalization of HMMs that use recursive HMMs to model time series. Here, several HMMs, possibly having the same structure, are combined via cross connections, so that each HMM models one time step [38] of a multivariate timed sequence. Such networks consist of output nodes Y_t, hidden states H_t and can optionally address input variables (X_t). The hidden states H_t in the DBN reflect multivariate attributes instead of a single random variable only (like it is the case for HMMs). Thus, the output variable Y_t can also be multivariate. Although this type of model can express very complex settings, the creation of such a net can be difficult. One aspect of the model is that one may have to know upfront how many timesteps to include in the model. If, e.g., the model only captures t timesteps, a prediction for the $(t + 1)^{st}$ timestep is not possible, because there was no training for this step. Moreover, the conditional dependency tables for such models may become very large, if Y_t covers many dimensions: each possible combination of feature values must be estimated from the training data, depending on the hidden states' H_t variable setting (and the setting of X_t, respectively). Even for binary multi-dimensional variables, this may become intractable. Nevertheless, algorithms that learn the structure of a DBN were proposed, like *Bayesian model averaging* [45] that estimates whether features (e.g., an edge in a graph) exist. Moreover, standard feature selection algorithms, such as forward or backwards stepwise selection for the identification of edges in such a model, or the *leaps and bounds* algorithm [42] can be applied, if the system is fully observable.

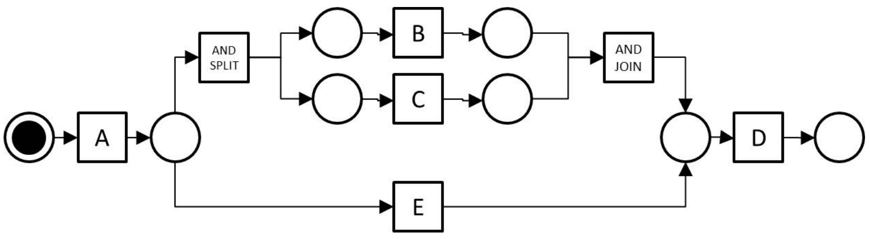


Figure 2.3: Illustration of a Petri net.

2.2.3 Petri Nets

Petri nets [93] address the problem of identifying models from discrete event logs. These models can later be used to explain and transfer the acquired knowledge. In general, Petri nets will only present the structural dependency, but will not give the joint probability $P(s_i|\lambda)$ for an event sequence s_i. Nevertheless, Petri nets can be used to learn log-based model structures. To describe Petri nets, there exist some variants, here Place/Transition networks will be introduced, which are based on nodes and transitions. Formally, a Petri net is a pair (N, s), where N is a tuple (P, T, F) of places P, transitions T ($P \cap T = \emptyset$) and directed edges (flow relation) $F \subseteq P \times T \cup T \times P$. The parameter s denotes the marking of the net. A marking is a bag over the set of places P, i.e. it is a function from P to the natural numbers $f : P \to \mathbb{N}$ that shows how many markers are given in the net. A node x can be an input (or an output) node of another node y if there is an arc going from x to y (or vice versa). The set of all input nodes X of a node y is denoted as follows: $\bullet x = \{y|(y, x) \in F\}$, and the set of all output nodes as $x\bullet = \{y|(x, y) \in F\}$, for any $x \in P \cup T$. The dynamics of Petri nets are given by firing rules that define when a transition is enabled: Transition $t \in T$ is enabled (indicated by $(N, s)[t >)$, iff $\bullet t \leq s$. The *firing rule* $\_[\_ > \_ \subseteq \mathcal{N} \times \mathcal{T} \times \mathcal{N}$ is the smallest relation for any $(N = (P, T, F), s) \in \mathcal{N}$[3] and any $t \in T, (N, s)[t > \Rightarrow (N, s)[t > (N, s - \bullet t + t\bullet)$. Figure 2.3 shows an example Petri net. Places are illustrated as circles while transitions are given by rectangles. There is also a token in this net, it resides in the source place (the leftmost node), which is illustrated by the additional black circle in this node. In this example the source node is enabled and firing the transition would move the token from the input place and puts it to the output place. As a result of this firing, place E and the AND-Split are enabled. Note that in a Petri net, tokens are consumed and produced, they do not travers the net. This can be illustrated in the AND-Split: if this transition fires then one token is consumed while there are two tokens produced. A sequence s is modeled (accepted) by a Petri net (N, s_0), if there exists a sequence of enabled transitions whose firing leads from s_0 to s, given the

[3]N is the set of all marked, labeled Petri nets.

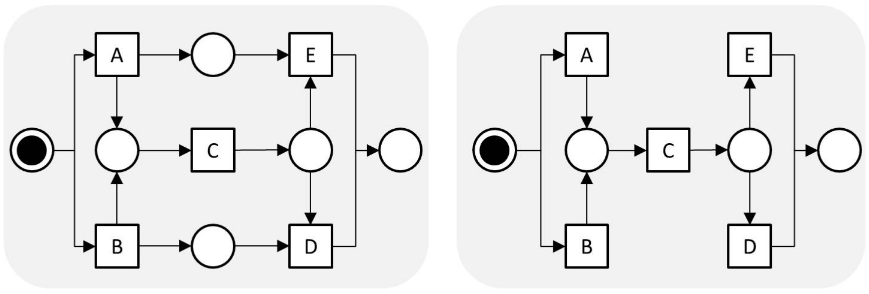


Figure 2.4: Examples of a net (left) that cannot be detected by the α-algorithm, but a similar net is returned (right)

structure of the net and its markers [94]. There exist other characteristics of Petri nets like *connectedness*, *boundedness, safeness* and *liveness*, which will not be described here. The structure of a Petri net is not only composed of places and transitions. As already shown in Figure 2.3, there are also standard building blocks like AND-splits/joins or OR-splits/joins[4] that were used to model the parallel, sequential or conditional processes. There, tasks are modeled by transitions while causal dependencies are modeled by places and arcs. Places can therefore also be considered as conditions that must take place before tasks. The structure of a Petri net is usually derived by first finding an ordering relation of the events and then combining them into an overall model. This does not include the frequency counts of the transitions nor an automatic distinction of the events. All Petri net miners must be given the set of events that may occur in an event log. Let's consider the α-algorithm as an example. First, the ordering of the events is extracted. This can simply be done by inspecting the ordering the events occur in the log. If A always follows B than there can be established an sequential ordering $A \rightarrow B$. Analogous, parallel orderings or even an unknown ordering can be extracted. However, it is assumed that the log is always complete and that every possible relation between two events is observed at least once if there is a relation. In contrast, it is not necessary to observe every firing sequence, because this may be impossible in a net with loops. Having these causal relations, places between them are created, which is the essential property of many Petri net induction algorithms. However, places have to be merged, if an OR-split/join is present. Moreover, short loops (of length one or two) are also a problem for the α-algorithm, which is also true for invisible tasks. Figure 2.4 shows an example of a so-called non-free choice net that cannot be identified by the α-algorithm. However, a similar net is returned instead. Although the α-algorithm cannot handle all types of net-types there exists work that has tackled some of the shortcomings of

[4]AND-constructs only have exactly one in (or outgoing) transition, while OR-constructs always have multiple in *and* outgoing transitions

the α-algorithm. The problem of short loops ($\alpha+$ -algorithm) and implicit dependencies ($\alpha\sharp$-algorithm) was solved [28, 103]. To include invisible tasks, they are first separated into SIDE, SKIP, REDO and SWITCH constructs and then a new ordering relation that reflects the invisible tasks is introduced. The $\alpha{+}{+}$ -algorithm is also able to mine short loops by changing the definition of log completeness (Loop-complete workflow log [28]) and by adapting the pre- and post-processing phases. There, the short loops are identified in a last step that connects the short loops to the existing places of the net.
However, for more complex logs, e.g. numeric logs, the structure is fixed, while the parameters of the net are estimated [104, 52]. Although Petri nets were successfully applied to model business processes and biological pathways, they are not applicable for the given problem setting (cf. Section 4). First, they are usually based on events that consist of one variable only and are therefore unable to model multi-dimensional events, and second, model selection and capacity control remain essential open problems. While, one straight forward way to handle multi-dimensional events would be to cluster them first, and then to learn the net structure (cf. Section 4.2.7) a solution for the net constraints like, e.g. liveliness, was not elaborated, yet.

2.3 Automata

This section introduces grammars and languages together with their induction algorithms and fields of application and discusses present methods for the retrieval of automata.

2.3.1 Grammatical Inference

Grammars are very closely related to abstract automata and consist of rule sets that describe how words can be built up. Chomsky was the first scientist that explored the properties of formal grammars. Nowadays, grammars are the basis of important software components like compilers and are essential for the understanding of the capabilities of software and to verify all kinds of systems like communication protocolls. Especially for programs that are built in a recursive structure. Consider the following example of a typical rule: $S \rightarrow S + S$. This expression means that a term can be created by combining two other arbitrary terms by a 'plus'. Such a rule is very typical for the way, how software is implemented. Another important application for grammars is to validate the structure of data, which is done by regular expressions. They describe in a symbolic way how valid data must be setup. Such a problem can be handled by regular grammars. A grammar (or a language) L is defined over an alphabet Σ: $L \subseteq \Sigma^*$ and describes a final set of symbol sequences. Chomsky described a hierarchy of grammars, following different properties that can also be considered as automata. Figure 2.5

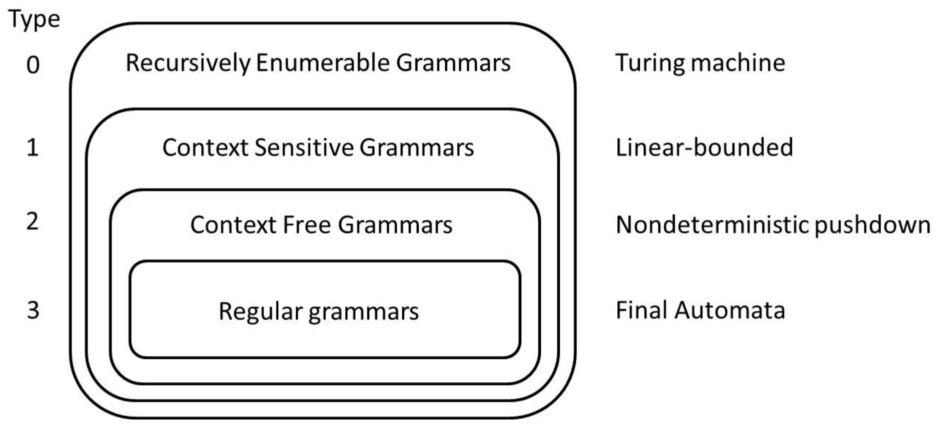


Figure 2.5: Chomsky hierarchy of grammars and their corresponding types of automata

shows the hierarchy of grammars as they were proposed by Chomsky and the corresponding type of automata (cf. Section 2.3.2). They differ in their characteristics, i.e. in the types of operations that can be applied to them and in the type of rules that are allowed for describing the grammar. The smaller the type of the grammar, the more expressive is its language because the number of constraints increase with the grammar type.
A grammar is usually represented by a set of rules of type $N \rightarrow [N|T]*$ where N denotes a non-terminal symbol and T is a terminal symbol. Depending on the grammar type, there are constraints on how the right hand side of the rule may look like. Regular grammars are the most restrictive ones, while Turing machines allow for every possible language.

Grammars can be used to describe, compress or model data. One advantage is that a grammar can handle unbounded sequences and that the resulting models are quite understandable [27]. One main focus in the field of grammatical inference is the induction of regular grammars in a variety of settings, like the induction of only positive examples. The reason why research mainly focussed on the induction of regular grammars is that it seems that learning context free grammars cannot be done in polynomial time. Inductive inference, the complexity of learning algorithms, active learning and PAC-learnability of grammatical inference are very frequently adressed topics [43]. However, heuristic methods for the induction of grammars were more successful [44]. Inducing algorithms for linear languages seems nevertheless promising and a number of postitive results have been published [84]. Using machine learning for the induction of grammars yields many approaches including work on VC-dimensions, version spaces, genetic algorithms and also state merging procedures (cf. Section 2.3.2). Some approaches rely on the Gold paradigm [21], which assumes that every word of the target language occurs at least once in the training data set S of strings $S \subseteq \Sigma^*$. Using the provided examples one by one the learner constructs a model M_l that converges to the final solution. It stops when there is an example s_n for which the induced target concept does not change any more $\exists n \in N : M_n = M_m, n < m$. Usually, the following notation is used: Σ (and Δ) refer to non-empty finite alphabets. Σ^* is equal to the set of all strings over Σ and λ is the empty string (it is the only word having length 0 and is included in every language). A language L is a subset of $\Sigma^* : L \subseteq \Sigma^*$. Let u and w be two strings, $u, v \in L$, then their concatenation is illustrated by uv. A context (l, r) is an element of $\Sigma^* \times \Sigma^*$ which can be wrapped around a string: $(l, r) \odot v = lvr$. Languages can be concatenated: $LM = \{uv|u \in L, v \in M\}$. The set of all substrings of a word w ($sub(w)$) can be defined as follows: $\{u \mid \exists l, r \in \Sigma^*, lur = w\}$. If applicable this is extended to sets of strings. This notation is used to deduce whether a word is part of a language. Depending on which deduction rule is applied, a particular set of languages can be derived. This can be achieved by implementing rule schemas. For example, a schema could define whether a word can be deduced to be in a language by knowing that another word is in the language. By using a set of such rules [21] a language can be identfied. Chain rules are used to simplify the induction process. The link equivalence classes of rule sets then essentially describe the same languages. Rules can either be *correct*, *certain* or *defeasible*. The first rules are always considered to be correct, either by axioms or by the combination of information. However, if there is enough information to conclude that a rule is not correct then it is considered as wrong. Still, the problem remains, whether an infinite language contains a certain word. As such a word may be very long, it may be impossible to conduct whether a rule is true or false. Therefore, each rule is considered as defeasible until there is evidence that it is wrong. To define whether a specific word is part of the language L, proofs are derived from the rules. Clark et al. [21] present a genetic algorithm that uses such a rule system to find a grammar for a (in general unbound) set of examples by retrieving information from a so-called oracle that defines whether or not an string is part of the language so far. This is similar to the famous L* algorithm by Angluin [9] (cf. Section 2.3.2). Using this type of notation and proof formalism many types of languages can be inferred, e.g. Multiple Context-Free grammars or even linear grammars.

There also exists an extension to stochastic grammars, where a probability density function over Σ^* ist given, i.e. there is a probability for each word $w \in \Sigma^*$ to occur in the language L. The stochastic grammars that form such a stochastic language need and additional factor: a probability that the grammar L creates the string/word w which is defined recursively:

$$p(X \Rightarrow \lambda) = p(X \rightarrow \lambda)$$
$$p(X \Rightarrow aw) = p(X \rightarrow aY)p(X \Rightarrow w)$$

To find such a grammar the RLIPs -algorithms was introduced that finds the equivalent automaton (cf. Section 2.3.2). It was applied to speech recognition with noise or other random errors. Another application for stochastic grammars is the identification of structural elements by markups in text documents [106]. One nice property of this approach is that the grammar evolves as new examples arrive and thus a better interactive tuning of the result is possible.

2.3.2 Automata Induction

The term automata was introduced in the 1930s by Alan Turing, who studied which problems can be computed and which not. This is also formalized by the term determinable that includes all problems that can be solved with computers. The second questions is, which problems can be managed by computers, i.e. can be solved within a time span that grows slowly with the size of input. Slowly means that the time can be approximated by a polynomial function. However, this work will not discuss such problems but aims at describing how automata can be used to formally describe processes or systems. In general automata can be considered as a system that consists of a set of states that describe some important properties of the system. Moreover, the system is described in a way that it is exactly in one state at each time point. As there is no memory, former states of the system are 'forgotten' when a new state is reached. Given an input sequence, which may be any symbol of an alphabet, the automaton changes its state according to the provided symbols. Note that there exist only a specific set of start states, i.e. states which 'accept' the first symbol. *Final* (*Accepting*) states describe whether the system is in a valid state. If such a state is reached after an input sequence, the sequence is considered as valid. Figure 2.6 shows a very

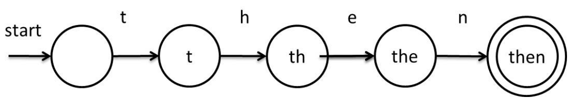
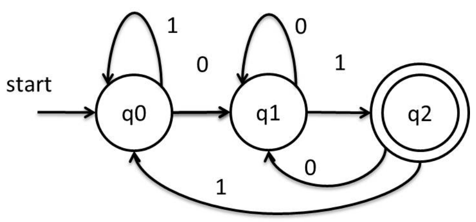


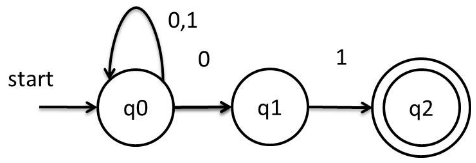
Figure 2.6: Example automaton

simple example automaton that consists of five states. If this automaton receives as input the letters $t - h - e - n$ it reaches the only accepting state. Thus, this automaton is built to parse (or to identifiy) the word *then*.

Types of Automata To describe the class of automata more formally, let's consider one main distinction of automata first: determinism. An automaton is deterministic if it can reside in exactly one state at each point in time. Formally a deterministic finite automaton (DFA) is a tuple $\Gamma = (Q, \Sigma, \delta, q_0, F)$, where



(a) Example DFA



(b) Example NFA

Figure 2.7: Top: Example of a DFA. Bottom: Example NFA. Both automata accept the language of words that end with 01

- Q is a final set of states
- Σ is a final alphabet
- δ is a transition function $\delta : Q, \Sigma \rightarrow Q)$ that defines for each state $q \in Q$ and symbol $a \in \Sigma$ the next state $q' \in Q$
- q_0 is the start state $q_0 \in Q$
- F is the set of final states $F \subseteq Q$

A DFA can decide, whether a word (sequence of symbols) is part of a language – whether the word is accepted. If a langugae, i.e. a set of words is a accepted by a DFA, this language is called regular language (cf. Figure 2.5). Nondeterministic finit automata (NFA), i.e. automata that can reside in several states simultaneously, also accept regular grammars, but are often easier to describe. This can be proven by the fact that each NFA can be transformed in a DFA. Such a type of automaton can be in two states simultaneously because there may be states that have several alternatives when a specific symbol is read. That means that there is more than one next state. Figure 2.7b shows an example for such an automaton. The difference to a DFA is that there are states for which there is no next state for some symbol $\sigma \in \Sigma$. Thus, the definition of such an automaton differs only in function $\delta : Q, \Sigma \rightarrow Q^*$. One important subclass of automata of DFAs are distinguishable automata. This property is fullfilled if there are no two states s and s' and their corresponding trajectories P_s and $P_{s'}$, where the two trajectories are too similar concerning one similarity metric m: $m(P_s, P_{s'}) \geq \mu, \forall s, s' \in Q$.

Probabilistic automata (PA) were developed to model probabilistic systems [87] and subsume Markov chains and Markov decision processes as well. Given an NFA a PA can be constructed with it by adding a probability distribution to each state's transitions that define the probability $p(q_i, a)$ that symbol a is observed after state q_i. As usual the sum of probabilities of the transitions leaving each state sum up to one.

Learning Automata As described before, automata, grammars and graphical causal models (like HMMs) are very closely related and can be used for similar problems. Due to their intuitive structure HMMs were the initial choice for modeling time series in many domains. However, for their correct parameter induction (usually the EM-algorithm is used), which is indeed only locally optimal, the correct underlying topology must be known. This is not necessary when inducing automata [36]. Note that in general each HMM can be transformed in (or at least approximated by) an equivalent automaton (having the same number of states) and vice versa (while here the HMM has not necessarily the same number of states) [36]. When inducing the structure of automata, there exist three main streams in the literature, using the state merging method [12], algorithms that guarantee PAC-learnability [22] (more generally probabilistic models) and a set of algorithms similar to the L^* algorithm [9]. The aim when learning probabilistic automata is the induction of a distribution from a sample that is as similar as possible to the (unknown) target distribution. In the easiest case some prior knowledge is given to the learner so that the topology of the model can be fixed before the parameter estimation. If this is not possible, then structure learning and topology modeling has to be done together [30]. Such approaches have already been shortly introduced in Section 2.2.1 and can also be adopted for specific classes of automata as HMMs can be transformed into automata (PDFAs).
The second main type of the induction of automata is derived from the L^* algortihm, which learns a minimal DFA for a given regular language. The main idea behind such methods is that there exists a learner that has to provide the final automaton. The learner may ask questions to a teacher who knows the correct automaton as well as a so-called oracle that can decided whether the proposed automaton is correct. This setting is schematically illustrated in Figure 2.8. The L^* algorithm is based on two sets $U \subseteq \Sigma^*$ (words that are candidates for identifying states) and $V \subseteq \Sigma^*$ (words to distinguish states). The learner identifies candidate words from $(U \cup U\Sigma)V$ and queries the teacher whether they are part of the language. The result of this query is stored in a table $\mathcal{T} = (\mathcal{T}, \mathcal{U}, \mathcal{V})$. The learner tries to identify a *closed* and *consistent* language H from this table and queries the oracle whether this is correct. If the automaton is not correct, the oracle returns a counterexample that is used to update the sets U and V accordingly. This

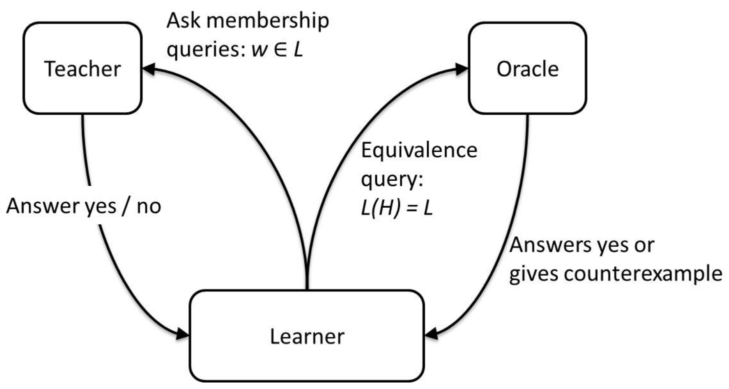

Teacher
Ask membership queries: $w \in L$
Oracle
Equivalence query: $L(H) = L$
Answer yes / no
Answers yes or gives counterexample
Learner

Figure 2.8: Illustration of the problem setting for by the L^* algorithm [9]

leads to the creation of new membership queries until the language is again closed and consistent. This is repeated until the correct language is identified.
Although there exists much theoretic work about the bounds and quality approximations using such algorithms, the third induction group (state merging methods) has proven well in practice. They require a significant smaller sample size to infer the given automataon (or grammar) [14]. Automata induction methods that rely on the state merging strategy are based on a so called prefix tree acceptor [30], which is a DFA that can only generate the data set S, i.e. no generalization is present. Upon such a structure the merging procedure is applied. Nodes of the PTA are merged (combined into one state) successively. The search space of such a method is given by all automata that can be derived by merging the states of a PTA, from which the best fitting model has to be selected. How 'best' is considered will be described in the following sections and is different in the specific approaches. Nevertheless, Algorithm 1 shows a generic learning scheme for induction methods relying on the state merging procedure. It uses the given data set S and a precision parameter that is dependent on the specific merge-criterion. The basic steps are first, the selection of pairs of nodes that are considered for merging (SelectStates), then the evaluation whether they are compatible (which is also depending on the specific algorithm) and if so the update of the current automaton. The update is essentially the merging of q and q'. The following algorithms can be considered as specifications of one of these main steps each. One example of algorithms that implements this basic structure is the Alergia algorithm [12]. The sequences are selected in alphabetical order and evaluated in pairs whether to be merged. This is the case when the following condition, derived by the Hoeffding bound, is fullfilled:

$$\forall a \in \Sigma :$$

Algorithm 1 *General state merging method* (Histories H, precicion param. μ)

```
1: $\Gamma \leftarrow PTA$
2: while stopping criterion not satisfied do
3:     $(q, q') \leftarrow SelectStates(\Gamma)$
4:     if Compatible(q,q',$\mu$) then
5:         $\Gamma \leftarrow Update(\Gamma, q, q')$
6:     end if
7: end while
8: return $\Gamma$
```

$$\left|\frac{C(q,a)}{C(q)} - \frac{C(q',a)}{C(q)}\right| < \sqrt{\frac{1}{2}ln\frac{2}{\mu}}\left(\frac{1}{\sqrt{C(q)}} + \frac{1}{\sqrt{C(q')}}\right).$$

This condition compares the outgoing transitions of state q and state q' respectively and also recursively the successore states of q and q'. If q and q' are compatible, they – as well as their successors, if appropriate – are merged to eliminate non-determinism. This strategy is therefore called *determinization by merging.* This algorithm identifies the class of probabilistic determininistic regular languages in the limit.
The basic algorithm was also modified to induce acyclic automata [73] reflecting finite languages. There, candidate states must be in the same depth of the PTA to be merged to ensure the acyclicity. Besides, states must be frequent to be merged. Otherwise they are combined into one state that reflects exceptions. Finally the transition probabilities are corrected by the maximum likelihood estimates:

$$\varphi(q,a) = \frac{C(q,a)}{C(q)}(1 - (|\Sigma| + 1)_{\varphi_{min}}) + \varphi_{min}$$

, where φ_{min} is the minimal transition probability in the automaton. Another adaptation is the MDI algorithm [88] that aims at finding of minimal sizes and small deviations from the data set S. Here again the two states q and q' are merged to a temporary solution Γ_1 and eventually with additional states to solution Γ_2. This is done if the divergence increment relative to the new (smaller) size of the automaton is less then μ:

$$\frac{D(\Gamma_0||\Gamma_2) - D(\Gamma_0||\Gamma_1)}{|\Gamma_1| - |\Gamma|} < \mu,$$

where $D(\Gamma_0||\Gamma_2)$ reflects the divergence of the automata Γ_1 and Γ_2.
The RLIPs- algorithm [13] also uses the state merging method on a prefix tree acceptor to find the minimal stochastic automaton for a given stochastic regular language from a positive data set only. Therefore, a branch and bound algorithm is used. This type of algorithm is also able to identify

cyclic structures of an automaton and was proven to converge having a lower bound on the provided data set. In 2006, Gavalda et al. proposed a framework (based on an algorithm of Clard and Thollard [22]) that is based on a state merging and splitting strategy [36]. The induction strategy of such algorithms is incremental although the necessary statistics are computed in a batch mode. Starting from an automaton that includes all knowledge that could be collected up to this timepoint, a new sequence is mapped against the automaton until its end is reached or one symbol of the sequence reaches a so called candidate node u. This is the case when there exists no transition labeled with σ_i out of the current (safe) node w. If there is no safe node v such that u is similar to v (which is tested every time after reaching u) then u is promoted a safe node and added to the list of final states of the automaton. This can be summarized as described in Algorithm 2. Moreover,

Algorithm 2 *CT Algorithm (Trajectories D) [36]*

```
1: for all $d \in \sigma_0, \ldots, \sigma_k$ do
2:   while $\sigma_i$ matches a state $s$ do
3:     a safe state can be reached
4:     i++
5:   end while
6:   if $i < k$ then
7:     candidate state $s$ is reached
8:     if $s$ is not large/frequent then
9:       retain node $s$
10:    else if $s$ has a similar state $s'$ then
11:      merge node $s$ with $s'$
12:    else
13:      promote $s$ a new safe state
14:    end if
15:  else
16:    add $\sigma_{i+1}, \ldots, \sigma_k$ to $D_{s\sigma_i}$
17:  end if
18: end for
```

this algorithm ensures that

For every PDFA M with n states, with distinguishability $\mu > 0$, such that the expected length of the string generated form every state is less than L , for any $\delta > 0$ and $\epsilon > 0$, the PDFA-Learn algorithm will output a hypothesis PDFA M' such that, with probability greater than $1 - \delta$, the maximum difference in the probability assigned by the PDFA to any string is at most ϵ .[36]

However, the algorithms proposed by Castro et al. [14] differs in one main aspect from previous work: they need smaller sample sizes but still provide the quality estimates of [36]. The algorithm presented by Clard and Thollard (for brevity the CT-algorithm) required all quality bounds as an input from the user and then calculated the required data set size that had to be provided afterwards. This could of course lead to significant drawbacks, if such a data set size does not exist for real world applications. Then such a method cannot be applied. The improved CT-algorithm requires as input the alphabet Σ, an upper bound L of the expeted length of emmitted strings of the target (the true underlying automaton), an upper bound n on the number of states of the target and the quality parameters δ (confidence) and ϵ (precision). The aim of the algorithms is to induce an isomorphic graph G to a subgraph A of the true target such that all frequent states of A have a corresponding state in G as well as frequent transitions. Additionally, the symbol emission rates deviate at least ϵ_1 (which is one parameter to form ϵ) from the predefined ones. By using these user specific thresholds quality guarantees can be given. This algorithm was tested on small sample sizes and indeed showed that it requires far less input to deduce the correct automaton. Moreover, first experiments with real world data sets (web logfile) showed that interpretable but non trivial structures can be identified. Just to mention remaining work, other methods to induce automata are based on the state splitting or error-correcting method. However, as this thesis does not include such strategies they will not be revised here.

Recently, a new type of automata inducing algorithms have been proposed [98, 95] that rely on the state merging procedure but do not claim quality constraints. The state merging procedure was chosen because it is currently considered the best solution for the task of learning a DFA [27]. The main advantage is that the structure of the model can be learned automatically. It does not need to be specified manually. Additionally in these algorithms, time information is incorporated into the final model, which is a deterministic real-time automaton (DRTA) [29]. The input for the algorithm is an event sequence $\tau = (\vec{e}_1, t_1)(\vec{e}_2, t_2)\ldots(\vec{e}_n, t_n)$, where each event $\vec{e}_i$ has an associated time stamp t_i, which reflects the time that has elapsed since the last events. Although the structure of a DRTA is like a DFA, each transition has a delay guard assigned. A delay guard is defined as an interval t_1, t_2 in $\mathbb{N}$ and specifies for which times the transition is allowed to take place. This is the case when time t lies in the interval defined by t_1 and t_2. Real-time automata are defined as a tuple $A = \langle Q, \Sigma, T, q_0, F\rangle$; a set of states (Q), an alphabet (Σ), a set of transitions (T), a start state (q_0) and final states ($F \subseteq Q$). The automaton is deterministic because there is exactly one transition $\langle q, q', a, \phi\rangle \in T$ for a state q, every symbol a and every time value $t \in \mathbb{N}$. To identify the automaton that is the smallest one, consistent with the input, a timed prefix tree is created. To define the final states of the model, the states of the prefix tree are merged, which means that two states are combined into a single one. The merging procedure is mainly dependent

on the final structure of the automaton. Automata not including a delay guard may use (k,h)-contextuality [3] to identify the states to be merged. Here, similar paths are identified and then $k-h+1$ states of the paths are merged. However, for automata with delay guards, the deterministic constraint can be violated after a merge: There may be two transitions with the same symbol and delay guard. This mismatch is resolved by a splitting procedure that creates two paths in the prefix tree with non-intersecting delay guards. Although state merging is currently state of the art, there exist different methods of how to merge and split states and transitions. One possibility is to do splits and merges depending on their consistency (derived by a χ^2 test [96]) and apply a rule set to identify whether a merge or a split is the best next step. Note that this algorithm relies on positive data only, which means that all input data are expected to be part of the language that is formed by the automaton and thus must be accepted by it. Another method to find the best split or merge is to compute the likelihood of an overlap of the two tails of a state in the prefix tree [99]. Again, a score is computed to select the best operation (merge or split). This operation is based on the red blue algorithm.
Still, one has to remark that automata and causal models are closely related. Comparing, e.g. automata to HMMs, the parameter λ of the automaton is again a transition matrix B, which is derived by ML estimation (parameter learning problem). It covers transition probabilities depending on multi-dimensional symbols and time: $b_{ij} = P(transition_{ij}|\vec{e}_k, t_k)$. The introduction of probabilities offers the possibility to construct a generative (automaton based) model and to compute the probability $P(S|\lambda)$ for each observation sequence S depending on the model parameter λ. Moreover, prediction (although in general not important for automata) is also a desired function in this work and will be discussed in depth in Section 4.1.3. In contrast, the evaluation task has no true equivalent in the world of automata. The same is true for smoothing and decoding. In general, it is not important to know in which state an automaton resides but if the automaton accepts the sequence S. This shows whether a sequence S is part of the language that is modeled by the given automaton. Moreover, deterministic automata provide no transition probabilities, so the calculation of state sequences is straightforward. Considering non-deterministic probabilistic automata, it is possible to compute $P(h|S)$ depending on the transition probabilities, so that smoothing, decoding and evaluation can in principle be conducted as in the domain of HMMs. However, as stated above, answering these questions is not the main task of automata. It is only of interest whether a sequence S is accepted by an automaton.

2.4 Applicability to the problem setting

The main idea of this work, was to enable the temporal analysis of biological and medical data sets. The important subdomains here are first the prediction of disease progressions in a population (cf. Chapter 2) and second the identification of relationships between genes. One task is to find out in which sequence a cell expresses which genes. Another task is to find out how the genes interact with each other and the resulting metabolic state of the cell. This also includes the transduction of signals within a cell, which is very complex and by no means already understood. Nowadays, microarray experiments allow for the simultaneous evaluation of several thousands of genes over a specific time period under various environmental constraints. Then, the genetic interactions and cell-responses to stress can be observed. Therefore, the information of how genes interact and depend on each other is very interesting and there are quite a lot of approaches to deal with such problems. One possibility is the extraction of temporal association rules (TARM)[65]. However, most of the approaches suffer from the problem that there exist very many genes but only a few timepoints of observations. Thus, using analytical/Bayesian approaches (covariance analyses and so forth) may not result in appropriate results as the number of training instances is very small and there may be too few observations for an approximately correct estimation. Besides, the incorporation of time constraints is also one essential point. The user wants to specify which time periods has to be modeled, i.e., how the gene expression ratios develope after, e.g. 10 minutes. The next issue is that the events or characteristics are not necessarily known upfront, so that the desired algorithm must find out states or appropriate variable settings without prior knowledge. One obstacle here is that there may be noise, which blurs events and lets them seem different although they are actually the same. One example here is that a genetic experiment incorrectly returns a negative expression for a gene because it was just not sensitive enough. Therefore, we define a list of characteristics that have to be covered by an approach to produce an understandable model that can handle the discussed problems. The model should

- produce a probabilistic generative graphical model,
- handle multi-dimensional variables,
- consider time stamps,
- allow for cycles, and
- generalize similar events due to noise.

Up to now, all the discussed approaches can deal with one or more of the desired properties, but all of them have limitations that are not easy to solve. However, automata have already been extended to take into account the time constraint. Therefore, we decided to push this extension forward and to create an automaton-based model that not only automatically defines its states from the given set of events, but also may address noise and cyclic structures.

2.4.1 Combining the best of the presented approaches

All existing automaton based algorithms are made for one-dimensional events only. Nevertheless, only few modifications are required to deal with multi-attribute events. To do so, an approach to combine probabilities with automata was presented [96]. It models one-dimensional symbols and time by using interdependent probability distributions, but transition intervals must be given in advance, which leads to a model with many parameters. In contrast, we solve the problem of multi-dimensional attributes by annotating states with dynamic profiles instead of symbols, and derive joint probabilities for transitions during the merge operation. In this way, selected properties of HMMs (transition probabilities, Markov property), causal networks (graphical representation), Petri nets (cyclicity) and automata (delay guards) are combined with a generalization approach, to obtain the desired functionality. Definition and inference of such an automaton will be presented in Chapter 4 and its subsequent extensions for large sparse data sets (Chapter 5) and to incoporate background knowledge (Chapter 7). Last, an approach to handle data streams, even with concept drift is presented (Chapter 9). These algorithms and their results are summarized and, further, interesting work, like an algorithm improvement for dense data sets, will be introduced.

Chapter 3

Preliminaries: Materials and Quality Measures

3.1 Data

In this section we briefly introduce the synthetic and real world data sets we used for the evaluation of the algorithms. Synthetic data are used to evaluate the correctness of the method and real-world data are used to examine the descriptive power. The real-world datasets encompass medical and biological data. Data sets that were used exclusively for the evaluation of specific parts of the PRTA induction are described separately in the specific chapters.

3.1.1 Synthetic data

A first benchmark is always whether the proposed algorithm can reproduce a predefined structure, i.e. an automaton, like it was done in existing work [36]. To show that the algorithm rediscovers an automaton that is known, histories were produced by a synthetic automaton (cf. Section 4.1 for the nomenclature of used automata). To do so, we created an automaton with 10 states and 10 attributes. This automaton produced 100 synthetic histories by traversing the states corresponding to the transition probabilities. For each history the profiles of the visited states were saved. Afterwards, those histories were used as input for the algorithm. Additionally, with an error ratio of 0.1, we randomly generated errors in the profiles to see if the automaton will be detected correctly although the states are not completely equal. With a probability of 0.1, each attribute of a profile is changed into its opposite. If it is one, it will be changed to zero, and vice versa. The goal of the introduction of errors is to test if the algorithm will nevertheless produce the correct structure although noise is present. The handling of errors is important because real-world data is expected to be error-prone. Figure 3.1 shows the synthetic automaton that produced the synthetic his-

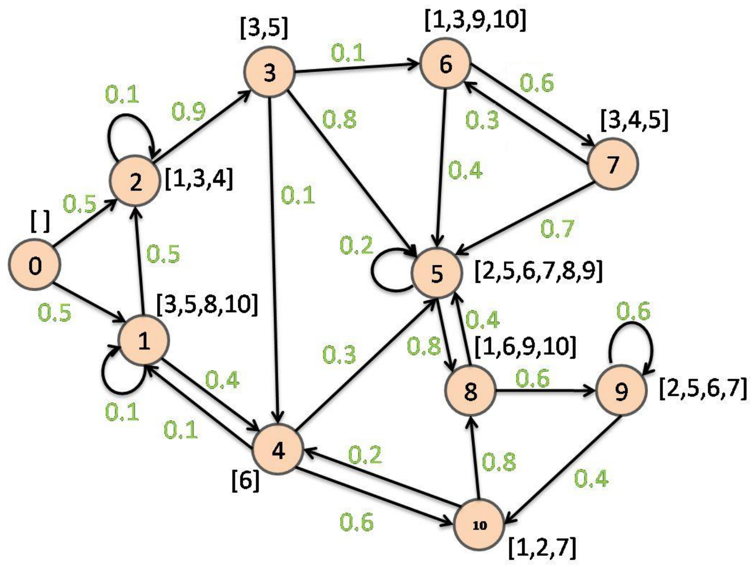


Figure 3.1: Illustration of the synthetic automaton used in the experiments. Each circle is a state, each arrow a transition. The actual profile of each state is given in brackets and each transition is labeled with its probability. For simplicity, the delay guard (not illustrated) on each transition is $\phi = [1, 1]$.

tories. Transitions are labeled with probabilities, and next to the states all attributes that are equal to one are shown (in brackets). They correspond the profiles. Delay guards are always equal to one and can therefore be neglected in this case. The described automaton was also used for a proof of concept and further stability analyses (cf. Section 4.2). Therefore, a data set containing 100 histories of average length 10 was produced by traversing the automaton corresponding to the transition probabilities. For a stability analysis (cf. Section 5.2), ten more synthetic data sets were created in the same manner. They serve as a starting point for a bootstrap analysis. From each of these ten data sets, 100 derived data sets are created by the bootstrap method. Thus, 1000 data sets are used in the stability analyses. To examine the algorithm's scalability for data streams, even larger data sets were created (cf. Section 9.2). They range from 50 to 100,000 histories, where the history length was fixed to be 100. Again, to account for data set variability, for each number of histories 10 data sets are created. To address the stability of the algorithms when noisier data sets are present, the synthetic automaton was also used to create data sets having a different amount of noise. This error ratio ranges from 0.1% to 10%. Another analysis addresses the algorithms dependency on the length of the histories. Thus, the synthetic automaton also served as a basis for data sets having different history lengths. They were varied between 10 an 1000, while the data set size was fixed to 10,000.

3.1.2 Real World Data Sets

Disease Group Data Set I

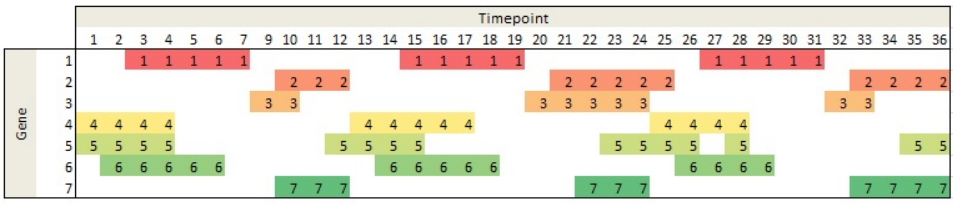
The first real-world, proprietary dataset comprises diagnosis data from 1000 persons from four years. Each diagnosis was grouped into one of 106 medically motivated disease groups (DGs) and saved with its timepoint, namely the year it occurred in. The grouping is based on the similarity of disease patterns and the probable progression, relying on the three-digit WHO ICD10[1] codes of chronic conditions. For each timepoint, we obtain 106 attributes that represent the event of the timepoint (cf. section 4.1). However, the attributes of events are quite sparse: Just 28 out of 106 DGs occur in more than 5% of the instances, 66% of all events just have one or two DGs set, and 99% of the events incorporate less than 11 DGs. Thus, only DGs that occur often (in more than 5% of the instances) were used to define events. Because this data set shows the progression of diseases, only instances having at least one timepoint with an event e where $|\vec{e}| > 0$ were incorporated into the histories.

Disease Group Data Set II

The last (non-public) data set covers historic diagnoses from 147'656 patients within four years on a quarterly basis[2]. Thus, this data set shows the progression of diseases within a population. As there are about 15'000 diagnoses (ICD codes), each provided diagnosis was grouped into one of 111 clinically homogeneous diagnosis groups that combine several diagnoses corresponding to their similarity and expected outcome. Using this compressed medical representation enables the user to manually inspect and judge the resulting states of the automaton. Therefore, a history is a feature vector for a patient and timepoint t describing the set of disease groups at timepoint t. We used two derived data sets to build the automaton. The first data set only contains disease groups (DGs) that occurred in at least 10% of all patients (22 attributes) in order to obtain the most frequent and important disease patterns ($P10$). The second data sets includes all disease groups that occurred in at least 1% of all patients to allow for a more extended inspection (76 attributes, $P01$). The final PRTA then shows which diseases an specific population suffers from and which transitions between such disease states occur and how often. This may enable physicians to better foresee future impairments.

[1]http://apps.who.int/classifications/apps/icd/icd10online/

[2]We gratefully acknowledge Gesundheitsforen Leipzig GmbH



		Timepoint																																
		1	2	3	4	5	6	7	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
Gene	1			1	1	1	1	1							1	1	1	1	1								1	1	1	1	1					
	2									2	2	2									2	2	2	2	2								2	2	2	2
	3								3	3										3	3	3	3	3								3	3			
	4	4	4	4	4								4	4	4	4	4								4	4	4	4								
	5	5	5	5	5							5	5	5	5								5	5	5	5		5							5	5
	6		6	6	6	6	6							6	6	6	6	6								6	6	6	6							
	7									7	7	7										7	7	7									7	7	7	7

Figure 3.2: Illustration of the binary yeast cell cycle data, the rows represent features, columns instances/different timepoints. If an attribute is equal to one, it is represented by the corresponding (colored) number.

Hepatitis Data Set

The next data set is the 2004 ECMLPKDD Hepatitis challenge data set[3]. It contains blood test results for 1236 patients suffering from either Hepatitis B or C between 1982 and 2001. Next to some demographic information like age and sex, the results of up to 36 tests are given for each individual examination. Some patients are only recorded once while others have a history of 401 records. However, 95.7% of the patients provide a history of at least two events, where one event is considered as the examination result of one day. The fillgrade of the attributes in this data set varies strongly. To obtain meaningful results, only the attributes that are present in at least 80% of the events were included in the histories. Moreover, each attribute was discretized in three subtypes: blood test result below normal, normal and above normal. For missing test results none of the attribute's values was set. Thus, the final events consist of 33 attributes (cf. Table 7.2 for a short overview). To evaluate the algorithm's scalability, several data sets of different sizes (50, 100, 150, 200, 300, ..., 900, 1000, 1236 histories) were created.

Yeast metabolism data

A further real-world data set holds the gene expression values of budding yeast [74] (GEO ID: GSE3431) that were recorded using Affymetrix chips (GPL90) for 36 timepoints with a delay of 25 minutes each. The data set covers the expression rates of 9335 genes of a synchronized (all cells start in the same state) cell culture and is freely available[4]. As cells also change in their metabolism, genes are expressed (used) at different time steps, revealing expression peaks in the data. These peaks give information about the current state of the cell, e.g. which metabolism is currently activated. However, the data is given as an expression series of continuous variables and not as zero/one vectors (gene is off/on). Therefore, the expression profiles

[3]http://lisp.vse.cz/challenge/ecmlpkdd2004/
[4]ftp://ftp.ncbi.nih.gov/pub/geo/DATA/SeriesMatrix/GSE3431/

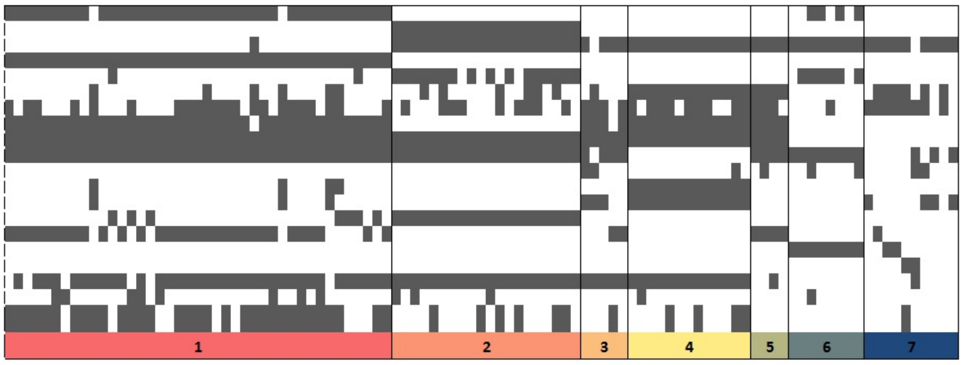


Figure 3.3: Illustration of the binary zoo data set, the rows represent features, columns instances. Each attribute that is equal to one is illustrated with a black box for each instance. The numbers at the bottom illustrate the class label.

were discretized via a sliding window approach (cf. Figure 3.2). A peak is considered as a raise of the expression level compared to the surrounding timepoints. Following this intuition, a gene's expression level was set to one (peak) at a timepoint t_i if the expression level $L(t_i)$ is higher than the average in the surrounding window. There are genes that show a periodic behavior, and they expose peaks in a maximal 12 time steps frame. Thus the window size was set to 12, meaning that the average was taken of the 12 timepoints before and after t_i. Note that other discretization methods (as discretization by mean and standard deviation) may not be as expressive because some peaks could be left out, due to their relatively lower height. To describe the state of a cell, only genes that are expressed at specific timepoints are considered interesting. This means that their expression level should clearly change over the monitored timepoints. Furthermore, each time step should be described by at least one expressed gene so that only genes were selected that reveal peaks and, moreover, peaks at different time steps. Because most genes are expressed at the same time steps, the selection only covers seven well-investigated genes. A Z-score normalized expression profile of these genes is illustrated in the lower part of Figure 4.7 while a binary illustration of this data set is shown in Figure The genes are numbered and highlighted with color at a timepoint, if their expression level was high enough.

Zoo Data Set

The zoo data set describes 101 animals by their morphologic features like, e.g. feathers or breathing and is used to assess the quality of the resulting clustering. Altogether, the data set contains 17 features, thereof 16 binary. The remaining feature (number of legs) was transformed into five binary features ('has 2 legs', etc.) to fit the given problem setting (cf. Figure

3.3). Additionally, a class (the corresponding genus) is provided so that a cluster quality assessment is possible. The data set can be found in the UCI Machine Learning Repository[5].

3.2 Quality Measures

The quality of the automaton is measured by several indicators. For the synthetic data sets the true underlying structure is known. Therefore, the recovery rate (RR) specifies how many states were correctly identified by the automaton: $RR = \frac{|Q_{corr}|}{Q_{total}}$, where the state is correctly identified when its set of MFPs corresponds to an original state. Additionally, the runtime and the final number of states of the approach is given. To evaluate the quality of the induced clustering, the Adjusted Rand Index [46] (ARI) is measured. It computes the overlap between a predefined clustering and the induced one. It is an adjusted version of the Rand Index [50] (RI). While the Rand index shows how many of the instances were grouped together (and apart respectively), the adjusted Rand index also compares these numbers to the amount of expected overlaps. The ARI also overcomes the limitation of the RI that its expected value is not constant. Let $C = \{C_1, \ldots, C_z\}$ be the partition induced by the cluster algorithm and $P = \{P_1, \ldots, P_z\}$ the predefined partition. Then, each pair of instances can be assigned to the same cluster or to two different clusters in each partition. Let a be the number of pairs belonging to the same cluster in C and to the same cluster in P. The ARI rates the agreement between C and P following Equation 3.1,

$$ARI(C,P) = \frac{a - exp(a)}{max(a) - exp(a)} \tag{3.1}$$

$$max(a) = \frac{1}{2}(|\pi(C)| + |\pi(P)|) \quad exp(a) = 2\frac{|\pi(C)| \cdot |\pi(P)|}{p(p-1)}$$

$$|\pi(C)| = \frac{1}{2}\sum_{k=1}^{z}|C_k|(|C_k| - 1) \quad |\pi(P)| = \frac{1}{2}\sum_{k=1}^{z}|P_k|(|P_k| - 1),$$

where p is the number of instances and $|C_k|$ gives the number of elements in the cluster C_k. Notice that when $ARI(C,P) = 1$, we have identical partitions. To judge the quality of the resulting transitions, the F-Measure is used.

$$F = \frac{2 * Recall * Precision}{Recall + Precision} \tag{3.2}$$

It is based on the number of correctly inferred transitions of all transitions to be found (Precision) and the fraction of correctly identified transitions (Recall). The better the prediction (or selection) of transitions the closer

[5]http://archive.ics.uci.edu/ml/datasets/Zoo

is the F-measure to one. An F-measure of zero indicates that no transition was identified correctly. Besides, the accuracy of the induced states must be evaluated. If the underlying structure of the automaton is known, it can be compared to the induced structure. First, the number of induced states is compared to the number of states in the original automaton (Δ_{States}). Then, the distance of each induced state to one original state having most similar profile is calculated. The average Eucledian distance is then expressed by the term L_{States}. These two measures have already been proposed in a similar form in existing work [14], where for the most frequent states of the target a corresponding state in the induced automaton should be induced. However, as the aim of this work is also to include infrequent events and thus states, a representative for each state of the true underlying automaton should be found. As described in the previous Section 3.1, when an evaluation is performed, where the underlying automaton is known, it is ensured that each event and thus state occurs at least once in the synthetic data set. Additionally, the runtime of the proposed method is monitored to evaluate its performance. Finally, the resulting models are visually inspected, to judge how good they can capture the information residing in the data sets and whether a user understands the resulting model. This is for instances evaluated by a selection of paths and their corresponding profiles. This can be compared to existing knowledge in medicine or biology.

Chapter 4

Learning PRTAs from Multi-Attribute Event Logs (*PRTA*)

This chapter describes the fundamentals of PRTA learning. First, PRTAs are defined and subsequently, two basic problems in the theory of automata learning are presented along with their solution. That is how words of a language can be described with such an automaton and second how they can be tested whether they are accepted by a PRTA. Then, the basic algorithm for the induction of PRTAs is presented and last, the possibility to make predictions with such a model will be described. The last section evaluates the proposed model by several experiments on synthetic and real world data sets.

4.1 Probabilistic Real-Time Automata

In this section, we present an algorithm for learning probabilistic real-time automata (PRTAs) which is, like the currently best method for learning automata [27], based on state merging in a prefix tree. Our type of automaton models a discrete event system (DES) [98]. Let dataset D of instances I be given $D = \{I_1, \dots, I_n\}$, where each instance I_i represents a sequence of timed events: $I_i = (\vec{e}_1, t_1)(\vec{e}_2, t_2) \dots (\vec{e}_k, t_k)$. This event sequence, ordered by the time of occurence, is called a *history*. An event e_i is a binary vector $\vec{e}_i = (a_{i1}, \dots, a_{im})$ that specifies whether attribute a_{ij} is observed $(a_{ij} = 1)$ in this event. Because the events e_i have a time stamp t_i assigned, a timed language model is created. Each event $(\vec{e}_i, t_i)$ can also be described by a conjunction of all attributes that are present at time t_i[1]. Let $|\vec{e_i}|$ denote the number of attributes equal to one in this event. Every time-stamp value

[1]This is similar to an itemset representation.

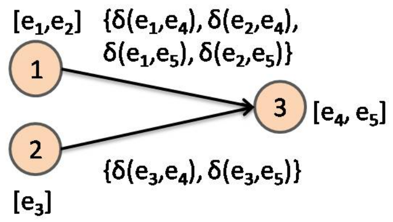


Figure 4.1: Event annotation of transitions and states. Each state's profile consists of the events that are incorporated in the state. Each transition from state q_i to q_j is labeled by the difference of the events that were observed between states q_i and q_j.

$t_i \in \mathbb{N}$ represents the time that has elapsed since the previous event of the instance has occurred. A *PRTA* is a directed graph with states Q and transitions T. Each state q_i holds its set of events E_i and is – for simplicity of notation – annotated by a so-called *profile* f_i. The profile shows the mean attribute/feature vector of all events that are mapped to q_i:

$$\vec{f_i} = \frac{\sum_{e \in E_i} e}{|E_i|}. \tag{4.1}$$

In other words, a profile is just a summary of these events. Transitions $t_{ij} \in T$ of the PRTA connect two states q_i and q_j and are annotated with a delay guard to reflect the observed time intervals between the connected states. A delay guard is defined as an interval $[t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}$, where $t_1(t_2)$ defines the minimal (maximal) number of time steps when this transition can be passed. Additionally, transitions $t_{i,j}$ of the PRTA are labeled with the set $T_{L_{i,j}}$ that describes the changes of the profiles from state q_i to q_j. These changes are expressed in the so-called *delta notation*: $T_{L_{i,j}} = \Delta(E_i, E_j)$, where

$$\Delta(E_i, E_j) = \bigcup_{\vec{e_k} \in E_i, \vec{e_l} \in E_j} \delta(\vec{e_k}, \vec{e_l}) \tag{4.2}$$

and $\delta(\vec{e_k}, \vec{e_l})$ is defined as the difference of the binary vectors $\vec{e_k}$ und $\vec{e_l}$: $\delta(\vec{e_k}, \vec{e_l}) = \vec{e_k} - \vec{e_l}$. Thus, the label is the set of differences between the elements of E_i and E_j and can be interpreted as the set of change vectors that are necessary to reach q_j from q_i. Figure 4.1 gives an example of how transitions and states incorporate the events and the corresponding differences. To complete the transition, it has assigned a probability $p_{i,j}$ of occurrence. The sum of all probabilities of outgoing transitions of a state is equal to one. In general, automata have a set of start (S) and final states (F) that are a subset of all states in the automaton ($S \subset Q \wedge F \subset Q$). In a PRTA, $S = Q$ and $F = Q$, because each state is allowed to be a start or final state. A PRTA is then formally defined as follows:

Definition 1 *A PRTA* Γ *is a tuple* $\Gamma = (Q, \sum, T, S, F)$*, where*

- *Q is a finite set of states*
- *Σ is a finite set of events to label the transitions*
- *T is a finite set of transitions*
- *$S = Q$ is the set of start states*
- *$F = Q$ is the set of final states*

A state $q_i \in Q$ is a pair $\langle E_i, \vec{f_i} \rangle$ where E_i is its set of events ($E_i = \{\vec{e_k} : \vec{e_k} \in C_i\}$) [2] and $\vec{f_i}$ is an attribute vector called its profile. Σ are all events $\vec{e}$ that are observed in the input data. A transition $t \in T$ is a tuple $\langle q, q', T_L, \phi, p \rangle$ where $q, q' \in Q$ are the source and target states, $T_L = \Delta(E_i, E_j)$ and ϕ is a delay guard defined by an interval $[t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}$. p defines a probability $p \in [0, 1]$ that this transition occurs.

4.1.1 Accepting Words

One task of automata is to decide whether they accept a given word of a language. This section will describe how this problem can be solved for a PRTA using the Δ-notation.

Definition of Words

Let $\Sigma = \{\vec{e_1}, \dots, \vec{e_k}\}$ be an alphabet over binary vectors. Then $L \subseteq (\Sigma, \mathbb{N})^+$ is a language over pairs of the alphabet Σ and time points of $\mathbb{N}$, and $(\vec{e_i}, t_j)(\vec{e_k}, t_l) \dots (\vec{e_m}, t_n)$ is a word with time labels from L. In such languages, a time point t_i denotes when its corresponding element $\vec{e_i}$ has occurred. Timepoints are given relatively, i.e. each time point reflects the time that has elapsed since the last event. As an example, consider language

$$L = \{(\vec{e_1}, 4)(\vec{e_4}, 2)(\vec{e_{18}}, 1),\ (\vec{e_2}, 3)(\vec{e_1}, 6)(\vec{e_5}, 3),\ (\vec{e_9}, 10)\}$$

Further, let the Δ-notation for a word $w = (\vec{e_i}, t_j) \dots (\vec{e_m}, t_n)$ be given by

$$\Delta(w) = (\delta(\vec{e_1}, \vec{e_2}), t_2)(\delta(\vec{e_2}, \vec{e_3}), t_3), \dots, (\delta(\vec{e}_{m-1}, \vec{e_m}), t_m) \tag{4.3}$$

Again, it shows the difference from one event to the next, and the time that has elapsed. The problem of deciding whether an automaton Γ accepts a word w (if w is part of the language L modeled by Γ) is transformed into a check if there exists a valid sequence $G = q_0 T_1 T_2 \dots T_{m-2} T_{m-1}$ of transitions, which comprises the word w in Δ-notation.

$$(\delta(\vec{e_i}, \vec{e}_{i+1}), t_{i+1}) \in (L_{T_i}, \phi_i) \quad \leftrightarrow \quad \delta(\vec{e_i}, \vec{e}_{i+1}) \in L_{T_i} \wedge t_{i+1} \in \phi_i$$

The first transition must leave state q_0 that represents w_1: $w_1 \in E_0$. A sequence is valid if and only if succeeding transitions share states (are adjacent):

$$\forall\ T_i, T_{i+1} \in G : Target(T_i) = Source(T_{i+1}) \tag{4.4}$$

$Source(T_i)$ ($Target(T_i)$) names the source (target) state of a transition T_i.

[2] C_i is a cluster of events and will be described in more detail in Section 4.1.2

Solving the Word Problem

The language of a PRTA is given by: $L_\mu = \{w \in (\Sigma, \mathbb{N})^+ \mid P_w > \mu\}$, where μ is a probability threshold. P_w describes the probability for a word w, given its state sequence $(q_1, \ldots, q_m)$ and $p(q_i, q_j)$, the probability of a transition from q_i to q_j:

$$P_w = \prod_{i=0}^{m-1} p(q_i, q_{i+1}) \tag{4.5}$$

In the case of a PRTA, $Q = Q_{accept}$ and $\mu = 0$, i.e., there must exist a path leaving from q_0 (with a joint probability greater than zero) that 'consumes' word w. State q_0 is exactly the state that represents $\vec{e_1}$: $\vec{e_1} \in E_0$. For this problem, it is easy to give an algorithm that terminates after a finite number of steps. Algorithm 3 shows the initialization of the problem (finding the

Algorithm 3 ParseWord (PRTA Γ, ArrayList w)

```
$q_0$ = findFirstState(PRTA, w[1])
**if** $q_0$ is not null **then**
  accept = ParseRemainingWord($q_0$, w)
**else**
  return 0
**end if**
return accept
```

Algorithm 4 ParseRemainingWord (State q, ArrayList w)

```
$t$ = TransitionWithDeltaAndTimeLabel($q$, $\delta(w[1], w[2]), time(w[2])$)
**if** $t = \emptyset$ **then**
  return 0
**end if**
$w = w \backslash w[1]$
**if** $|w| == 0$ **then**
  return 1
**end if**
return $p(t)\times$ ParseRemainingWord ($Target(t)$, w)
```

initial state q_0) and then calls the search for a transition sequence. Algorithm 4 describes this search[3] and the stopping criterion. If there is no edge with the required label, the automaton does not accept the word. In contrast, if the 'last' edge is found, the automaton accepts the word and returns the probability of the word.

[3] $|w|$ gives the length of the word w, $w[x]$ the xth event of word w, and $time(w[x])$ the time stamp of $w[x]$.

4.1.2 Induction of a PRTA

In the following, we describe how PRTAs can be learned. The top-level algorithm is shown in Algorithm 5. As input, the algorithm expects a finite set of histories. From the histories, the algorithm first constructs a prefix

Algorithm 5 InducePRTA (Histories H, Parameter $params$)

```
$prefixTree \leftarrow$ createPrefixTree($H$)
$M \leftarrow$ calculateDistanceMatrix($prefixTree$)
$res \leftarrow$ cluster($M$, $params$)
while $res \neq \{\}$ do
    $C \leftarrow$ getNextCluster($res$)
    $prefixTree =$ mergeStatesInPrefixTree($C$)
    $res \leftarrow$ deleteFromResult($C$)
end while
computeQualityMeasure($prefixTree$)
return $prefixTree$
```

tree acceptor (PTA). A PTA is a PRTA in the form of a tree in which exactly one path exists to any state. Each leaf represents one or more instances from the input set. If input histories have the same prefix, then they share the path of this prefix, while the suffix has its own path. When a new history is put in the PTA, there are the following possibilities:

1. No prefix of the history is represented by an existing path in the PTA.
2. There is a path that represents a prefix of the history.
3. There is a path that represents the whole history.

In the first case, a new path starting at the root from the prefix tree is inserted in the PTA. The probabilities of all transitions on the path are set to one, and all delay guards are set to the current time stamp. In the second case (if the PTA shares a prefix with the history), all probabilities on the equivalent path are updated corresponding to the annotated probabilities. Consider a transition from state q_i to q_j on the prefix path of the history and assume that q_i has k other outgoing transitions. For the transition that is covered by the new history, the according probability p' is updated to $p' = \frac{|q_j|+1}{|q_i|+1}$, where $|q|$ denotes the frequency of a state. This ratio actually is the maximum likelihood estimation $b_{ij} = P(transition_{ij}|\vec{e}_k, t_k)$ introduced in section 2.2.1. For all remaining outgoing transitions $t_l (l = 1, \ldots, k)$ of state q_i, the probability p'_l is recomputed by $p'_l = \frac{|q_l|}{|q_i|+1}$. If the time constraint t of the history does not meet the time constraint of the existing transition, the delay guard ϕ'_k is expanded so that it includes the new time constraint: $\phi'_k = [a, b]$, where $a = min(\phi_k, \phi_l)$ and $b = max(\phi_k, \phi_l)$. If the end of the path that reflects the shared prefix is reached, a new path with the

remaining events of the history is appended, following the description of case one. In the third case, all transition probabilities and delay guards are updated as described for the second case, but no additional path is added to the PTA (this procedure is also referred to as determinization). After creating the PTA with all input histories, the goal is to produce a PRTA that is minimal. Minimal means that a minimum number of states should be derived, but reflecting a maximum of information. This condition is heuristically motivated by *Occam's Razor*. The parameter that leads to the minimal model is usually given by the user, in our case, a certain distance threshold between mergeable states (the distance measure is discussed in section 4.2.1).
To obtain a compact model, merges of nodes in the prefix tree are performed. A merge is an operation where two states q_i and q_j are combined into one new state q_k. Because homogeneous states shall be identified, clustering is applied. In general, a merge step is the aggregation of all states belonging to a cluster into one new state with a new profile. A merge combines all profiles $\vec{f_i}$ of the states q_i to be merged into one single profile $\vec{f_k}$ by their weighted mean:

$$\vec{f_k} = \frac{1}{\sum_{q_i \in C_k} |E_i|} \sum_{q_i \in C_k} |E_i| \times \vec{f_i} \tag{4.6}$$

Which states are to be merged is identified via clustering. Therefore, a cluster assignment for each state in the prefix tree must be found.[4] In general, the input for a cluster algorithm is a distance matrix (or a distance function and the instances respectively). However, when constructing an automaton, the input for the clustering is a prefix tree. During the clustering, each state of the prefix tree is handled as an individual instance. The attributes of the instance are the values of the state's histogram (cf. equation 4.1). They can be used as a basis for the computation of distances between states. Let us consider the clustering as a function c (cf. equation 4.7) that maps each state q to a cluster identifier $k \in \mathbb{N}$.

$$c(q) : Q \to \{1, \ldots, k\} \tag{4.7}$$

Then it is possible to evaluate each possible clustering $c_i(q)$ with some quality function G (consider, e.g. the silhouette coefficient or an optimal inter/intra cluster distance). The result of the clustering algorithm is the mapping $c^*(q)$ that maximizes the quality function.

$$c^*(q) = argmax_i \; G(c_i(q)) \tag{4.8}$$

Note that depending on the application domain, the user can decide which distance function, clustering algorithm and quality function is best suited. By using the best function $c^*(q)$, the merge procedure creates for each cluster identifier k one new state in the prefix tree by merging all states q which

[4]In general, the order of PTA construction and clustering is irrelevant, they are just required before the state merging is started.

are mapped to cluster k. Formally, the inverse function $c^{-1}(q)^*$ returns for each cluster identifier k the set of states that are mapped to it. The automaton is created by merging all states q of clusters k one after the other. To preserve consistency, update operations on transitions have to be performed. If two states q_i and q_j are to be merged and there are no transitions t_k where $t_k = \langle q_k, q'_k, T_{L_k}, \phi_k, p_k \rangle$ and $t_l = \langle q_l, q'_l, T_{L_l}, \phi_l, p_l \rangle$ with $q_k = q_l, q'_k = q_i$ and $q'_l = q_j$ (they do not share a predecessor), change t_k to $\langle q_l, q'_k, T_{L_l}, \phi_l, p_l \rangle$ (re-link the transition) and compute the new profile of q_i using equation 4.1. q_j can be deleted from the prefix tree. If there exist two transitions t_k and t_l with $q'_k = q_i \wedge q'_l = q_j \wedge q_k = q_l$ (they share the same start but not end state), the transitions have to be merged additionally. This means that $\langle q_k, q'_k, T_{L_k}, \phi_k, p_k \rangle$ is to be updated to $\langle q_k, q'_k, T_{L_k}, \phi'_k, p_k + p_l \rangle$. $\phi'_k = [a, b]$, where $a = min(\phi_k, \phi_l)$ and $b = max(\phi_k, \phi_l)$.[5] The updated probability p' is again calculated with the counts of the states $p' = \frac{|q'_k|+|q'_l|}{|q_k|} = p_k + p_l$. Note that the labels of the transitions are not updated until the end of the merge procedure. Then, each state holds its set of events and the labels T_{L_k} can be easily computed by calculating the difference between the two sets E_i and E_j. Algorithm 6 shows this procedure.

Algorithm 6 CreateTransitionLabels (q_i, q_j)

```
for each $e_k \in E_i$ do
  for each $e_l \in E_j$ do
    add $\delta(e_k, e_l)$ to labels of transition $t(q_i, q_j)$
  end for
end for
```

One additional property of PRTAs is that a label $\delta_l \in T_{L_k}$ is only present on exactly one outgoing transition of a state q_i. Given that the underlying distance based clustering optimizes the inter and intra cluster distance, respectively, it can be shown that $\nexists\ \delta_l : \delta_l \in T_{L_k} \wedge \delta_l \in T_{L_m}$. The proof works by contradiction: consider the case where there exist three states $q_0 = \langle \{x_1, x_2\}, f_0 \rangle$, $q_1 = \langle \{y_1\}, f_1 \rangle$ and $q_2 = \langle \{y_2\}, f_2 \rangle$, with the transitions $t_1 = \langle q_0, q_1, \delta_l, \phi_1, p_1 \rangle$ and $t_2 = \langle q_0, q_2, \delta_l, \phi_2, p_2 \rangle$, i.e. two transitions, each holding label δ_l. Moreover, let δ_l be defined as $\delta(x_1, y_1)$, $y_1 \notin E_2$ and $y_2 \notin E_1$. Let us further assume, without loss of generality, that δ_l consists of three consecutive blocks, e.g., $\delta_l = (+1 + 1 - 1 - 1000)$. Let k be the first position of the 0-block at the end. Following the assumption that δ_l exists on both transitions, we can specify constraints for x_2 and y_2. First, $x_{1_j} := x_{2_j}$ and $y_{1_j} := y_{2_j}$ for all $j < k$, because there is only one possibility for $y_j - x_j$ to be equal to -1 (and 1) in a binary setting. Second, $y_{2_j} := x_{2_j}$ for $j \geq k$, because otherwise δ_{lj} is not 0. Additionally, there must exists at

[5]In other words, we conduct the least general generalization on the time intervals to be merged.

least one $x_{1_j} \neq x_{2_j}$ to ensure that $y_2 \neq y_1$. Otherwise, y_1 and y_2 are equal and thus clustered together, so t_2 cannot exist. With these constraints: (1) $x_{1j} = x_{2j}$, $y_{1j} = y_{2j}$ $(i < k)$, (2) $y_{2_j} := x_{2_j}$ $(i \geq k)$ and (3) $y_1 \neq y_2$ the distance $d(x_1, y_1)$ reveals to be equal to $d(x_2, y_2)$, because the distance for parts $i \leq k$ is 0 (2) and the remaining distances are equal (1). As y_1 and y_2 must reside in different clusters, the clustering then returns a solution where pairs of objects with the same distance are clustered together once, and not a second time. This is inconsistent and also shows that the inter and intra cluster distance in this clustering cannot be optimal. This leads to the conclusion that for all suitable distance based clusterings there is no δ_l that occurs on more than one outgoing transition of one state q_0. Finally, this conclusion even allows to check each learned automaton whether the intra cluster distances are above a threshold to ensure that each δ_l occurs only once.

The delay guard generalization is motivated by the use of positive instances only. Remember that all input instances shall be accepted by the automaton, indicating that they are positive. Furthermore, we assume that when no continuous timeframe is present in the data, it is not because it does not exist, but because of lack of data.[6] However, the profiles of the states q_i and q_j have to be updated with their weighted mean. Transition t_l is deleted. The same operations have to be applied on all outgoing transitions of states q_i and q_j. After all merges are conducted, a quality measure of the clustering is computed. Like in every clustering problem, the proportion of inter- and intra-cluster distances is of interest. The silhouette coefficient (SC) [47] evaluates this proportion independent of the number of resulting clusters and is a measure for the homogeneity of the states of the automaton. The SC of a state q_j representing a cluster C is calculated in the following way:

$$SC_C = \sum_{o \,\in\, C} \frac{b(o) - a(o)}{max\,\{a(o), b(o)\}} \tag{4.9}$$

$a(o)$ is the distance to the own cluster center, while $b(o)$ denotes the distance to the second next cluster center. The SC for the automaton is computed by averaging the SC for each state. It always holds that $-1 \leq SC \leq 1$. Good results are expected above an SC of 0.5. However, a high SC does not necessarily reflect the best clustering, since $SC_C = 1$ for all $|C| = 1$ or $|C| = X$ where X is the number of all instances. Generally, SC_C increases with smaller clusters, because $SC_C = 1$ for all states C that consist of one example only.

The clustering method can be different for each use case. We decided to cluster with a divisive hierarchical cluster algorithm because it enables us to

[6]This assumption can be made in the medical application domain presented below. Here data is not recorded in regular time steps but whenever people go to the physician. So, there are gaps in the data recording process, which are taken into account by the above assumption.

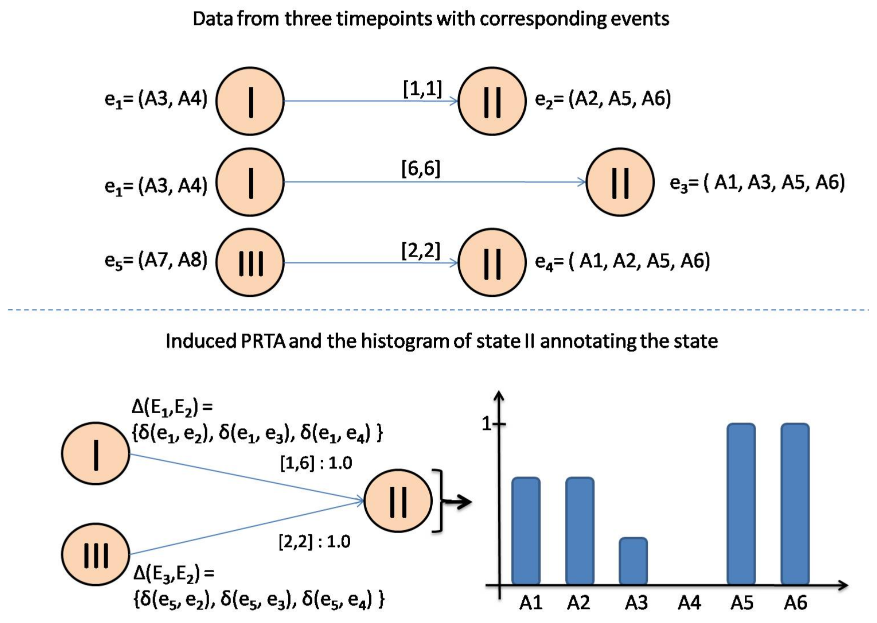

Data from three timepoints with corresponding events
e1= (A3, A4)
I
[1,1]
II
e2= (A2, A5, A6)
e1= (A3, A4)
I
[6,6]
II
e3= (A1, A3, A5, A6)
e5= (A7, A8)
III
[2,2]
II
e4= (A1, A2, A5, A6)
Induced PRTA and the histogram of state II annotating the state
Δ(E1,E2) = {δ(e1, e2), δ(e1, e3), δ(e1, e4) }
[1,6] : 1.0
[2,2] : 1.0
Δ(E3,E2) = {δ(e5, e2), δ(e5, e3), δ(e5, e4) }
1
A1 A2 A3 A4 A5 A6

Figure 4.2: Example creation of a PRTA

define a distance threshold that has to be fulfilled by the clustering. Furthermore, correlations between states can be investigated, and upper and lower distance thresholds can be set according to domain-specific constraints. In contrast, it will not be possible to tell a priori how many states are to be expected. This is why k-means like clustering methods are not appropriate in our scenario. Diana [47] is a divisive hierarchical clustering algorithm which computes a final dendrogram that shows how 'related' the states are.[7] With this dendrogram and a specific distance constraint, the user can create a suitable clustering. The dendrogram is cut according to the distance constraint, and all objects that are still connected form a cluster. Nevertheless, one critical point for any clustering is to choose an appropriate distance measure. As this is domain-dependent, it will be discussed in the section on experimental results.

In Figure 4.2, a merge step during the construction of a PRTA is illustrated. We see here that the states are annotated with a profile and that transitions consist of all observed possibilities to end in a state, a path probability and a delay guard. The upper part of the figure displays three sample histories in a prefix tree before the merging starts. For simplicity, the root of the prefix tree is not shown. The number in the states indicates to which cluster the states belong to. Two of them are in cluster one, three are in cluster two and only one state in cluster three. On all transitions only one event

[7]Preliminary experiments with a variety of other clustering algorithms like K-medoids [47], DBScan [5], EM, and Farthest First did not produce any usable results.

that leads from the left to the right state and the delay guard is displayed. A delay guard of [1,1] means that the event followed exactly one time step after the first event. The annotation of $A2$, $A5$, $A6$ on the transitions means that these attributes were observed after this state. The lower part of the figure shows the automaton after the merging step (without the root). The transitions and delay guards are updated following the rules. The number behind the colon reflects the probability of this path. In this case, they are always equal to one because there exists no splitting transition.

4.1.3 Predicting with an Automaton

In this section, we explain how such an automaton can be used to make predictions. With a PRTA, we cannot only map processes that are reflected in the data but also make predictions about how the next state of an instance will be. Of course, it is also possible to predict series of subsequent states. The task of the prediction for a new instance can be formalized as follows: Given an instance x denoted by its feature vector $\vec{f_x} = (f_1, f_2, \dots, f_n)$, we want to identify the profile $\vec{f^*} = (f_1^*, f_2^*, \dots, f_n^*)$ it will develop after l time steps. A prediction for an instance is done by first identifying the state in the PRTA that is most similar to the given event distribution of the instance. This is the start state q_{start} for the prediction:

$$q_{start} = argmin_{q_i} d(\vec{f_x}, \vec{f_i}) \tag{4.10}$$

Distance function d will be introduced in a subsequent section. Given an arbitrary state q, let $q_1, \dots, q_k$ denote the states with incoming transitions from q, and $p_1, \dots, p_k$ be the probabilities on those transitions. Moreover, let $[t_{1,1}, t_{1,2}]$ to $[t_{k,1}, t_{k,2}]$ denote the delay guards for the transitions. Then the predicted profile given l time steps starting with state q is defined as:

$$\begin{aligned} f^*(q,l) = & \sum_{t_{i,2} > l} p_i \times f_q + \\ & \sum_{t_{i,2} \leq l} p_i \times f^*(q_i, l - t_{i,2}) \end{aligned}$$

The next state is predicted according to the transition probabilities and their delay guards. The first summand represents the case where state q is not left, because it consumes all the ‘remaining’ time. Thus, no other following state is considered for the prediction. The second sum represents the case where the state *has to be left*. In the latter case, this means that the predicted profile of the next state is used. If q does not have any outgoing transition, then $f^*(q, d) = f_q$, i.e. the profile on the state itself. To obtain a prediction for the test instance, we apply f^* to q_{start}. Note that we make the assumption that the automaton stays maximally long in a state, i.e. the transition is made as late as possible. Moreover, to leave a state, the delay guard has to meet the time constraint l: $l > [t_{k,1}, t_{k,2}]$. Then, the remaining time l' for the next steps is reduced by the maximum of the delay guard $l' = l - t_{i,2}$. These constraints lead to a definite prediction of the profile: There is only one possible solution for the prediction. Another assumption would be that a transition is made as early as possible. Again, one definite prediction is achieved. However, an arbitrary time consumption of each state could be desired as well. In this case, the prediction would be dependent on all (valid) possible time consumptions of each state. Accordingly, the prediction is the averaged profile of all resulting predictions. Using the later approach may result in an exponential number of possible results and a highly blurred predicted profile. That is why a definite prediction was chosen in this chapter. However, the choice of a prediction constraint is likely to be highly domain-dependent. By calculating joint profiles, we can predict the change of attributes depending only on the instance's attribute setting. However, if one is interested in the development of a certain state, the automaton gives probabilities for future states. This is a main advantage of the automaton, because many comparable algorithms only give one prediction. Consider a system that has alternatives for each state (e.g., medical therapy options and outcomes or biochemical pathways), meaning that a certain percentage of instances will take either the one or the other path. So, a distribution of resulting states is the desired output. The automaton aims at modeling such alternatives and their corresponding probabilities.

4.2 Experimental Results

In this section, we present the results of two experiments to test the algorithm. First, a proof of concept is conducted and subsequently an application to real-world medical data is presented to investigate the descriptive and predictive power. Afterwards, a quantitative analysis of the algorithm is presented. Last, a comparison to Multi-Output HMMs is presented. To adapt the approach to a given application, we now have to be more specific about the distance measure used for clustering. Therefore, we first elaborate on such a measure for our medical application.

4.2.1 Distance Measure for Medical Applications

In general, every distance measure can be applied to the learning algorithm but they should be adjusted to the application domain of interest. In the field of medicine, where an event reflects morbidities, the presence of a disease is more important than its absence, because the development of diseases shall be explored. For this purpose, the design of a distance measure shall reflect that, when an attribute is present, it influences the distance stronger than its absence. If it is absent, it means that the person does not suffer from a certain disease. This is further underlined by the fact that most diseases are comparatively rare. Most people of a population are healthy (have no chronic conditions). This is why the presence of diseases should receive more weight than the absence of diseases. To design a distance measure with this property, assume two events $\vec{e}_i$ and $\vec{e}_j$. Similarity should increase when the intersection of attributes that are equal to one in $\vec{e}_i$ with the set of attributes that are equal to one in $\vec{e}_j$ increases. If $|\vec{e}_i|$ grows, but $|\vec{e}_i \cap \vec{e}_j|$ stays the same, the similarity should decrease, because the proportion of the intersection is lower. As $|\vec{e}_i \cap \vec{e}_i|$ approaches $|\vec{e}_i|$ or $|\vec{e}_j|$, the similarity should also grow. All these requirements are fulfilled by the Tanimoto coefficient. Nevertheless, if the proportion of the intersection of $\vec{e}_i$ and $\vec{e}_j$ is equal to the intersection of two other states $\vec{e}_k$ and $\vec{e}_l$, but $|\vec{e}_i| < |\vec{e}_k|$ and $|\vec{e}_j| < |\vec{e}_l|$, the similarity of $\vec{e}_k$ and $\vec{e}_l$ should be lower because there are more attributes set than in the other vector. They are just more morbid than the compared $\vec{e}_i$ and $\vec{e}_j$. This property is not met by the Tanimoto distance and therefore we have to introduce an additional term that captures the size of $|\vec{e}_i|$. The Hamming distance is a measure that compares unequal attributes and adjusts a lower similarity in this case. A distance function that meets all the discussed requirements can be expressed as a combination of the two distances Tanimoto and Hamming. Tanimoto captures the relative distance (magnitude of intersection) and Hamming reflects the absolute distance (number of equal attributes). Introducing a parameter $\alpha \in [0, 1]$, the influence of Tanimoto and Hamming on the final distance can be controlled. The resulting distance function is:

$$d(x, y) = \alpha d_{rel}(x, y) + (1 - \alpha) d_{abs}(x, y). \tag{4.11}$$

If α is one, we obtain a Tanimoto distance, if it is zero, we obtain the Hamming distance. We chose α to be equal to 0.75, so we will obtain a high influence of Tanimoto (equation 4.12) and a rather small one of Hamming (equation 4.13):

$$d_{rel}(x, y) = 1 - \frac{n_{11}}{n_{x1} + n_{y1} - n_{11}} \tag{4.12}$$

$$d_{abs}(x, y) = \frac{n_{01} + n_{10}}{n} \tag{4.13}$$

In equations 4.12 and 4.13, n_{11} is the number of attributes set in both events, n_{x1} (resp. n_{y1}) is the number of events set in event x (resp. y), and n_{01} (resp. n_{10}) is the number of ones set in x but not in y (resp. set in y but not in x). The resulting distances lie between zero and one. Another adaptation to the domain of the algorithm is that events that do not have any attribute equal to 1 are left out of the history. We assume that an instance that already had an event with $|\vec{e}| > 0$ will always have at least an event with $|\vec{e}| > 0$ afterwards. Otherwise, this indicates an error in the data.

4.2.2 Results

This section focuses on the results of the algorithm with Diana clustering on synthetic and real-world data. The a priori definition of cut-off thresholds and an evaluation of the predictive accuracy is presented. At first, a proof of concept is demonstrated. In this section the accuracy of the model in terms of the number of states and transitions as well as their annotation is examined. Experiments on the real-world data sets give results of how meaningful the automaton is from the application point of view, e.g. if the structure reflects the known disease progressions and if it can predict future states correctly.

4.2.3 Proof of Concept

As described in Section 4.1 the induction of a PRTA is based on clustering. It therefore needs a distance constraint, indicating how similar events must be to form a cluster. This constraint is the cut-off in the dendrogram that is returned by Diana clustering. As the correct structure of the automaton is known in this experimental setting, a lower and upper limit for the cut-off can be set. A cut-off of 0.3 would lead to a merge of states 5 and 9, which is not desired. To clarify this upper limit constraint, remember state 5 ($[2, 5, 6, 7, 8, 9]$) and state 9 ($[2, 5, 6, 7]$). Their distance is composed of the Tanimoto part (cf. equ. 4.12) which is equal to $\frac{1}{3}$ and the Hamming contribution that is 0.2. Their weighted sum (according equation 4.11) is 0.3. To follow the requirement that they must not be merged, the cut-off must be lower than 0.3. Additionally, it should also be greater than 0.21 to merge state 2 ($[1, 3, 4]$) with a (possible) state that holds one additional attribute (e.g., $[1, 2, 3, 4]$). Thus a good automaton is expected with a cutoff between 0.21 and 0.3. When exploring the resulting number of clusters and the *SC* for the clustering in the interval between 0.26 and 0.3, a plateau of 10 states and an *SC* of 0.97 arises. This indicates that a stable automaton can be found in this interval. For higher cut-offs, the number of clusters and the *SC* is dropping sharply. Lower cut-offs lead to more clusters that are purer but do not incorporate exceptions in the states. The resulting automaton for a cut-off of 0.3 is shown in Figure 4.3. It matches the true automaton structure perfectly. There is only one exception in the structure. There exists a transition from state 2 to state 7 that is not present in the original automaton. This is due to the data. Remember that random errors were introduced in the data to check whether the automaton is able to find exceptions in states. This causes state 2 to be followed by event [3,4,5] instead of [3,5] in one sequence. So event [3,4,5] matches the profile of state 7 ([3,4,5]) perfectly and was merged with state 7 instead of state 3. Additionally, the *SC* shows that the states are homogeneous. The annotation of the states in the PRTA shows that events 'with errors' were actually included in the state. There are profiles that exhibit low frequencies of 'wrong attributes'. However, this finding shows that the algorithm works correctly because it has to group *similar* events. This makes the automaton less prone to data errors and produces stable results. Moreover, the user can identify the main

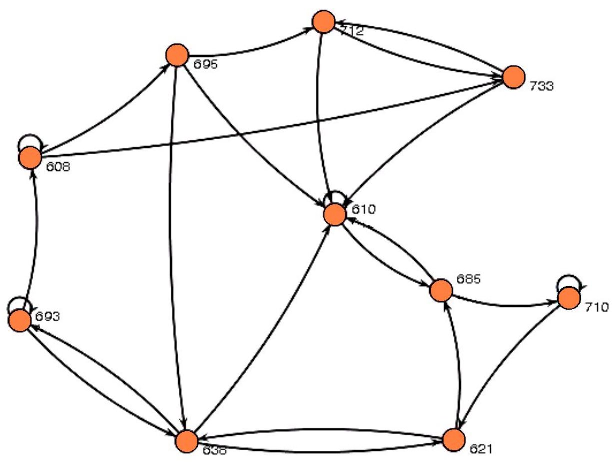


Figure 4.3: Results of algorithm on synthetic dataset. The layout is analogous to Figure 4.2. Each node is labeled with an internal number that is computed by our implementation.

properties of a state and check how often an attribute occurs. The main properties of a state are those attributes that occur commonly, while exceptions are all attributes that have a low frequency in a profile. Exceptions might be errors in the data but may also reflect rare cases in a system progression. Finally, the probabilities on the transitions match those of the original automaton quite well. We encounter at most a deviation of 0.15 in the annotation of probabilities, where the largest deviations are apparent for small transition probabilities. This experiment shows that the algorithm can rediscover the correct structure of an automaton.
To quantify the stability of the algorithm (not illustrated), we repeated this experiment 1000 times on different synthetic datasets that were obtained from the predefined automaton. The correct number of states of the automaton was induced in 31,8% of the cases, while in 50% of all runs the automaton's number of states resided between 10 and 12. Altogether, the number of states of the final automaton varies between 9 and 14 states. The Euclidean distance of the induced states to their closest original lies between 0 and 0.25, which indicates that the induced profiles are very similar to the true ones. The states that were identified worst are states 9 and 7, because they are very similar to states 5 and 3, respectively. This also led to a bad identification of the transitions of these states (only in about 50% of the cases), while the remaining transitions were almost always correctly identified. These numbers indicate that the algorithm finds automata that are correct or very close to the original one.

4.2.4 Extraction of an Automaton on Real World Data

Results on diagnostic data To test our algorithm on real-world data, an automaton was built on 1000 samples of the real-world diagnostic dataset. Again, a priori assumptions are needed to define an upper and lower bound for the clustering cut-off. Here they were based on medical assumptions. States with $|\vec{e}| = 1$ (only one attribute is equal to one) shall not be merged with states where $|\vec{e}| = 2$. The motivation for this constraint is that individuals with one disease shall be separated from those that have already acquired two or more diseases and are thus multimorbid. This distinction can show the beginning and the progression of diseases and will help to interpret results. Furthermore, the cut-off should be set greater than zero. This constraint enables the automaton to merge states that are similar, but not the same (due to errors in the recording of the data). This constraint cannot be fixed any further because the profiles of the resulting states are not known in advance. Thus, meaningful automata are expected with a cut-off ≤ 0.4 . We computed the automaton for several cut-offs and extracted its corresponding silhouette coefficient (SC). As expected, the silhouette coefficient drops as the cut-off increases. The higher the cut-off, the higher is the variance in the clusters. This leads to an approximation of inter/intra cluster distances and thus a reduction of the SC. However, it is interesting to notice that at a cut-off of 0.38 there is a sharp drop of the SC of about approximately 18% and a drop of the number of clusters (20%). Therefore, one can assume that a meaningful cut-off threshold is reached and thus more states are merged. The same behavior can be observed when we apply the algorithm to only 100 instances. Therefore the cut-off 0.38 was chosen to create the PRTA. The resulting automaton contains 518 states and has an SC of 0.83.
The first quality criterion of the automaton is its potential to reflect typical medical knowledge with its structure. The resulting automaton contains 19 states that are highly connected to other states (also called hubs, cf. section 4.2.5). They are mostly states with only one or two DGs (e.g., hypertension and lung disease [e.g., COPD or asthma]). States like this are very frequent in the histories, so it is evident why those states are connected to many others. Besides hubs, loops and paths are created. A loop is a path in the automaton where start and end states are the same. The inspected loops resemble individuals with multiple diseases (multimorbid), which do not change their general morbidities (e.g., diabetes, heart failure, hypertension), but differ in their comorbidities (e.g., urologic disorders). Moreover, progression of diseases as well as improvements that reflect typical medical 'careers' are present in the automaton. We conclude that the automaton shows meaningful medical patterns.
A second quality parameter is the predictive power of the automaton. To evaluate the predictive power of the algorithm, the dataset was split into a training (75%) and a test (25%) set. The split was provided as part of an industrial case study. For each of the k histories with n events of the test

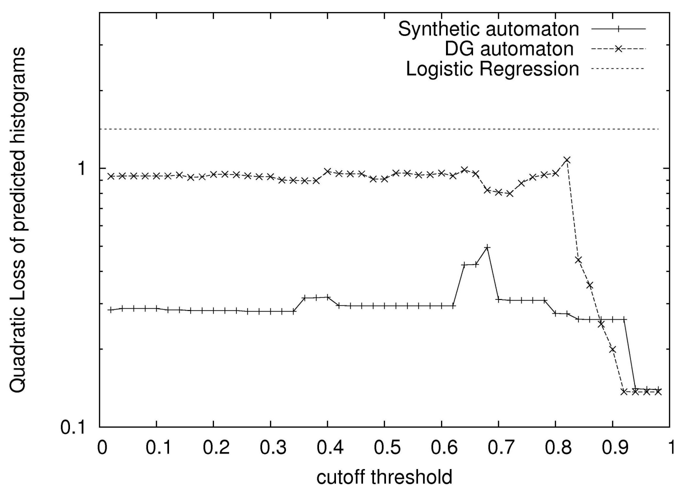


Figure 4.4: Loss of predictions depending on cut-off value

sample, $n - 1$ predictions are executed. First, the history's start state is taken to predict its second state. Then the second state is chosen for prediction and is evaluated by the third state. This is repeated until the end of the history and results in 360 one step predictions. To predict the profile for all examples starting in state q, the joint predicted profile is calculated. Subsequently, this joint predicted profile is compared to the true joint profile of these instances by computing the quadratic loss. For the whole automaton, the average loss is computed by the weighted average of the quadratic loss of all states. Each loss is weighted by the number of instances that start in state q. Using other scoring functions for the evaluation of predicted probabilities, like Kullback Leibler or information loss, is not appropriate because they rely on the log function which is not defined for zero.
We compared the results to the well-established method of logistic regression, which is considered a standard in this application domain. Here, the prediction of events is achieved by applying one logistic regression for each attribute (DG), which results in 28 logistic regression models. The input for the logistic regression is the known state profile of the preceding state which is a feature vector. The logistic regression will lead to one prediction for each attribute of the output vector, which together form the profile of the prediction. Again, the resulting profile is compared to the true profile for groups of the same input vector. For both methods, histories that only contain one state were excluded from prediction, because no validation sample is present. Note that in both methods the Markov assumption holds, both methods predict the next state depending only on the current feature distribution. The automaton's loss lies between 0.89 and 0.97 for similarity cut-offs up to 0.4 (cf. Figure 4.4). Logistic regression achieves an average loss of 1.42 between the true and predicted profiles. This in-

state	A46	A56	A58	A19	A80	A91	A83	A84	A86	A92	A131
1259	**A46**	A56		**A19**	**A80**	**A91**					
1256	**A46**	A56	A58	**A19**	A80	**A91**					
934	**A46**		A58	**A19**	**A80**	**A91**	A83	**A84**	A86	**A92**	A131
932				**A19**	**A80**	**A91**		A84	**A86**	A92	**A131**
835				**A19**	**A80**	**A91**	A83	A84	A86	**A92**	**A131**
929				**A19**	**A80**	**A91**		**A84**		A92	A131

Table 4.1: Example of the development of multimorbids: The features are as follows: A46 coagulation disorder, A56 psychosis, A58 depression, A19 Diabetes, A80 heart failure, A91 hypertension, A83 heart attack, A84 coronary heart disease, A86 other heart problems, A92 arrhytmia, A131 renal failure

dicates that the automaton may find better predictions than the logistic regression approach. Regarding the predictive power with higher cut-offs, the loss slightly increases but when the cut-off exceeds 0.8, the average loss quickly decreases again. This is because there are so many different states in the clusters, that groups resembling the average population are created. Of course, they will also get predictions for the average population, which is better in the predictive power, but will not enable predictions on a person but only on a group level. However, predicting the profiles for single persons is a main use case, and thus higher losses must be accepted. Compared to the synthetic automaton, the real-world data automaton is worse in its predictive power because it has more attributes and a higher variability in the states and histories. The synthetic automaton has quite the same predictive ratio at each cut-off because of the small variability in the data. So, smaller predictive errors can be observed for the synthetic automaton. Initial experiments on the prediction accuracy showed that for smaller testsets (e.g. in a 10fold cross validation setting) the average prediction accuracy decreases. This may be caused by the fact that neither in the true test set nor in the prediction the average population profile is computed but only a fraction of all possible profiles are considered. Thus the variation is high and the difference between the true and predicted profile may increase. However, a detailed exploration of the predictive power remains for future work.

Investigating patterns of a (real-world) diabetes automaton In this paragraph we illustrate the medical knowledge represented by an automaton. We focus on a qualitative evaluation, not on a quantitative at this point. An automaton for patients suffering from diabetes was created on 100 randomly sampled patients. The data was available for 4 years and states were derived on a half-year basis. One state comprises data from quarter two and three of a year or quarter four and the subsequent first quarter, for seasonal reasons. Only frequent (occurring in more than 5% of all patients)

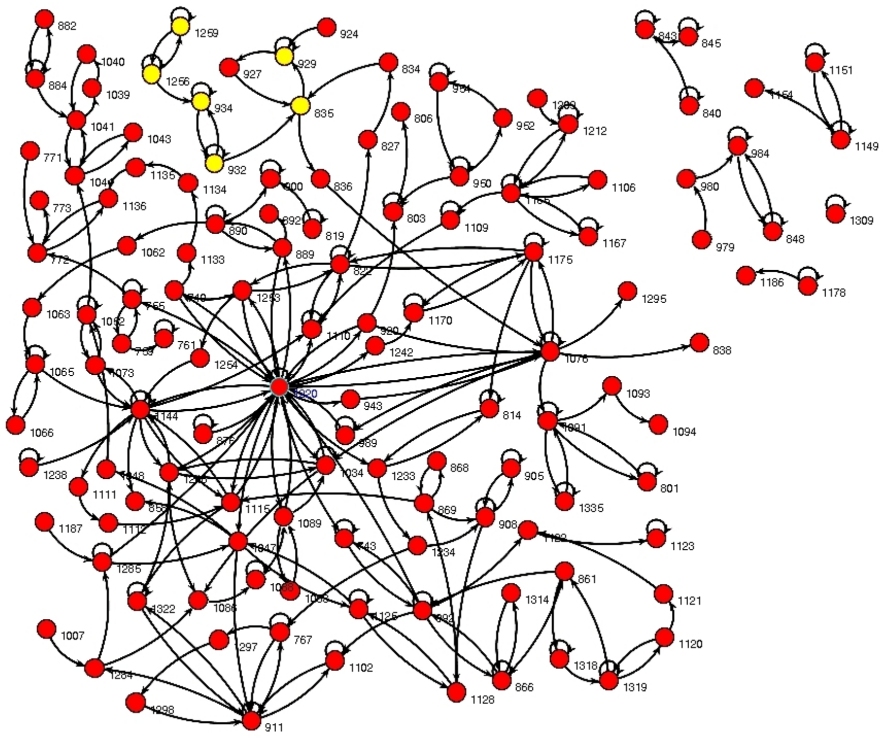


Figure 4.5: Results of a diabetes population

attributes were considered. The resulting automaton is displayed in Figure 4.5. There are five patients, which cover states that no other patient has. That is why they form separate small components. They are shown at the upper right part of the figure. Overall, the automaton contains 115 states. The *SC* of this automaton is 0.62. It is interesting that there are many states that form cycles as well as self-loops. That means that patients alternate between those states. This also reflects the normal coding behavior of physicians, which often diagnose the same diseases in successive quarters. However, we can now choose a state in the automaton, analyze its properties and check which development it can undergo. Because this automaton is built only for patients with diabetes, most states cover a diabetes attribute. But this is not true for all states. Some of them are pre-diabetic states or are produced from patients that were once misclassified as diabetic patients but did not again receive this diagnosis. Of course, it is most interesting if typical disease progressions can be found in the automaton. Therefore, we picked a path of the automaton. It shows possible developments of the health status when having diabetes. The attributes in the states belonging to this path are presented in Table 4.1. The headline defines all attributes that show up in the path. If an attribute is present in the state, it occurs

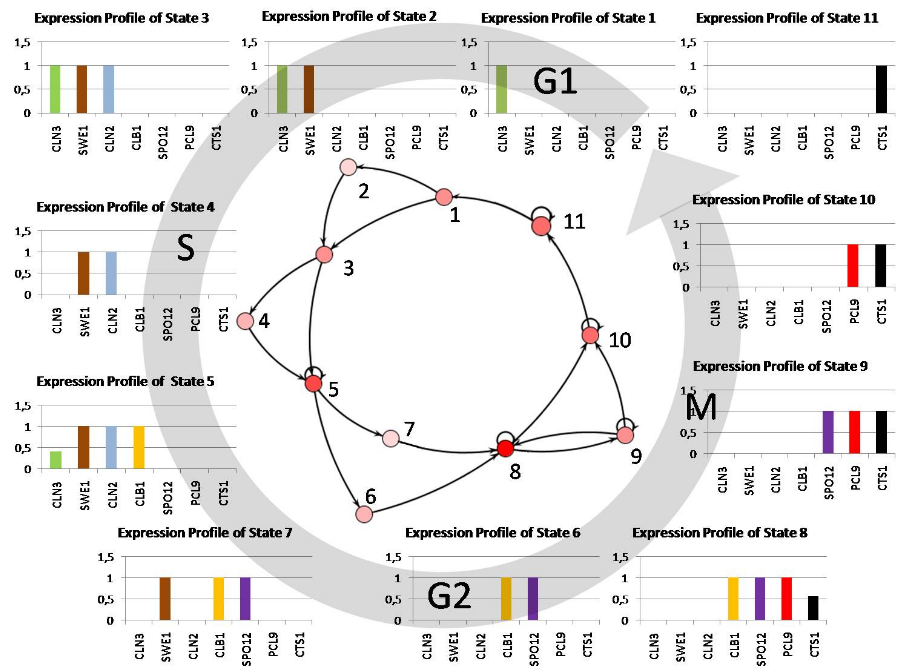

Expression Profile of State 3
Expression Profile of State 2
Expression Profile of State 1
Expression Profile of State 11
Expression Profile of State 4
Expression Profile of State 10
Expression Profile of State 5
Expression Profile of State 9
Expression Profile of State 7
Expression Profile of State 6
Expression Profile of State 8
CLN3
SWE1
CLN2
CLB1
SPO12
PCL9
CTS1
G1
S
G2
M
1
2
3
4
5
6
7
8
9
10
11

Figure 4.6: Automaton for the yeast cell data: In the center the automaton is displayed; for each state the corresponding profile is given. The height of the bar shows the probability of the events and the bars correspond to the genes in Fig. 4.7.

in the cells. Bold attributes indicate that 100% of the instances clustered in this node hold this attribute. The path highlighted in the upper left part of Figure 4.5 begins at node 1259 and ends at node 929. It shows how the patients' diseases shift from a diabetes and rather mild heart disorder to severe heart problems along with diabetes. Attribute 46 reflects a coagulation dysfunction which is a risk factor for myocardial infarction and coronary heart disease (CHD A84). As time progresses, a heart attack may occur (A83), but in every case it leads to CHD and further heart problems (A86). Additionally, renal problems (A131) are present when coagulation dysfunction persists. However, as more diseases appear, the coagulation dysfunction is no longer present. This could have two reasons. First, it is not coded anymore, because it is not severe enough. Second, because it is a strong risk factor it will be treated and thus does not occur anymore. Regarding this path and the overall structure, we can conclude that the algorithm finds meaningful patterns that reflect current medical knowledge. When applying the algorithm to other data, equivalent medical patterns were found as well.

Results yeast metabolism data In this paragraph, another application based on yeast cell data is presented. It will illustrate how automata can be used to infer knowledge about biological processes. In this experiment, gene expression data of the budding yeast is modeled by an automaton, which is displayed in Figure 4.6. It consists of 11 states and 21 transitions with an SC of 0.87. The cutoff was set to 0.224. It was derived by the fact that due to the quite low number of attributes, states shall be very pure (and distinct) and thus merges shall be allowed only when there are at least four attributes in a state, which is more than half of all attributes. Inspecting the structure of the automaton, the most apparent fact is that it is a cycle. Thus, the automaton presents a periodic pattern that the organism passes through, which can also be discovered in the original data. This is known to be the cell cycle of the budding yeast. In a well nourished culture, yeast grows and divides in a constant manner. The process is divided into four coarse steps: the G1, S, G2 and M phase. During the G1 phase the cell is growing, and then the DNA is duplicated in the S phase. Subsequently, the cell prepares for division in the G2 phase and finally divides in the M phase. In each phase specific genes are expressed that control the processes of the cell. Investigating the expression profiles of each state in the automaton, specific expression profiles can be found. Comparing these profiles to known activation gene patterns (cf. Figure 4.7 upper part), the stages of the cell cycle can be annotated to the states [74]. The profiles show that each gene is expressed at a specific stage and that genes also have a specific temporal ordering. Most importantly, state 1 represents the start of a cell cycle and state 11 is its end.

But not only the structure of observed biological processes, also temporal characteristics can be reflected by the automaton. Usually, cells rest some time after a division, which leads to a lag before the next cell cycle starts. This prolonged resting phase is captured by the delay guard on the transition from state 11 to state 1. It does not only allow for immediate transitions but also a delayed transition after two time steps. Moreover, stages lasting more than one time step (e.g., G2) can be visited more than once, represented by a self loop. This means that the cell stays in a state for a while. Additionally the automaton captures the possibility that expression peaks/drops may occur faster or slower. These alternatives are illustrated by, e.g. the transition from state 5 to 8 via state 6 or 7 respectively. While state 7 still holds gene SWE1, it is not present in state 6, which shows that here the expression rate has dropped faster than in the former case. Comparing this to the original data in Figure 4.7, the quick drop of gene SWE1 can be seen at timeframe 11 to 13, while a slower drop occurs in timeframe 22 to 26. Such an alternative is also presented in the transition from state 1 to state 3 (traversing state 2 potentially). The direct connection shows that genes SWE1 and CLN2 are expressed at the same time, while state 2 shows that gene SWE1 comes first. However, the probabilities on the transitions indi-

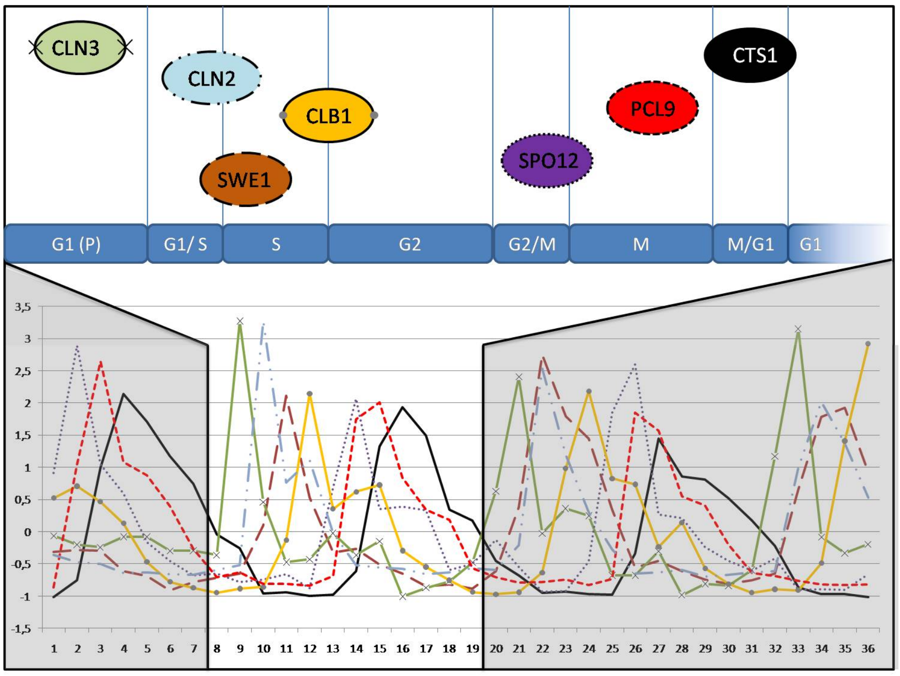
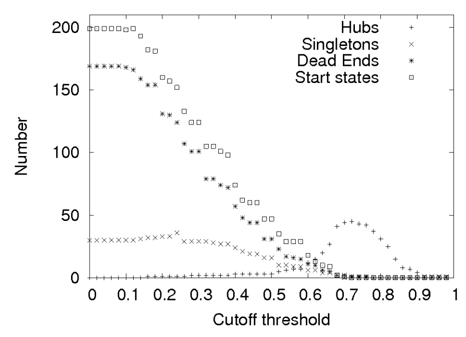
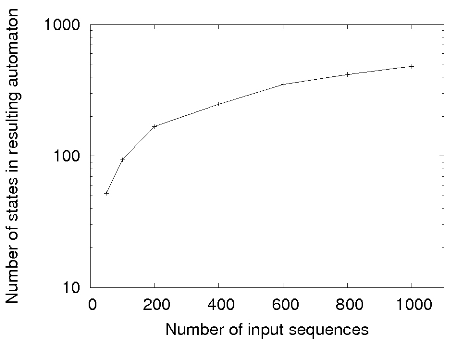


Figure 4.7: Expression profiles of seven selected genes of the yeast data set: The lower part shows the Z-score normalized raw values of the genes. The color (and style respectively) of the lines matches the gene's name in the upper part (e.g., the red (dashed) line corresponds to gene PCL9, the line style in the lower part appears again at the border line style of the shapes). In the upper part, the expression peaks of all genes for one timeframe are displayed. The timeframe spans from time point 8 to time point 20. Each gene's expression peak occurs three times.

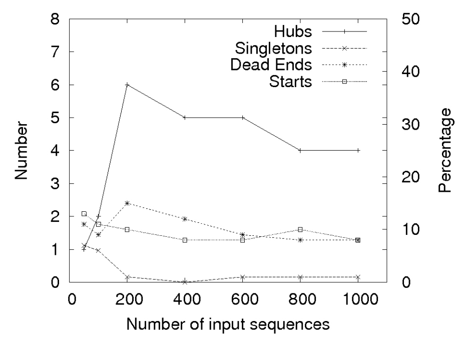
cate that the first one is more frequent and thus to be expected. Using such information, this experiment shows that the automaton can uncover biological processes by their recorded characteristics. Even more, the automaton provides the possibility of combining data from different cell cultures that are not synchronized. Until now, cells have to be in the same state before the experiment can take place. Using automata, it would be possible to combine data from different phases because equal expression profiles will be mapped to the same state. Nevertheless, it has to be noted that results may of course differ, depending on the data quality and discretization method. However, the method shows even more potential if the time resolution and recording length is further improved. With a higher time resolution, the states can be further explored (e.g., by substates) and an extended recording could show rare cases and alternative transitions.



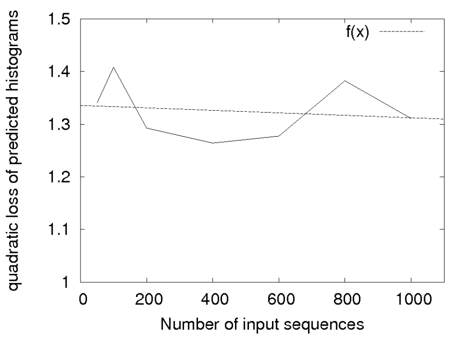
(a) Structure for different cutoffs



(b) Number of states for different input data sizes



(c) Structure for different input data sizes



(d) Resulting loss with different input data sizes

Figure 4.8: Structural dependencies for different cutoffs and numbers of input data

4.2.5 Empirical Evaluation

In this section, some empirical properties of the algorithm are explored. The behavior depending on its main parameters, the number of histories (input sequences) and cut-off values, are studied in detail. Additionally, a runtime analysis will be presented. First, the correlation between the number of states (and other structural features) and increasing cut-offs of the clustering will be shown. Second, an investigation of those features depending on the number of input sequences is presented.

Cut-off Dependencies

To evaluate the structural features of the resulting automata, the number of *hubs*, *singletons*, *dead ends* and *start states* relating to the cut-off is presented (cf. Fig. 4.8a). We define a hub as a state that has transitions to at least 10% of the remaining states in the automaton. *Singletons* are states that do not have a transition to another node. *Start* and *dead end* states only have outgoing and incoming edges respectively. With an increasing cut-off, similar states are clustered and therefore the number of hubs increases as well. This is because the number of states drops while the number of transitions stays constant. However, for the highest cut-offs the number of *hubs* drops again, because even *hubs* are combined into fewer states. *Singletons* become fewer with increasing cut-off. Again, they are merged with other states that have a connection to other states and thus they are no singletons no more. The same is true for *dead end* states. The number of states in the final automaton drops linearily (after a cut-off of 0.1) as the cut-off of the hierarchical clustering increases (not shown). For cutoff values greater than 0.9, only one state appears in the resulting automaton. The reason for this is that with a higher cut-off, states that are less similar will be clustered and thus form fewer states.

Dependency on the Number of Input Sequences

Figures 4.8b to 4.8d show how the number of input sequences affects the resulting automaton. In Figure 4.8b, the number of final states in the resulting automaton is shown. It is interesting to note that the number of states does not grow linearly but seems to converge to an upper limit. This may be explained by rare disease characteristics which are only present after a certain sample size. For low sample sizes, many frequent patterns are discovered and so the number of states increases rapidly. But subsequently, mostly rare cases are incorporated into the model, which leads to a slower growth rate of the automaton. However, the final number of states should then not significantly increase further. Regarding the number of *hubs*, Figure 4.8c shows that with an increasing number of input sequences the number of *hubs* (left axis) that appear in the final automaton is quite stable. Keeping in mind that a hub occurs when a state is connected to more than 10% of the remaining states, this indicates that a certain number of disease states frequently appear in the histories. This can be true for, e.g. a 'hypertension' diagnosis or a metabolic problem, which are both frequent, alone and in combination with other diseases. The frequency of *singletons*, *dead ends* and *start* states (right axis) drops with an increasing number of input sequences. That indicates that even rare cases are observed more frequently and therefore occur in more sequences, which leads to a higher transition rate in these states. Finally, the quadratic loss of the predicted histograms with a growing number of input data is evaluated. The results of these experiments are illustrated in Figure 4.8d. The calculated interpolation of the data points indicates that the overall loss of the predicted histograms decreases with more input data. This can be explained by the inclusion of rarer cases which only appear in larger datasets. Thus, events that need rare information can be predicted more accurately.

IS	50	100	200	400	600	800	1000
RT (s)	4	3	6	23	60	120	240

Table 4.2: Runtime (RT) depending on number of input sequences (IS)

Runtime Evaluation

We determined the runtime for a varying number of input sequences. All experiments were conducted on a 1.7GHz machine with 2GB RAM where the PRTA is implemented in Java with an interface to R and Weka to use the clustering algorithms. As Table 4.2 shows, PRTAs can be created within acceptable time frames. The runtime of the algorithm is mainly impacted by the clustering method. So, the decision for one or the other clustering method can result in significantly higher or lower running times. As Diana clustering is used in this approach, the runtime is bounded by $O(n^2)$. This is also reflected in Table 4.2, where the runtime is illustrated.

4.2.6 Comparison with an Multi-Output HMM

Although HMMs do not provide multi-attribute structure learning, a comparison of the automaton to a Multi-Output HMM (MOHMM) [10] is presented here, which is a special case of DBNs [38]. We used the implementation in MoCaPy[8] to create a state-space HMM that allows to be trained with 28-dimensional input vectors. The result is a 28-dimensional output vector described by 28 discrete output nodes. The output nodes are only dependent on the hidden node in one slice. The structure was chosen a priori, following a suggestion by one of the developers. Because we assume independence of attributes, no dependencies in the output layer were introduced. Furthermore, because the prediction is only dependent on the given event, one hidden node per slice is considered appropriate. The MOHMM was trained on the disease group data with exactly the same training and test sets as the ones for the automaton. To look for the best possible solution, we changed the internal node size of the hidden node from 2 to 50. The resulting loss is shown in Table 4.3. We calculated the prediction by a 500-fold sampling of the MOHMM and selected the sequence with the best log likelihood. Again, the quadratic loss was computed. The best prediction was achieved with a node size of 20 which resulted in a quadratic loss of 3.22. This is due to the fact that the MOHMM has a higher probability of predicting the presence of attributes, although they are not observed in the data. Additionally, the internal node size allows for more variables which at first improves the loss but eventually causes the performance to degrade slightly. To explore how the loss depends on the number of nodes per slice, MOHMMs with 2 and 3 nodes per slice were computed and evaluated as

[8]http://sourceforge.net/projects/mocapy/

NS	2	3	5	10	20	28	50
Loss	4.4	3.7	3.5	3.5	3.2	3.3	3.3

Table 4.3: Loss for different node sizes (NS) using MOHMMs

described before. No improvement over the given results could be achieved; the resulting loss ranges between 3.98 and 4.54. This may also be due to the higher number of parameters which cannot be adjusted accurately.

4.2.7 Comparison with Process Mining Algorithms

The detection of automata is not only closely related to HMMs but also to the field of process mining. One standard algorithm is *alpha++* [102] that discovers Petri nets from event logs. It is implemented in the ProM framework [92], which was used for a comparison here. As described in section 2.2.3, Petri nets are able to reveal process progression, and thus it is very interesting to compare the capabilities of such a miner within the given problem setting. However, *alpha++* does not include an automatic event grouping mechanism like the clustering in the presented approach. The group an event belongs to has therefore be defined by the user. That is why we follow a slightly different procedure here, to compare automata with process models and to use the original algorithms of Petri net detection. To derive the corresponding Petri net of a given log sequence, two consecutive steps are needed (cf. Figure 4.9). First, a group (cluster) membership for each event has to be identified in the original data. Then, a new log has to be derived by the original log and the cluster assignments of each event. This is done by replacing each event ID with the cluster ID it belongs to. Last, the altered log is used to create the process model. With this approach the parallel clustering and modeling step was separated into consecutive steps: first clustering and then model construction. Following the experiments of Section 4.2.2, the first step is to check whether a given structure can be rediscovered with the *alpha++* algorithm. Again, the synthetic model (cf. section 3.1.1) was used to extract a log sequence that can be handled by the ProM framework. It was ensured that the log contained all events and connections as well as all needed sequences for loops. For a self-loop on event q_i, *alpha++* needs at least one sequence $s_x = q_0 \dots q_i q_i \dots, q_k$, while for a 2-loop structure between states q_i and q_j there must be a sequence $s_y = q_0 \dots q_i q_j q_i, \dots q_k$. Applying the two-step strategy, a first evaluation shall reveal if the *alpha++* algorithm can recover a known structure with a perfect log sequence, meaning that there are no noisy events in the log. With this constraint, the clustering step can be left out, because of the absence of noise, there are no 'similar' events. Each event is expected to be recorded correctly. Therefore, all events e_i would be assigned to cluster C_i, which is trivial to implement. Moreover, if the miner finds the correct model of a

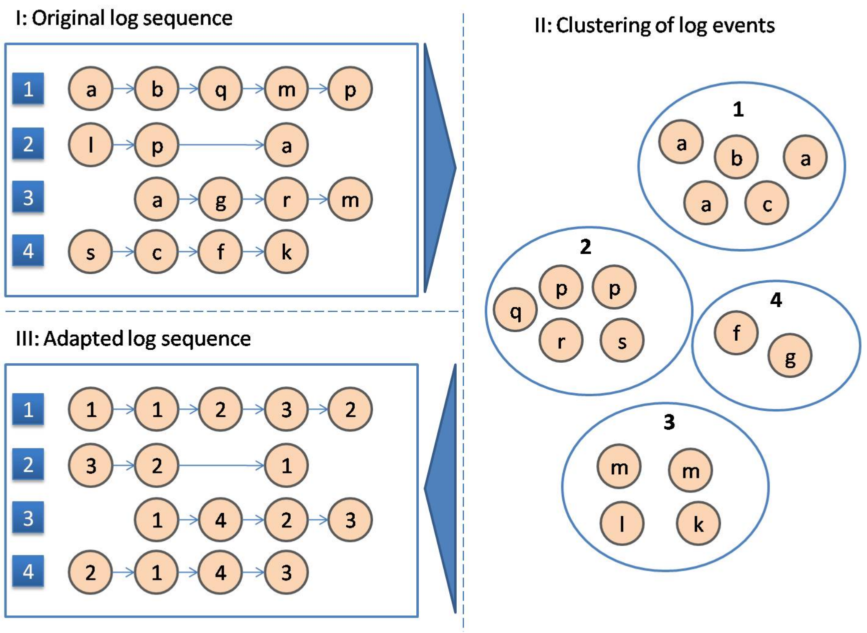


Figure 4.9: Illustration of the sequential approach. In the upper left corner, the original input sequences are displayed. Events are first clustered (right) by treating each event of the sequence as an individual instance. Again, a mapping of each event to a cluster identifier is achieved. By using this mapping and the sequence information of the original data, a dataset as input for a process miner can be derived (lower left part): In each sequence the event identifier is replaced by the corresponding cluster identifier.

noise-free log, it will also be able to find the correct structure in a noisy log with a preceding clustering. In this case, the quality of the model is mainly dependent on the clustering and not of the *alpha*++ algorithm itself.
The *alpha*++ algorithm was applied to noise-free synthetic data; its resulting Petri Net is displayed in Figure 4.10 B. Inspecting the model, there are two main errors. Transition[9] 2 and 5 are not at all connected to other transitions. There is only one connection to the start place (and end place respectively). Second, there is a connection from state 7 to the input place of 6 and 4. This indicates that 4 may come after 7, which is neither indicated in the model nor in the log. The reason why *alpha*++ places this connection is that there is no other possibility to connect 6 and 7 without risking that markers may become stuck in the net (e.g., by inserting an additional place between 6 and 7). This requirement is induced by the demand that only

[9]In the field of Petri nets, states are called transitions. They model discrete events. Events are connected via places that describe conditions. They enable transitions via markers that are produced and consumed in the net. Transitions are displayed as rectangles, places as circles.

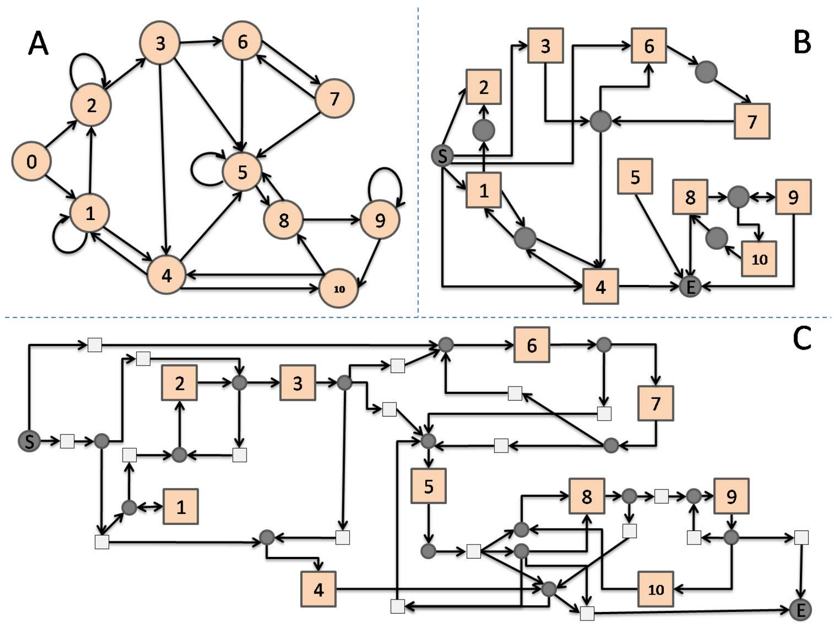


Figure 4.10: Resulting models using *alpha* ++ (B) and *alpha* # (C) compared to the given synthetic model (A)

valid Petri nets shall be discovered. Moreover, in the *alpha*++ algorithm there is no check if invalid connections exist. This leads for example to the connections between 8, 9 and 10 where all required connections are modeled but there is now an additional direct connection between 8 and 10, which is not present in the original model. In fact, this misplacement can be handled by the *alpha* # algorithm [103]. It was designed to detect invisible tasks that can model SIDE, SKIP, REDO and SWITCH constructs. To check the beneficial impact of the invisible tasks, *alpha* # was subsequently applied to the same log. The resulting Petri net is displayed in Figure 4.10 C, where white rectangles are invisible tasks. It shows major improvements over the first Petri net: state 2 and 5 are included correctly, the connections between 8, 9 and 10 are correctly discovered and the wrong connection from 7 to 4 is left out. Nevertheless, there are errors as well. First, connections from and to transition 4 are missing. Only 3 to 4 and 4 to 5 are included in the model. Both two-loop structures between 1 and 4 as well as 4 and 10 are completely missing in the Petri net, although these connections were actually present in the log. *Alpha* # here fails to assign the correct connections. This is a serious error, because the model is simply incorrect and moreover, such two-loop structures are frequently present in the underlying domain (cf. Figure 4.5 for examples). Second, the Petri net is quite confusing due to many invisible tasks that were inserted to model all dependencies. This can be illustrated by looking for the self loop of transition 5. It is indeed

present in the model but stretches over two invisible tasks and thus may not be practical for users to infer information about unknown systems. The same can be detected at the 8 to 5 connection. Such connections can easily be overlooked or even misinterpreted by the user.
Summarizing these results, both algorithms are not able to correctly reveal a known structure in the domain of automata detection. The main reason for this is that they are designed to detect discrete distributed systems. Their main purpose is to model parallelism in a quite linear ordering, which might not be given in a more complex system. Additionally, Petri nets have to satisfy constraints regarding liveliness, reachability and boundedness. Induced Petri nets should avoid dead transitions and thus exclude desired connections or over-generalize, respectively. To provide the same possibilities as in the presented automaton, all transitions should be reachable from the start place. This will allow the modeled system to enter at arbitrary states. To do so, there must be a connection from the start place to each transition in the Petri net. This is only possible, if each event is at least once the first event in a history, which cannot be guaranteed a priori. A further question is how to model probabilities on connections. Consider the input place of 5 in Figure 4.10 C. There are four possibilities of how the input place can be reached from other transitions. However, there is only one connection from the place to transitions 5. Thus, the transition can only be labeled with one event. If delay guards and probabilities of the incoming connections are different, it cannot be properly deployed on one connection, because once the place was visited, one cannot decide which connection was used. In contrast, making small modifications, labeling of definite connections would be possible as well as the annotation of transitions with profiles. Regarding all discussed issues, we can conclude that Petri nets first do not always discover all dependencies of events in complex systems (as in the presented medical domain), and second, are not yet adapted to handle varying transition probabilities of precedent events, while still satisfying the standard model constraints. Due to these limitations, Petri nets cannot yet be used for the given problem statement.

4.3 Conclusion

In this chapter, we proposed a new method for learning process models in the form of probabilistic real-time automata (PRTA) for multi-attribute event logs. To learn such models, a prefix tree is created, in which states are merged when similar enough. State merging is employed because it is currently the best method for learning finite automata in grammatical inference [27]. In order to identify states to be merged, a divisive hierarchical clustering method is used. The algorithm was evaluated on synthetic data, for which the true underlying process was known. Moreover, it was tested on real data from a medical and a biological application domain to examine the resulting structure. The experiments showed that the automaton, in particular its hubs, paths and loops, but also its overall structure, can be related to domain knowledge. To compare the ability of structure identification, standard process mining algorithms were applied on the same synthetic data set. However, they were not able to reveal the correct structure of the underlying process, because of the additional constraints they have to fulfill. Finally, to evaluate the predictive power of the PRTA, the distance of the predicted profiles to the profiles of the true next states was computed. The predictions of the PRTA were compared to the predictions of a combined logistic regression approach, which is considered a standard in this application domain. Additionally, a Multi-Output HMM was trained and tested for its predictive power. The results suggest that the automaton-based prediction performs favorably compared to both logistic regression and Multi-Output HMMs. In the future, we want to further explore the automaton's properties (like the stability, representation of domain knowledge and predictive power) depending on different dataset characteristics. Moreover, we would like to investigate the use of pattern mining techniques for the creation of states (cf. Chapter 5 and 9) and the annotation of states by predicates [49].

Chapter 5

Scalable Induction of Probabilistic Real Time Automata (*SPRTA*)

The induction of real time automata (RTA) was introduced recently [96, 95, 97] and is currently based on a state merging procedure. The PRTA (cf. Chapter 4 [81]) is derived from this type of automaton, but additionally includes profiles and transition labels as well as transition probabilities in the model. Thus, it is able to handle multi-attribute event logs, contrary to RTAs. This enables modelling organism or population development instead of only the relation between, e.g. genes or organisms. To induce a PRTA, the state merging procedure uses the result of a prior clustering step (DIANA [40]). DIANA is a batch clusterer and performs best compared to a variety of other cluster algorithms like k-Medoids, DB-Scan and EM that have also been considered for the induction of automata [81]. DIANA is the only clustering algorithm that was able to induce reasonable results, i.e., find more than one cluster, without using a predefined cluster number, but only relying on a distance constraint. Moreover, so far no clustering with a symbolic description (e.g., frequent patterns) was applied to the induction of automata. In contrast, in the field of regular clustering, there are examples that also group time series data where the final clusters have a symbolic description [31]. The clustering is achieved in a two-step process: first, a local pattern mining method constructs patterns on the attributes of the instances. These patterns are then used to construct predictive clustering trees (PCTs). Next to that, Itemset Constrained Clustering (IC-Clustering) [85] finds clusters having an itemsets description. It uses a branch and bound approach to come up with its results. Co-clustering (bi-clustering, formal concepts) also finds clusters of objects that share frequent patterns and was intensively explored for the standard setting of binary data [7], n-ary relations [15], numeric data sets [70] and useful applications [8]. In graph mining there also exists an approach (Robin) that finds sets of nodes that share a similar itemset [35]. This idea is similar to one part in the automata induction, namely the merge procedure and thus, gives a nice hint of how to improve this step. All of these methods need several scans over the database, which makes them impractical for massive data sets. However they inspired us to propose an efficient one-pass clustering algorithm that finds clusters of objects sharing a maximal set of attributes for the induction of PRTAs.

5.1 Algorithm

This section first, defines the desired model (Section 5.1.1) and second, gives the general approach of how to induce it (Section 5.1.2). Then, a detailed description of the proposed clustering – the essential step of the induction – and especially the used decision function is given in Section 5.1.3. Last, a method to postprocess the resulting clustering is presented.

5.1.1 Problem setting

The task is to model a timed language model over a database D. Let D be a database of histories H_i: $D = \{H_1, \ldots, H_n\}$. A *history* H_i is a sequence of timed events $H_i = (\vec{e}_1, t_1)\ (\vec{e}_2, t_2) \ldots (\vec{e}_l, t_l)$. The event sequence is ordered corresponding to the time label (t_j) of the events. Note that the time labels need not necessarily form equal intervals, thus a varying amount of time can pass between successive events. An event e_i is a binary vector $\vec{e}_i = (a_{i1}, a_{i2}, \ldots, a_{ij})$ consisting of j attributes, where a_{ij} is equal to one if the attribute was observed in this event.[1] We define a probabilistic real-time automaton (PRTA) as a directed graph, where each state q_i is annotated by a profile f_i that represents the events E_i that are mapped to this state. Thus, the profile shows the mean attribute/feature vector of all events that are mapped to q_i:

$$\vec{f}_i = \frac{\sum_{e \in E_i} e}{|E_i|}. \tag{5.1}$$

Transitions $t_{ij} \in T$ of the PRTA connect two states q_i and q_j and reflect changes of events via annotated labels T_{L_j}. These changes are expressed in the so-called *delta notation*: $T_{L_{i,j}} = \Delta(E_i, E_j)$, where

$$\Delta(E_i, E_j) = \bigcup_{\vec{e_k} \in E_i, \vec{e_l} \in E_j} \delta(\vec{e_k}, \vec{e_l}) \tag{5.2}$$

and $\delta(\vec{e_k}, \vec{e_l})$ is defined as the difference of the binary vectors $\vec{e_k}$ und $\vec{e_l}$: $\delta(\vec{e_k}, \vec{e_l}) = \vec{e_k} - \vec{e_l}$. The transitions also restrict the time during which such changes may occur via a delay guard $\phi_j = [t_1, t_2]$ that denotes the minimal

[1]This is similar to the notation of itemsets and thus an event can also be regarded as an itemset.

and maximal time steps when this transition can be passed. Moreover, a transition is annotated with a probability p_j of occurrence. The sum of the probabilities of all outgoing transitions of one state is equal to one.

Definition A PRTA Γ is a tuple $\Gamma = (Q, \sum, T, S, F)$

- Q is a finite set of states
- Σ is a finite set of events to label the transitions
- T is a finite set of transitions
- $S = Q$ is the set of start states
- $F = Q$ is the set of final states

A state $q_i \in Q$ is a pair $\langle E_i, \vec{f_i} \rangle$ where E_i is its set of events and $\vec{f_i}$ is an attribute vector called its profile. Σ are all events $\vec{e}$ that are observed in D. A transition $t \in T$ is a tuple $\langle q, q', T_L, \phi, p \rangle$ where $q, q' \in Q$ are the source and target states, T_L is its label and ϕ is a delay guard defined by an interval $[t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}$. p defines the probability $p \in [0, 1]$ that this transition occurs. The goal is to induce a probabilistic real time automaton Γ that models D and is minimal. Minimal means that the least number of states with respect to some parameter setting must be found [81, 96].

5.1.2 Basic Algorithm

In this section, the basic algorithm for the creation of a PRTA is explained. Algorithm 7 shows the subsequent steps of the approach which is further illustrated in Figure 5.1. The main steps of the induction are (1) the creation of a prefix tree acceptor, (2) the clustering of the nodes of the PTA and (3) the merge of all nodes in a cluster. In the following each step is explained in more detail.

1. The PRTA is induced by first creating a prefix tree acceptor (PTA) [95, 81]. A PTA is a tree of histories, where shared history prefixes are merged to one path. Thus, each node $s_i \in PTA$ corresponds to a set of events E_i. Moreover, a profile is added to each node in the PTA which is equal to E_i. Note that during the PTA creation the data set is only accessed once, while the remaining steps are conducted on the PTA itself. Therefore, the performance is mainly dependent on the following steps.

2. After the PTA creation, the next step is to cluster the nodes of the PTA. In this chapter, we propose to use an online clustering method for this task, for which the motivation is now briefly described and further explained in detail in Section 5.1.3. To cluster the nodes of the PTA, each node s_i of the PTA is treated as an individual instance,

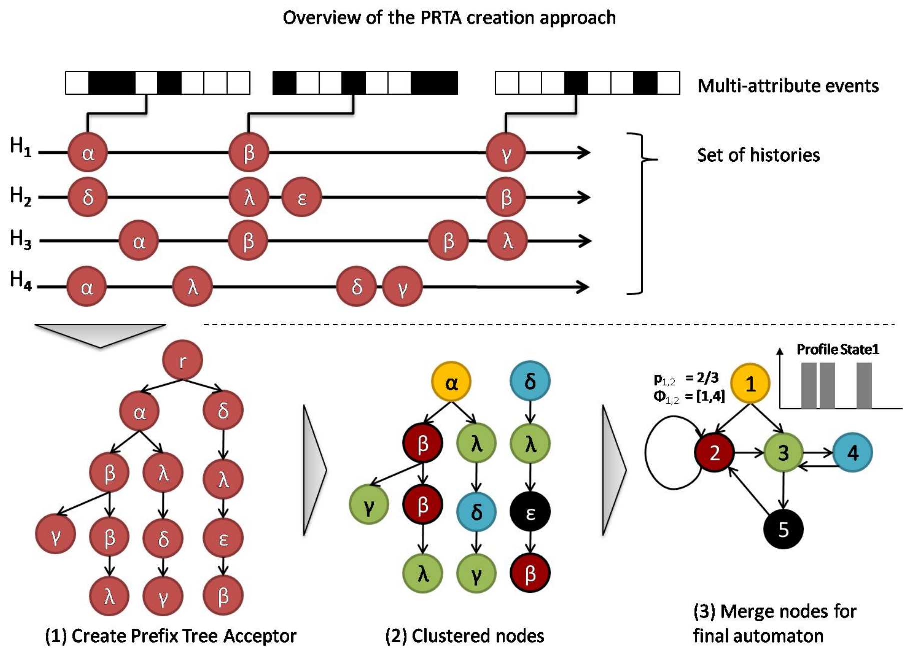

Overview of the PRTA creation approach
Multi-attribute events
Set of histories
H1
H2
H3
H4
Profile State1
P1,2 = 2/3
Φ1,2 = [1,4]
(1) Create Prefix Tree Acceptor
(2) Clustered nodes
(3) Merge nodes for final automaton

Figure 5.1: General overview of the PTA creation. Top: input data - a set of histories consisting of multi-attribute events. The black and white boxes indicate a binary event (α, β or γ). An event is fully described by this binary feature vector. Bottom: successive steps of the PRTA creation. First a PTA is created (left) and then its nodes are clustered (middle). Relying on this clustering, the nodes of the PTA are merged to form the final states (right). For state 1 the profile and for transition $t_{1,2}$ the delay guard and the probability are given, exemplarily.

for which a decision function $fNC(s_i, C)$ (presented in Section 5.1.3) identifies whether s_i belongs to a cluster C_j in the current clustering C (Alg.7, line 4). If this is the case ($k \neq -1$), then the instance is added to cluster C_j (cf. line 6). Otherwise, a new cluster is created and instance s_i is put into the new cluster (cf. line 8). The function $fNC(s_i, C)$ that decides whether an instance belongs to a cluster is based on the following basic idea: The instance is to be placed in that cluster whose maximal frequent patterns (MFPs) best cover the new instance. Then, all instances in a cluster share a maximal amount of properties and are thus very similar. As usual, a pattern is frequent if its frequency exceeds a minimum support threshold of θ and maximal if it is not a subset of another frequent pattern [40]. For the given problem setting, any online MFP identification algorithm can be used that finds a cluster's new set of MFPs after a new instance is placed in the cluster. However, in this chapter, a method is applied that finds

Algorithm 7 *SIPRTA* (Histories H, double θ)

```
1: PTA $pta$ = createPTA($H$)
2: C = createEmptyClustering();
3: for all $s_i \in pta$ do
4:     $k = fNC(s_i, C)$
5:     if $k \neq -1$ then
6:         $C[k]$.addInstance($s_i$, $\theta$)
7:     else
8:         $k = C$.addNewCluster();
9:         $C[k]$.addInstance($s_i$, $\theta$)
10:    end if
11: end for
12: validateAssignments($pta$, $C$)
13: for all $C_j \in C$ do
14:     mergeAllInstancesOfCluster( $C_j$)
15: end for
```

the updated MFPs by only regarding the known MFPs and the new instance [80] and thus perfectly fits the given problem setting.
After each node $s_i \in$ PTA is assigned to a cluster C_j, a postprocessing step validates these assignments (cf. line 12). This step is discussed in more detail in section 5.1.4. The general idea is to identify nodes that were incorrectly assigned to a cluster early in the clustering process and to reassign them to another (better) cluster.

3. Finally, all nodes belonging to the same cluster $C_j \in C$ are merged (cf. line 14) and annotated with a profile of their corresponding events. A merge is an operation that combines two states q_i and q_j into one new state q_k by joining the profiles $\vec{f_i}$ and $\vec{f_j}$ in a single one $\vec{f_k}$ which is equal to their weighted mean:

$$\vec{f_k} = \frac{1}{\sum_{q_i \in C_k} |E_i|} \sum_{q_i \in C_k} |E_i| \times \vec{f_i} \tag{5.3}$$

Depending on the states that are merged, associated transitions t_l and t_k potentially also must be merged if they have the same source and target state. This is done by joining the labels, fusing the delay guards to a new delay guard $\phi'_k = [a, b]$, where $a = min(\phi_k, \phi_l)$ and $b = max(\phi_k, \phi_l)$ and adjusting the transition probabilities. The final PRTA is the PTA, where all nodes belonging to a cluster have been merged.

5.1.3 Finding the Best Suited Cluster

This section describes in detail how the decision function works that identifies the best cluster for an instance (cf. Algorithm 7, line 4). First, the necessary notation is introduced, then two desired properties of the resulting clustering are described and third, the decision function is explained in detail. Last, an example illustrates the decision function.

Preliminaries

The identification of the best cluster for an instance is based on the comparison of the instance and the set of MFPs of each cluster C_i (denoted as p_{C_i}). To compare an instance with a pattern p (a set of features), the *coverage* relation is used. In general, a pattern p covers an instance I_i if each item (attribute) of the pattern is also present (equal to one) in the instance. Function $cov(I_i, p)$ (coverage, Equation 5.4) returns the number of items of the pattern p that also occur (are equal to one) in instance I_i:

$$cov(I_i, p) = |\{a_{ij} \mid a_{ij} = 1\} \cap p| \tag{5.4}$$

Similarly, function $covRatio(I_i, p)$ (coverage ratio) gives the fraction of p's features that are covered by instance I_i:

$$covRatio(I_i, p) = cov(I_i, p)/|p|$$

. If $covRatio(I_i, p)$ is equal to one, the pattern covers the instance. Along the same lines, the coverage can be defined for one instance and a set of patterns. This is necessary, if a cluster C_j has several MFPs P_{C_j}. Then, the maximal coverage of all patterns is calculated similarly to a complete linkage approach: $cov(I_i, P_{C_j}) = \max_{p_l \in P_{C_j}} cov(I_i, p_l)$. Moreover, the coverage can be constrained to only consider patterns with a coverage above a cluster specific threshold $\xi_{C_j} \in [0, 1]$ that defines the minimal coverage of a pattern:

$$cov_\xi(I_i, P_{C_j}) = \max_{p_l \in P_{C_j}} \{cov(I_i, p_l) \mid cov(I_i, p_l) \geq \xi_{C_j} * |p_l|\}$$

Finally, the function that identifies the best suited cluster k for instance I_i should lead to two clustering properties:

1. There must be one MFP for each final cluster to charaterize all states: $\forall C_i \in C : P_{C_i} \neq \emptyset$
2. Each instance must belong to the cluster with the best corresponding coverage ratio to produce clusters with a low intra cluster distance: $I_i \in C_k \leftrightarrow C_k = \text{argmax}_{C_i \in C}\, covRatio(I_i, P_{C_i})$

In the following, we will show how each of these conditions is fulfilled by the proposed decision function and the additional postprocessing step.

A Decision Function for the Online Clustering

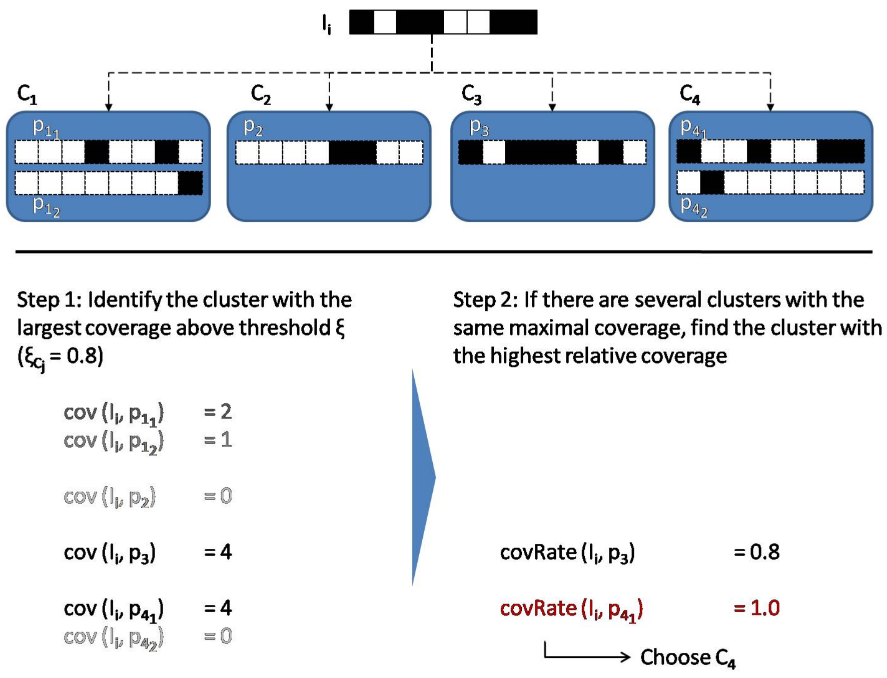
In this section the function fNC is presented. It selects the best suited cluster for an instance during the clustering process. As the clustering takes place in an incremental manner, the best or final MFPs for each cluster are not known in advance. Thus, at the beginning of the clustering process some amount of uncertainty should be allowed. This means that a new instance does not need to be covered by the pattern(s) of a cluster, but only to a specific coverage ratio. The more instances a cluster comprises, the higher should the associated coverage ratio be to ensure that large, meaningful and well-covering patterns are found. The uncertainty term is expressed by the minimal coverage ratio ξ and depends on the number of instances that already belong to a cluster. For small clusters it should be small, while the minimum coverage ratio must increase for large cluster. Moreover, for clusters containing only one instance, at least half of its pattern should be covered. The idea behind this is that the instances in a cluster should share at least half of the items. Thus, the coverage ratio for clusters with one instance $|C_j| = 1$ must be equal to 0.5. Equation 5.5 shows how the term ξ is currently set:

$$\xi = f(C_j) = 1 - \frac{1}{1 + |C_j|^{4/5}} \tag{5.5}$$

For larger clusters only instances that are mainly covered by the MFP(s) can be added to the cluster as ξ is approaching one for $|C_j| \to \infty$. Of course, any other function that satisfies the described properties could be used. However, we chose this function because the root-term parameters of Equation 5.5 do not grow as quickly as, e.g. $1 - \frac{1}{1+n}$ for larger $|C_j|$, but still quickly enough. Moreover, this function ensures condition 1). As there is an overlap of at least one item between a new instance and the MFP of a cluster, there is always a MFP that consists of at least one item. The final decision function for the online clustering algorithm that incorporates all previous conditions is shown in Equation 5.6:

$$fNC(I_i, C) = \arg\max_j \left\{ \frac{\max\{cov_\xi(I_i, P_{C_j})\}}{|\arg\max_{p_l \in P_{C_j}} cov_\xi(I_i, p_l)|} \right\} \tag{5.6}$$

In other words, if there are several clusters with patterns having the maximum overlap, we pick the one where also the biggest portion of this pattern is covered (see also the subsequent example). For instance I_i, the cluster having the largest and best covering MFP is preferred, to keep the MFPs as large as possible. This leads to large differences between the final states, and they are thus easier to interpret.



Ii
C1
C2
C3
C4
p11
p12
p2
p3
p41
p42
Step 1: Identify the cluster with the largest coverage above threshold ξ (ξCj = 0.8)
cov (Ii, p11) = 2
cov (Ii, p12) = 1
cov (Ii, p2) = 0
cov (Ii, p3) = 4
cov (Ii, p41) = 4
cov (Ii, p42) = 0
Step 2: If there are several clusters with the same maximal coverage, find the cluster with the highest relative coverage
covRate (Ii, p3) = 0.8
covRate (Ii, p41) = 1.0
Choose C4

Figure 5.2: Example instance assignment. Top: Instance I_i and clusters with the corresponding MFPs. The patterns are drawn with dashed lines. Bottom: The two main steps of the decision function. Left: The calculation of the coverage for every pattern and the selection of patterns that satisfy the minimum coverage constraint ξ_{C_j} (black font). Right: The coverage ratio calculation finally gives the best cluster (red font).

Example

This section gives a short example to illustrate the proposed decision function. Figure 5.2 shows the main steps that are conducted when a new instance is to be assigned to a cluster. The top of the figure shows the instance (indicated by a partially filled rectangle, each filled square indicates that attribute $I_{ij} = 1$) and the four existing clusters along with their corresponding MFPs (slashed rectangles). For simplicity, we assume that each cluster contains the same number of instances and that ξ is equal to 0.8 for each cluster. Below the horizontal line, the two main steps of the decision function are illustrated. First, the coverage $cov(I_i, p)$ for each pattern p in cluster c_j is computed. All patterns (and clusters respectively) having a coverage larger than the corresponding minimum coverage $\xi * |p|$ are further processed, the remaining are not considered (indicated by light gray). The pattern(s) with a maximum coverage are selected (black font). Note that C_1 is not further considered although $\text{cov}(I_i, p)$ exceeds $\xi_{C_j} * |p|$ because it is not maximal. If there is more than one pattern with a maximum coverage, the coverage ratio is computed to choose one final cluster. This is the second step. The pattern (and the corresponding cluster) that has the highest coverage ratio (indicated by red) is selected.

5.1.4 Postprocessing

Because the best separation of the instances (and the corresponding MFPs) is not known in advance, wrong assignments can occur at the beginning of the clustering. This is due to the small minimum coverage for small clusters. This leads to clusters that contain outliers, i.e. instances that would fit better in another cluster after all instances were processed which again would violate condition (2). To address this problem, a postprocessing step is conducted (*validateAssignments* in Algorithm 7 line 12). For each instance (node s_i) the assigned cluster is validated whether the coverage of its MFPs is maximal compared to all other clusters. If this is not the case, s_i is assigned to the cluster with the highest coverage. Thus, condition (2) is also fulfilled by the proposed method.

5.2 Experiments

This section first introduces the data sets that were taken for the evaluation of the proposed induction method. Then, the performance of the approach is presented based on a synthetic data set. This includes the rediscovery of a known automaton, a stability analysis and specifically, a cluster quality examination. Last, the proposed method is tested on real-world data sets to evaluate its informational content. Moreover, the results are compared to an existing method of automata induction (using DIANA), if applicable. All experiments were conducted on a 2.7GHz machine (Ubuntu) with 2GB main memory. The quality of the automaton is measured by several indicators. If the true underlying automaton structure is known, the recovery ratio (RR) specifies how many states were correctly identified by the automaton: $RR = \frac{|Q_{corr}|}{Q_{total}}$. A state q is correctly identified ($q \in Q_{corr}$) if its set of MFPs corresponds to the binary profile of an original state. To judge the accuracy of the resulting transitions, the F-Measure is used. The quality of the induced clustering is measured by the Adjusted Rand Index [46] (ARI). Additionally, the runtime and the final number of states are given.

5.2.1 Performance on the Synthethic Data Set

First, the resulting structure of the automaton on the synthetic data set is shown. Second, the runtime and structural dependence on the number of input histories and the minimum support θ are presented. Third, a stability analysis shows how often a correct automaton can be found and which states are hardest to detect. A comparison to the DIANA based approach is also

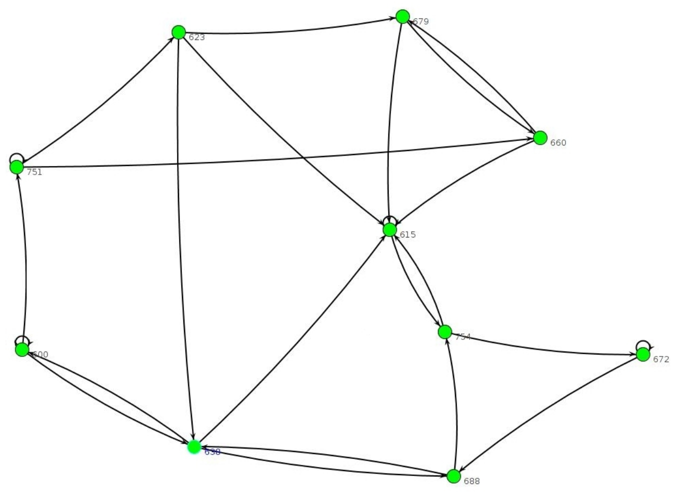


Figure 5.3: Resulting automaton for the online approach with postprocessing (Software screenshot). Again, the circles represent the states, arrows transitions. The states are aligned corresponding to the circles in Figure 3.1 having the most similar profile.

given. Last, the quality of the induced online clustering is compared to the DIANA based induction via the *ARI* difference.

Identification of a Known Automaton

First, a known automaton is to be rediscovered which was already successfully accomplished by the induction method based on the DIANA clustering [81]. The automaton given in Figure 3.1 is used for this analysis. Figure 5.3 shows the induced automaton. For the final automaton we observe the following. The MFPs of the final clusters (and thus states) exactly match the predefined states and all transitions are inferred correctly (not illustrated). There is only one exception: transition [1,3,4] to [3,4,5] (state 2 to state 7) is not present in the original automaton. Because of the introduced errors in the data, event [3,4,5] was created instead of [3,5] after event [1,3,4]. This of course matches the later profile so that it is not assigned to state [3,5] but to [3,4,5]. However, this is a data induces error and not a systematic drawback of the algorithm. The transitions and the corresponding probabilities were also identified correctly, with the largest deviation of 0.15. The benefit of the induction of automata becomes more evident if such a timed data set shall be modeled with other process models, like e.g., Petri nets. We thus also compare the proposed approach to the *alpha#* [103] algorithm that is an improvement of the *alpha++* algorithm. As such process miners do not include an automatic event grouping mechanism (like the clustering), the data set was preprocessed for this task and the resulting states were pre-

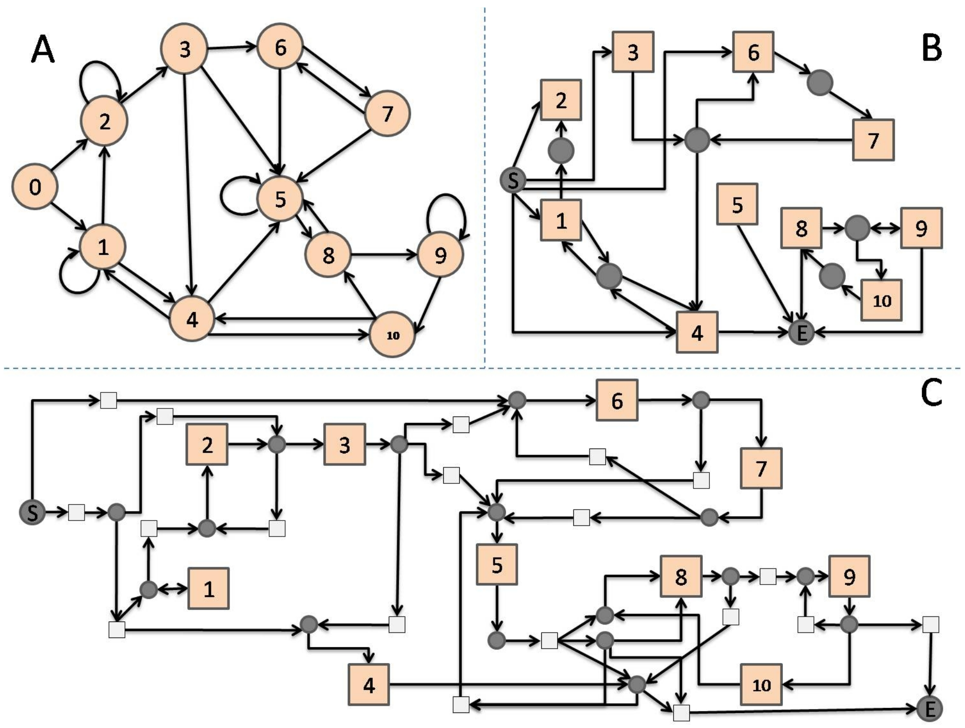


Figure 5.4: Resulting Petri nets using *alpha*++ (B) and *alpha*# (C) compared to the given synthetic model (A), cf. Figure 3.1.

sented to the algorithm. Thus only the states' relations and not the states themselves had to be found. Figure 5.4B and C show the resulting models for the *alpha*++ and *alpha*# algorithm. Note that there, the squares are called states while the circles are called places. Moreover, Figure 5.4 C also includes hidden states (white squares) to model e.g., switches. Although the overall structure is identified correctly, there are states in both models that do not correctly reflect the data: e.g., *alpha*++ fails to correctly infer the relations of state 2 and 5 while *alpha*# does indeed better, but also misses transitions (e.g., from state 7 to state 4). This is due to the fact that Petri nets have to satisfy constraints regarding liveliness, reachability and boundness. Therefore, for more complex temporal relationships as given in the current problem setting, such models are not appropriate.

Parameter Dependence

This section analyzes some structural properties of the automaton for the parameters: minimum support and number of input instances. Figure 5.5 shows two important properties of the resulting automaton when varying the minimum support θ in the clustering step. First, for a higher θ, the number of states decreases. This is the result of the decreasing size of the

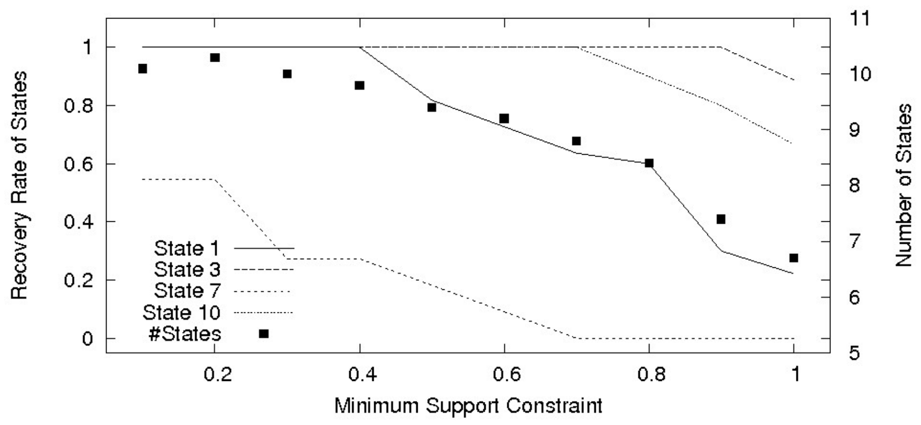


Figure 5.5: Recovery ratio (RR) for states 1, 3, 7 and 10 of the synthetic automaton.

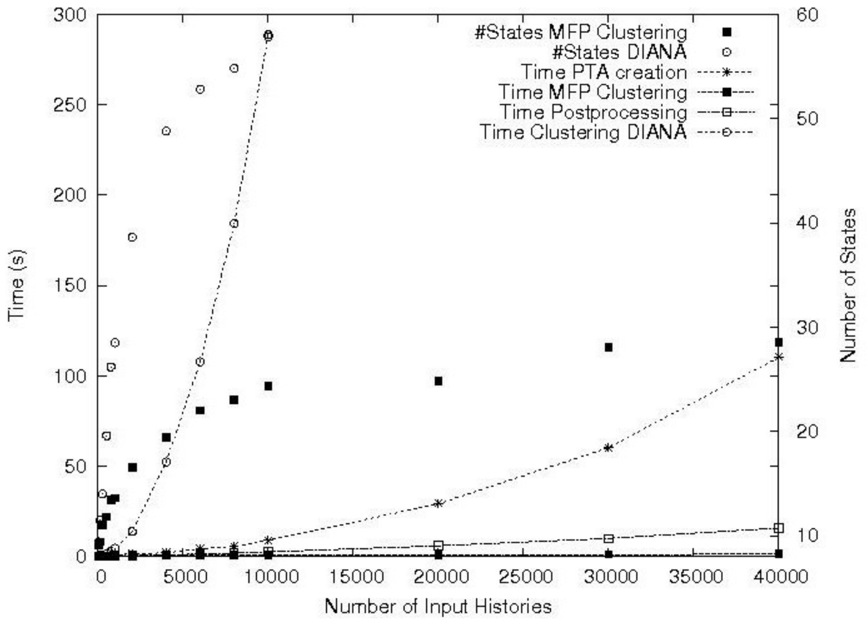
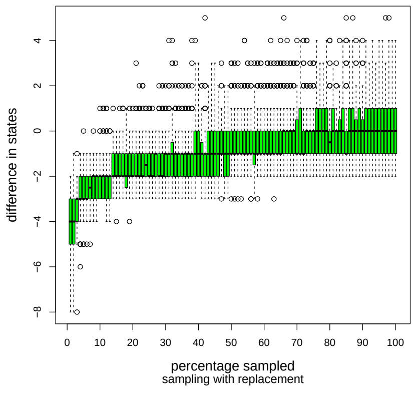


Figure 5.6: Runtime and final number of states with increasing data set sizes for the synthetic data sets.

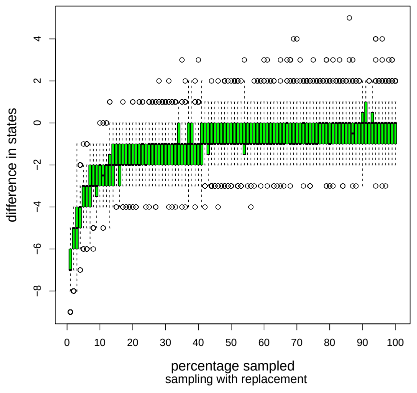
MFP found in the clusters. As there are exceptions in the data, not every concept of the state is present in each event. Thus the frequencies of large patterns decrease and only small MFPs are found. These small MFPs are of course better to cover than large ones even if the cluster size is large and thus less clusters are necessary for a good coverage ratio. The number of transitions (not shown) decreases simultaneously, because the number of states is also decreasing. Second, there are states that are identified better than others. The lines in Figure 5.5 exemplarily give the RR for the states 1,3,7 and 10. State 7 is very difficult to recover due to two reasons: a) it is very similar to state 3, thus a separation is not always possible and b) it only has one incoming transition with a small probability. Therefore, this event is not often present in the data and can also be regarded as an exception of state 3. Additionally, state 10 is harder to find with increasing minimum support. Again, this is due to the similarity to state 5. For the remaining states a quite stable recovery ratio is observed, but the detection rate decreases with increasing minimum support, due to the smaller MFPs. Another important parameter is the number of input histories, and how this number affects the runtime of the approach. This is shown in Figure 5.6 once for the presented online approach as well as for the approach based on DIANA clustering. We tested our approach on up to 40,000 histories, which corresponds to a data set having 220,000 events (because each history has 5.5 events on average). Note that the traditional approach (using DIANA clustering) can handle only 2000 histories of the same length due to its high complexity. The runtime of the online approach is divided into three separate runtimes: the time that is needed for the PTA construction, the time for the clustering and the time of the postprocessing step. For the DIANA approach only the time for the clustering is given, because the PTA construction time is the same and moreover, no postprocessing step is conducted. As expected, the runtime increases for larger data sets for both approaches. However, the runtime difference for large data sets between the online approach and DIANA is tremendous. The clustering part of the online approach now is the fastest step. Considering the number of final states, both methods result in too many states. This can be explained by exceptions in the data. As more data is available, rare events, i.e. events that strongly differ from their original states, are observed more often. Thus, they are also clustered in a separate group. However, the number of final states for both methods converges, but altogether the online approach identifies fewer states than the DIANA based induction. This indicates that it may be able to better find the main information in a data set.

Stability Analysis

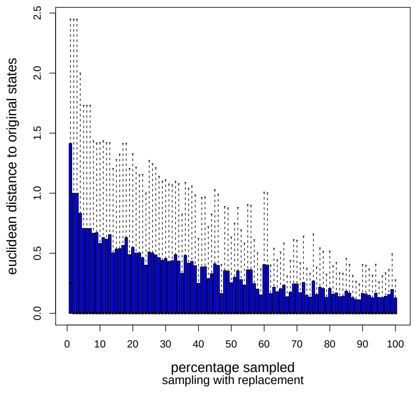
In this section the stability of the online approach is presented. Figure 5.7 shows the result of a bootstrap analysis. Figure 5.7b gives the difference



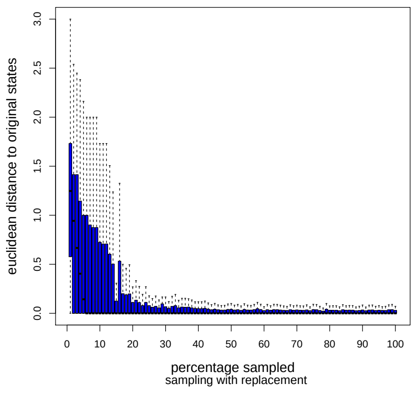
(a) Boxplot: Difference of states DIANA



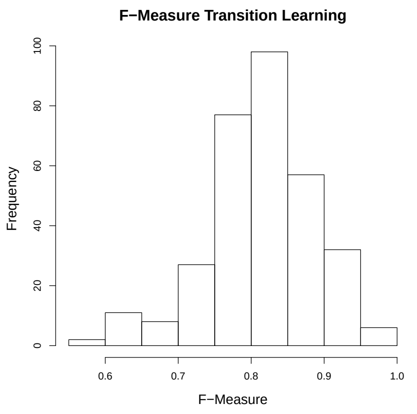
(b) Boxplot: Difference of states online



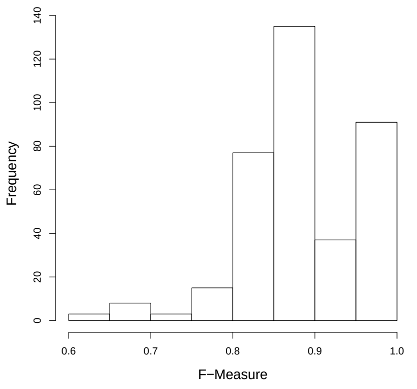
(c) Loss of learned states DIANA



(d) Loss of learned states online



(e) F-Measure of transitions DIANA



(f) F-Measure of transitions online

Figure 5.7: Bootstrap analysis: Structural dependencies for the DIANA based clustering compared to the online approach, (DIANA $c = 0.3$, online $\theta = 0.5$)

between the number of learned and expected states (10) on the 10 synthetic data sets of 100 histories. The higher the sample size, the more likely the correct number of states is found. Although the average number of states is 10 for a sample size above 75%, there are exceptions that find 11 or 9 states. A smaller number of states can be due to a missing example in the data, a higher number of final states may be due to exceptions in the data that are placed in a separate cluster. Figure 5.7d shows the average distance between the learned states and the most similar states of the original automaton. For small sample sizes, this distance can be large, but it rapidly decreases for larger data sets. Figure 5.7f shows how often the transitions were identified correctly. For this figure, only automata that induced the correct number of states were regarded: DIANA found the correct number of states in 318 cases, while the proposed online approach successfully found the 10 states in 369 cases. Each learned state is mapped onto its most similar original state. Then, the corresponding transition between the learned states can be validated whether they are correct or wrong and thus whether true or false positives/negatives are identified. The figure shows that in most cases the transitions are identified correctly. Precision and recall (not displayed) are mostly above 80%. However, recall is the limiting factor as it has a worse distribution than precision. This indicates that the found transitions are correct, however some transitions are missing. Comparing these results to the DIANA based induction (cf. Figure 5.7a - 5.7e bottom) the presented online approach outperforms it in each category: The number of inferred states is closer to the original (cf. Figure 5.7b bottom: DIANA overestimates the number of states) and has a smaller variance. Moreover, the distance of the states to their closest original is smaller (cf. Figure 5.7d bottom) while fewer data samples are necessary to reach this quality (for sample sizes of 20% the final state distance is already achieved). Thus, also the transitions are learned better (cf. Figure 5.7e bottom: the distribution of the F-Measure of DIANA is shifted to the left).

Quality of the Online Approach

Another parameter to evaluate the quality of the online clustering approach is the resulting *ARI*. The zoo data set is used for this task because it also has a class label assigned to each instance. To validate exclusively the clustering, only the resulting clustering for the nodes of the PTA is inspected, and not the automaton itself. Table 5.1 shows the resulting *ARI* for a DIANA based induction and the online approach. Note that the results for the different parameters (c = distance cutoff and θ = minimum support) cannot be compared directly because they have a different influence on the clustering. However, the results show that a similar quality can be achieved by the online approach. Its best *ARI* of 0.84 reaches the best value of the DIANA approach. Moreover, the online approach achieves at least an *ARI*

D	$c = .1$	.2	.3	.4	.5	.6	.7	.8	.9	1.0
	0.17	0.35	0.47	0.49	0.58	0.86	0.80	0.48	0.22	0.0
O	$\theta = .1$	.2	.3	.4	.5	.6	.7	.8	.9	1.0
	0.40	0.36	0.56	0.50	0.60	0.58	0.77	0.84	0.63	0.7

Table 5.1: *ARI* for the DIANA clustering (D) and online clustering (O) depending on their primary parameter for the zoo data set

of 0.36, which is better than the DIANA clustering. Finally, these numbers show that although the online approach induces fewer clusters than DIANA, these clusters also make sense and thus the clustering informative.

5.2.2 Performance of the Online Approach on a Real World Example

In this section, examples of how to use such an automaton on real world data are given. Moreover, the runtime and structural dependence on the number of input histories for these data sets are presented.

Yeast Gene Expression

A first experiment addresses the knowledge one can gain from such a type of automaton. Figure 5.8 presents the automaton (with the corresponding profile for each state) that was induced for the yeast gene expression data set. The first important point that can be observed from this figure is that it shows a cyclic structure, which is known to be the cell cycle. The automaton is thus capable of correctly identifying the stages of life in a population or of individuals. Second, each state is annotated with a specific gene expression profile that shows the current metabolism of the cell. Following the cycle, one can easily observe which genes are activated after another and which genes are co-expressed. Third, a resting phase between state 1 and 2 is identified via the delay guard ($\phi = [1, 2]$) on the corresponding transition. This shows that the cell can rest between two cell dividing phases. There is only one state that could be better separated from the other state: the profile of state 6 is not as specific as the others. This is due to the fact that there is one instance included that should occur as a separate state before state 6. This also leads to state 7 being a loop structure. However, this assignment is due to the initial variability in the clusters.

Hepatitis Data

This section presents the results of the online approach on the Hepatitis data set. To test the performance, differently sized data sets from the original one were derived. Compared to the synthetic data set, the Hepatitis data set is

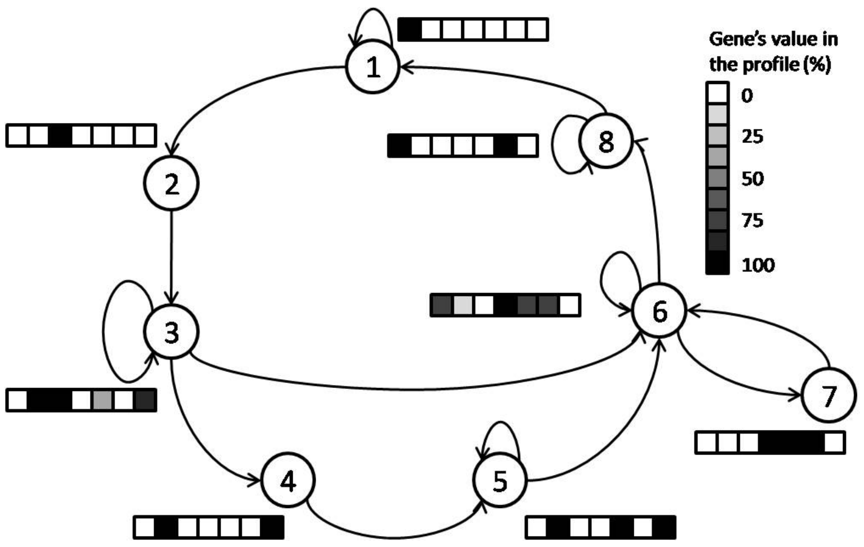


Figure 5.8: Automaton for the yeast data set. For each resulting state (circle) the corresponding profile of the state is given. The profile is illustrated by the boxes next to each state where each gene's expression values is indicated by a grey-scaled square. The expression profile for gene i is given by the ith square, beginning from the left. The darker, the higher is the gene's expression value.

a difficult one, because it is very dense and has more attributes. Moreover, the histories are long and the resulting PTA is also very large, because only few histories have the same prefix. Figure 5.9 gives the runtimes for the Hepatitis data set for the presented approach. (The results for DIANA are not displayed as only the smallest data set could be processed.) Even for the smallest number of histories (50), DIANA needs longer (218 s) than the online approach (173 s) for the largest data set (1236 histories). This is caused by the huge number of states on which DIANA operates: 2465 states in the PTA must be clustered for the smallest data set. Although this is still feasible, DIANA runs out of memory for the larger Hepatitis data sets. But also for the online approach, the runtimes are higher. Note that the 1236 histories lead to a PTA with 51,699 states. However, comparing the number final states (188) of the online approach for 1236 histories to the DIANA clustering with only 50 histories (338 final states), again a more compact representation can be inferred. Another interesting aspect of the online approach is the distribution of the number of MFPs per final cluster and their lengths. Figure 5.10 shows how these distributions vary for more and more input histories. Note that for a better visualization not every data set size result is displayed. Figure 5.10 shows that the majority of clusters is represented by long MFPs: mostly more than two attributes contribute to

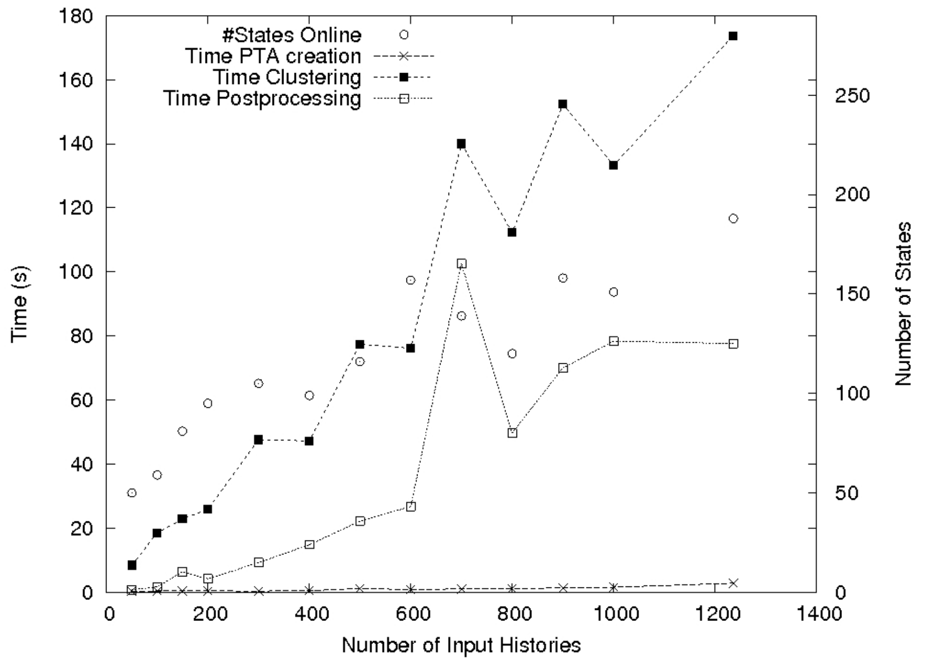


Figure 5.9: Runtime and number of states of the online approach for the Hepatitis data sets.

a MFP. This distribution is quite constant over the different data set sizes. Next to that, in most cases, only few representatives are induced by the clustering (cf. the lines in Figure 5.10). Moreover, the more histories are used for the creation of the automaton, the fewer MFPs are present in the clustering. This indicates that instances sharing large patterns are grouped together, and thus homogeneous states are found.

5.3 Conclusion

We presented a scalable method of learning probabilistic real-time automata (PRTAs), a new type of model that captures the dynamics of multi-dimensional event logs. It is based on a state merging process, which is the current state of the art in automaton induction. As the state merging is guided by a clustering in our approach, the clustering procedure needs to scale well to make the overall approach scalable. Therefore, we employ an online clustering procedure as a plug-in that is based on maximum frequent patterns (MFPs) to decide where to assign states. The approach is compared to induction variant using DIANA clustering. The proposed online method is faster than the one based on DIANA, computes a more compact automaton structure and also works for significantly larger data sets. In future work,

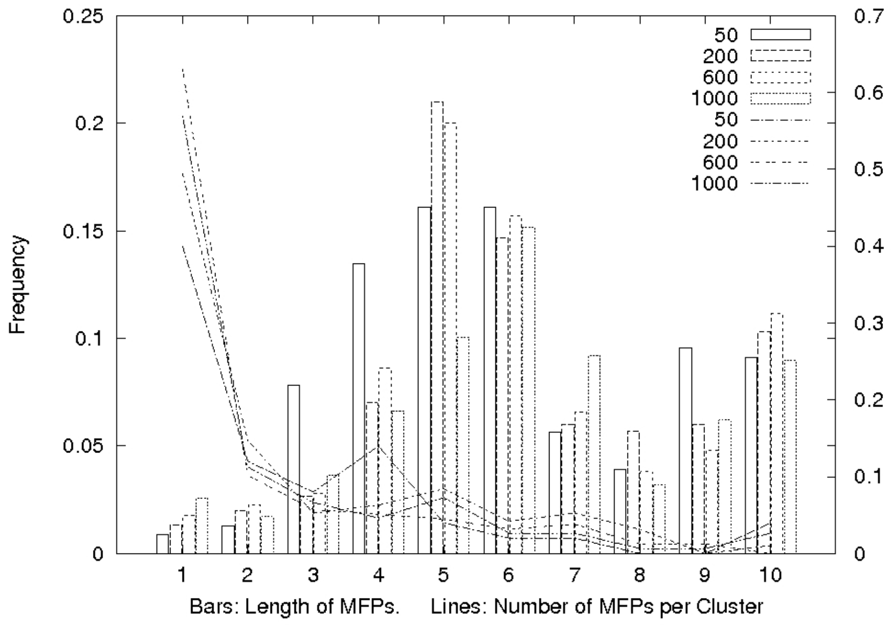


Figure 5.10: Distribution of the length of MFPS (number of items in a pattern) and number of MFPs per cluster (seconday axis).

we would like to improve the creation of the PTA, which is currently another bottleneck in the induction of automata.

Chapter 6

Augemented Itemset Trees

Data streams are ubiquitous today due to the surge of data produced by sensor networks, social networks and high throughput methods in biology and medicine, to name just a few areas of interest. For streams of binary data, one of the most basic and fundamental data types, online versions of classical data mining methods for itemset mining have been developed [17, 91]. These methods allow for mining patterns in sliding windows or for the full amount of data via, e.g. data buckets. While the sliding windows approach addresses the task of concept drift and does not provide exact solutions over the complete data stream so far, the approach based on buckets faces a number of problems. First, it is not feasible if main memory is limited (below the expected data set size) or I/O operations are expensive. Second, it is not incremental in the strict sense of the word, with updates using newly observed instances. Third, the detection of change is not easy, because either it is delayed due to large buckets, or the mining process is slowed down due to small buckets for which the pattern extraction step must be repeated frequently. In this chapter, we consider the incremental case of maximum frequent pattern (MFP) mining where the set of solutions is available after each instance without having to store previous instances explicitly. This can also be seen as a step towards a batch-incremental algorithm for online MFP mining, in which the solution set is updated after a batch of instances. Algorithms for online MFP mining can be useful for various purposes, e.g. for building fast incremental clustering algorithms on 0/1 data with maximal frequent itemsets as cluster representatives, in analogy to a graph clustering approach that was proposed recently [83]. More specifically, we introduce the Augmented Itemset Tree (AIST), a new data structure that helps mining maximum frequent patterns incrementally. It is a mixture of itemset trees (IST) [39] and FP-Trees [41], combining advantages of both: a canonical ordering of patterns and a connection between patterns sharing the same items. The first idea is that search starts with maximal pattern candidates P, which is opposite to traditional search strategies, beginning with 1-itemsets and iteratively increasing the length of the candidate patterns. But if the maximal patterns are expected to be large, i.e. consist of more than half of all possible items, a top-down search strategy is more appropriate: the number of candidates of the pattern space in each candidate level is symmetric $\binom{n}{k} = \binom{n}{n-k}$ (e.g. $|P_1| = |P_{n-1}|$). If maximal patterns are large, then fewer candidates are evaluated, when starting with maximal patterns instead of beginning with patterns of size one. A second strategy is that the set of candidates is composed of all former MFPs and the patterns of the new instance. These patterns serve as a starting point for the search of the new maximal frequent patterns, which prevents unnecessary patterns from evaluation.

6.1 Related Work

Frequent pattern mining is a well-studied problem and many improvements have been achieved since its introduction [1]. In this part of the thesis, we consider the case that the set of maximal frequent patterns is to be identified in an incremental manner. This also includes that instances are presented to the algorithm one by one, i.e. no instances are kept in memory and only one data scan is allowed. In the following, we discuss various solutions that were proposed for an online setting. The earliest approach, Apriori [16], relies on a candidate generation and test procedure involving several scans over a database. This is of course not possible for the incremental case, because the final size of the data set is not known in advance. However, there are adaptations that are based on Apriori via piece-wise mining of bags of instances or using a graph structure to identify the positive or negative border in modified databases [90, 91]. Han *et al.* [41] developed an algorithm that uses a tree-like structure for the mining process, the so-called *frequent pattern tree (FP-tree)*. It stores the content of the given data set by scanning the database twice and creating a pattern tree, where the items are included corresponding to their frequency. This property forces the second database scan. Next, the FP-tree uses the pattern growth technique to derive frequent patterns. Each path of the tree stores items in decreasing frequency. A main disadvantage of such a tree structure for the incremental case is that the frequency of the patterns is not known in advance. This implies that the tree must always be restructured [55] when the frequency of itemsets changes. If a reordering occurs frequently, such an algorithm will be less efficient for incremental data. An approach to use FP-trees in an incremental setting was proposed [17], but does not reach the performance of other methods [55]. Another group of algorithms mines frequent patterns using a data structure similar to itemset trees. An efficient algorithm, CanTree, that incrementally mines frequent itemsets in a finite database was proposed recently [55]. It outperforms FP-based algorithms and also explains why a tree with a given canonical ordering is to be preferred in an incremental setting: No reorganization of the data structure is necessary. However, CanTree is not designed to keep track of the current maximal frequent itemsets, which is the problem addressed here. Instead, the mining step is only conducted after the database scan is finished. In a possibly infinite online setting, this may never be the case. Therefore, a modified version of the CanTree has been developed that is able to keep track of the current maximal frequent patterns. We apply a technique similar to previous work that describes how to mine frequent closed patterns for data streams [54] based on the current closed sets and a new instance. However, this theoretical work neither proposes an efficient data structure nor provides an experimental evaluation. Approaches to mine frequent patterns incrementally, but in a different setting, were also proposed in recent years (Moment [18], SWIM [63], DISC [20], FLAME [32], DSM-MFI [58] and CP-tree [53]). Moment, SWIM, DSM-MFI and DISC find frequent patterns in a sliding window over data streams and use instances for later evaluations. FLAME and CP-tree identify approximate patterns. Moment uses the closed itemset tree (CET) data structure that represents the border of frequent patterns in its nodes. Four node types differentiate between frequent patterns and promising patterns, i.e. patterns that are not yet frequent but may become frequent. Although its node frequency update strategy is very efficient, it needs to keep track of the current transactions of the sliding window. Otherwise it is not able to update relevant node counts for patterns that were not yet incorporated into the CET. The same argument applies to the SWIM algorithm [63] and the DISC strategy [20]. For unrecorded patterns that may become frequent, additional passes over the current window are necessary. Similarly, MFP Mining based on the SG-Tree [59] also stores the itemsets that were observed but uses the same top down strategy to find the MFPs. In contrast to these approaches, the AIST works for the instance-incremental setting, where each instance is only observed once, patterns over the complete data stream are of interest and an immediate response is necessary.

In the remaining of this chapter, we describe the problem setting and the data structure to support the solution. Subsequently, each of the main steps of the algorithm is explained in detail.

6.2 Problem statement

Let D be a data stream of instances x_i, each instance being a vector of l binary attributes ($x_{ij} \in \{0, 1\}$). We assume that one instance is observed after the other, starting with $x_1, x_2, ...$ The goal is to find all MFPs for the data stream part $D_m = \langle x_1, \ldots, x_m \rangle$, in other words, for each new instance x_m, the updated set of maximal frequent patterns. In the following, we will use the itemset notation for patterns and instances: Each instance x_i is transformed into an ordered itemset $I_i = \iota_1, \ldots, \iota_z$, where item ι_j is included in the itemset if $x_{ij} = 1$. This turns a data set into a multiset of itemsets. As usual, the support of an itemset I_i, *support*(I_i), is defined as the percentage $|Y|/|D_i|$ of itemsets Y in D_i such that $I_i \subseteq Y_j$. Informally, this is the relative frequency of instances that include the itemset I_i. All patterns exceeding the minimum support constraint (*minsup* constraint) $\theta \in [0, 1]$ are frequent (F), infrequent otherwise ($\bar{F}$). The *more general* relation $p \prec q$ implies an ordering upon patterns p and q [1]. Pattern p is more general than pattern q iff p occurs whenever q occurs, i.e. all items of p are also included in q but not vice versa. For the pattern domain of items, $p \prec q$ is defined by set inclusion $p \subset q$. Pattern p is called *subpattern* of q, while q is a *superpattern* of p. Let the set of all patterns p that are more general than q be denoted by G:

$$G(q) = \{p \mid p \prec q\} \tag{6.1}$$

Accordingly, $S(q)$ comprises all patterns that are more specific than g. A pattern p is *maximal* if there is no other pattern q that is frequent and a superpattern of p.

$$p \in MFP \leftrightarrow p \in F \wedge \neg\exists q : q \in F \wedge p \prec q \tag{6.2}$$

If a pattern p is a subset of an instance I_i, we say that the patterns *covers* the instance, or alternatively, *occurs in* the instance. The final problem can hence be expressed as: For a data set of instances I, successively update the set of MFPs with respect to a given minimum support threshold θ.

6.3 Proof of main concept

The basic assumption of the approach is that the updated MFPs are determined by the MFPs known up to that point and subsets of the new instance. Similar approches also make use of this property [54, 75] To prove that this is sufficient, we first summarize how the set of updated frequent patterns is identified and then, how the corresponding MFPs can be derived. Therefore, we divide the set of all patterns P into four subsets $P_1, \ldots, P_4$ that define whether a pattern is frequent (F) and covered (C) by a new instance (cf. Figure 6.3, right): $P_1 = \{p|p \in F \cap \bar{C}\}$, $P_2 = \{p|p \in F \cap C\}$, $P_3 = \{p|p \in \bar{F} \cap C\}$ and $P_4 = \{p|p \in \bar{F} \cap \bar{C}\}$.

Theorem The set of (M)FP (F') including a new instance, is a subset of the patterns of (1) all former (M)FP (F) and (2) all patterns occurring in the new instance (C).

Proof *Start of induction, A(1), regarding the first instance*
There are no frequent patterns in an empty set: $F = \emptyset, \bar{F} = \emptyset$. The set

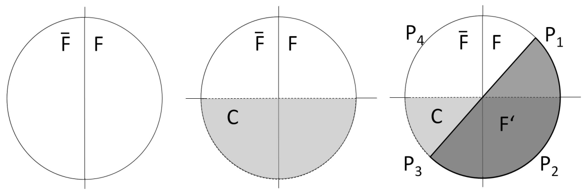


Figure 6.1: Left: Each pattern initially belongs to either the set of frequent (F) or infrequent ($\bar{F}$) patterns (white). A new instances induces the set of contained patterns (C, light grey). The updated set of frequent patterns (F', dark grey) then is a subset of F and C. Right: *HeaderTable* (left) with the first links for each item ι. Dashed arrows indicate the *nextNodeList*. Right: *pattern tree*.

of frequent patterns (F') of a one-instance (I_1) set is equal to all patterns p that occur in the instance I_1, $C = \{p | p \in G(p_{I_1})\}$, because they all have frequency one. As condition (2) applies here, the claim is true for an initially empty set. The first MFP is p_{I_1}.
Induction step, A(n+1), regarding the next instance
For each subsequent instance the following is given: There are n patterns in the observed patterns set (P), without the new instance I_{n+1}. A pattern is called observed, if it was covered by one of the instances $I_1 \dots I_n$. For each observed pattern $p \in P$ with absolute frequency k, the relative frequency is known: $f(p) = \frac{k}{|I|}$. Each pattern belongs either to the set of frequent (F) or infrequent patterns ($\bar{F}$), $F \cup \bar{F} = P$. Incorporating instance I_{n+1}, the frequency of a set of patterns ($p \in C \subseteq F \cup \bar{F}$) is increased (cf. Figure 6.3). All patterns in set P_4 cannot be in F' because their frequency after the insertion is lower than before $f(p)' = \frac{k}{n+1} < \frac{k}{n} = f(p) < \theta$, so that they cannot exceed the minimum support threshold θ. Contrary, all patterns in set P_2 are frequent, because their frequency increases and thus cannot drop below the minimum support threshold. Set P_1 includes patterns that may become infrequent, because they do not occur in the new instance and thus their frequency can drop below the minimum support threshold. Set P_3, in contrast, holds all patterns that were not frequent before, but may become frequent, because their frequency is increased by the new instance. Thus, F' consists of patterns in P_1, P_2, P_3: $F' \subseteq P_1 \cup P_2 \cup P_3$. □

Next, all MFPs must be identified. A pattern $p_r \in F'$ is a MFP for all frequent patterns $p \in F'$ that are more general than p_r: $p \in G(p_r)$. If an old MFP p_r was in P_2, there are two possibilities when a new instance arrives: either the new instance I_{n+1} covers p_r or not. In the second case, p_r will stay MFP (condition (1)) – no pattern in P_2 can be larger. In the first case, again two cases apply: I_{n+1} is equal to the pattern, then p_r will also stay MFP or I_{n+1} is a superpattern of p_r. Then, p_r is still frequent but may not become maximal, because a superpattern of p_r can become frequent. This may be true for all patterns p_r^* 'between' p_r and I_{n+1}: $I_{n+1} \preceq p_r^* \prec p_r$. For this situation condition (2) applies, the set of the new MFP is derived from the new instance I_{n+1}. To avoid that, F' is inspected whether a pattern is not maximal anymore. If $p_r \in P_1$, it may become infrequent, but some p may nevertheless be frequent. In this case, following condition (1) a depth first search (DFS) is conducted that tests each subset of p_r until a new MFP p'_r is found: $p'_r \in G(p_r)$. Finally, only patterns $p \in P_3$ remain that are now frequent, where no MFP was identified yet. As only patterns that occurred in the new instance $p \in G(p_{I_{n+1}})$ may become frequent (condition (2)), the new MFP is also a subpattern of $p_{I_{n+1}}$ and is identified via DFS.

6.4 Main idea of the used data structure

The basic assumption of the approach is that the updated MFPs are determined by the MFPs known up to that point and patterns covered by the new instance. The proof is ommitted here due to space constraints but is similarily shown for other approaches [54, 75]. Intuitively, it is clear that no new MFP can emerge if it was neither in the old set of MFPs nor in the new instance. The data structure that makes use of this property is called Augmented Itemset Tree (AIST). Figure 6.2 shows the main idea of an AIST: It consists of a pattern tree (right part) and a *headerTable* (left part). It is a mixture of FP-trees [41] and ISTs [39] and designed to fulfill two main requirements: (1) quick look-ups for patterns and (2) quick subset checks. Each node represents a pattern with the last item being ι_j, where the identifier j of a node corresponds to item ι_j. The first requirement is solved by using an itemset-tree like structure: nodes follow a canonical ordering in the tree (top-down arrows), i.e., the parent of a node n has a smaller identifier than n itself. To cope with the second constraint, each node is linked to another one with the same identifier but of smaller or equal tree level. This sets up a list (*nextNodeList*) that connects all patterns sharing item ι_j, ordered by decreasing tree level. This idea was already used in FP-trees and reduces unnecessary subset checks. A *headerTable* stores the first link for each item ι_j. Although it is a mixture of two known data types, it is still different from them. FP-trees are not canonically ordered and ISTs do not include cross-links between patterns sharing items. The combination of the two concepts enables efficient mining of MFPs in an online setting.

6.5 Definition of the AIST

The purpose of the AIST is to keep track of all relevant patterns, i.e., MFPs or former MFPs. Since the AIST is not forced to store all possible (subset) patterns, it saves a lot of space and is also computationally more efficient.

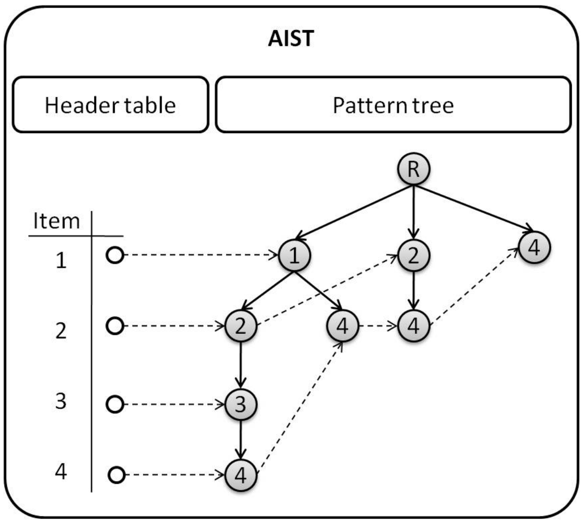


Figure 6.2: AIST scheme. Left: *HeaderTable* with the first links for each item ι. Dashed arrows indicate the *nextNodeList*. Right: *pattern tree*.

As stated above, the required data structure must support the following operations: (1) store patterns of itemsets and (2) quick subset checks, for (a) frequency updates and (b) maximality checks. The first requirement is fulfilled by the canonically ordered nodes, which represent patterns. In the following, the nodes are therefore called pattern nodes (PNode). A PNode representing a pattern with prefix p and the last item ι is denoted by $[p]\iota$, with $[]$ being the empty pattern (root). A PNode possesses several properties:

- Identifier($[p]\iota$): the last item identifier ι
- Level($[p]\iota$): the number of items in $[p]\iota$
- Freq($[p]\iota$): the number of instances covered by $[p]\iota$
- Maximal($[p]\iota$): true, if $[p]\iota$ is frequent and maximal
- NextNode($[p]\iota$): a link to its next node $[p']\iota$
- Parent($[p]\iota$): a link to $[p']\iota' = p$
- Children($[p]\iota$): a set of links to its children

The entry 'nextNode($[p]\iota$)' points to the next PNode ($[p']\iota$) that also includes item ι. Moreover, node $[p']\iota$ is always of lower or equal level than $[p]\iota$: Level($[p]\iota$) $\geq$ Level($[p']\iota$. Following the 'nextNode($[p]\iota$)' links ensures that (1) no larger itemsets and (2) no itemset not sharing at least the last item are examined. This design supports the second requirement. The *headerTable* stores the initial pointer for an identifier's $nextNodeList$, i.e. the PNode with the highest level containing identifier ι:

$$H(\iota) = argmax_{[p]\iota}(level([p]\iota))$$

. The *pattern tree* of an AIST can be defined recursively: it is empty or it consists of a PNode r and an empty or nonempty list of *pattern trees* $t_1, \ldots, t_n$ (list of children). Each PNode r_i is connected to r by an edge (parent link) and r_i's identifier is strictly greater than r's. The root of the AIST is the PNode of level 0, with an empty identifier, thus being the empty pattern. Each PNode represents the pattern that consists of its own identifier and all identifiers of its parents.

6.5.1 InsertPattern

Roughly speaking, the algorithm works as follows (cf. Algorithm 8): For each instance, the maximal occurring pattern (namely the pattern that includes all items from the instance [Algorithm 8 lines 3 to 5]) is inserted into the AIST and if it is frequent, added to the set of MFPs (Algorithm 8 line 11). If not, it serves as a candidate for the MFP search (cf. Section 6.5.2) which is conducted subsequently. Last, all former MFPs are inspected whether they are still frequent and, if not, they again serve as candidates. This implies that subsequently the set of MFPs is updated. As the insertion of patterns is a major step in this procedure, it will now be presented in detail. Each item ι of the occurring pattern P is inserted in the AIST one by one, canonically ordered. Inserting means that either a new node $[p]\iota$ is created, if there is no path $[p]$ with child ι, or that the frequency of $[p]\iota$ is increased by $freq$. Beginning at the root, it is checked whether there already exists an item in the *children's list* of the current node $[p]$ with identifier ι. If this is not the case, a new child $[p]\iota$ is created with the given item ι and prefix $[p]$. Subsequently, the link $nextNode([p]\iota)$ is set, i.e., the node $[p]\iota$ is inserted into the $nextNodeList$ of item ι. Then, the frequency of the pattern is increased by the given frequency count ($freq$, usually one) plus the frequency counts of all supersets (retrieved by method $setNextNode$). Subsequently, all subset frequencies are also increased by $freq$, in order to preserve consistency. This is done in function $UpdateMoreGeneralCounts$. If the current pattern p_i is a MFP, it is added to the set containing all MFPs.

Example: Let the AIST of Figure 6.2 be given where pattern $\{1, 3, 4\}$ is observed. As all items are subsequently inserted into the tree, for item $\iota = 1$ the root (which is the current node) is examined whether it has a child $\iota = 1$. This is true, the current node is set to []1 and its frequency is increased. The next item is $\iota = 3$. Because []1 has no child [1]3, this node is created and added (with frequency 1) to the children list of []1. Then,

Algorithm 8 InsertPattern

```
Require: pattern (canonically ordered) $P$, int freq
 1: active = AIST.root
 2: for each item $\iota \in$ P do
 3:   if !( active has child [p]$\iota$) then
 4:     create [p]$\iota$ and add [p]$\iota$ to active's children
 5:     SetNextNode([p]$\iota$)
 6:   end if
 7:   node = child [p]$\iota$ of active
 8:   node.increaseFrequency(freq)
 9:   UpdateMoreGeneralCounts(node, freq)
10:   if isMFP(node) then
11:     addToMFPs(node)
12:   end if
13:   active = node
14: end for
```

it is inserted in the *nextNodeList* of $\iota = 3$, which is after $[1, 2]3$. It has no successor, because no other pattern containing item 3 is currently stored in the AIST. During this insertion, the exclusive counts of all supersets are derived (cf. the paragraph on SetNextNode), which is then added to the frequency of [1]3. Next, item $\iota = 4$ (level 3) is inserted in the same manner. Its place in the *nextNodeList* of $\iota = 4$ is after $[1, 2, 3]4$ (level 4) and before [1]4 (level 2). Finally, the AIST has an additional path $[1, 3]4$.

SetNextNode

The method *SetNextNode* (Algorithm 9) finds the correct predecessor and successor for a PNode $[p]\iota$ in the *nextNodeList*. Additionally, the frequencies of all supersets are summed up and finally added to $[p]\iota$'s frequency. Starting at the *headerTable*'s entry for item ι, all nextNode links are followed until the first PNode $[p']\iota$ with $Level([p']\iota) \leq Level([p]\iota)$ is reached. Then the pointer of $NextNode([p]\iota)$ is set to $[p']\iota$ and the pointer of $NextNode([p']\iota$'s predecessor) to $[p]\iota$. While following the $NextNodeList$, the visited PNodes $[p'']\iota$ are explored whether they are supersets of $[p]\iota$ and provide a frequency strictly greater than the elements $[p']\iota$ that are already in the set SsP. If this is the case, they are added to the set of $[p]\iota$'s supersets (SsP). (cf. Algorithm 10). When the final position $[p^*]\iota$ of $[p]\iota$ in the $NextNodeList$ is determined, SsP holds all supersets of $[p]\iota$ and their exclusive frequencies, respectively. This means that the count of all occurrences of the pattern has the count of all occurrences of their superpatterns substracted. Thus the counts of subpatterns are not repeatedly counted. The sum is then added to the frequency of $[p]\iota$.

Algorithm 9 SetNextNode

```
Require: AIST, PNode $[p]\iota$
  $SsP = \oslash$
  if Headertable contains $[p]\iota$.Identifier then
    if Headertable($[p]\iota$.Identifier).Level $<$ $[p]\iota$.Identifier.Level) then
      $[p]\iota$.nextNode = Headertable($[p]\iota$.Identifier)
      Headertable($I$.Identifier) = $[p]\iota$
    else
      active = Headertable($[p]\iota$.Identifier)
      if active.isSuperpatternOf($[p]\iota$) then
        update$SsP$(active)
      end if
      while active.hasNextNode() do
        if active.nextNode.Level $>$ $[p]\iota$.Level then
          active = active.nextNode
          if active.isSuperpatternOf($[p]\iota$) then
            update$SsP$(active)
          end if
        end if
      end while
      $[p]\iota$.NextNode = active.nextNode
      active.NextNode = $[p]\iota$
      $[p]\iota$.freq = $SsP$.freq
    end if
  else
    Headertable(I.ID) := $I$
  end if
```

Algorithm 10 UpdateSsP

```
Require: $SsP$, PNode $[p'']\iota$
  if $[p'']\iota$.isSuperpatternOf($[p]\iota$) then
    for each pattern $[p']\iota$ in $SsP$ do
      if $[p']\iota$.freq $<$ $[p'']\iota$.freq then
        $SsP = (SsP \cap [p'']\iota)$
      end if
    end for
  end if
```

Example: Reconsider the example of Section 6.5.1, where the PNode $p = [1,3]4$ was inserted in the AIST. When inserting item $\iota = 3$ (or 4 respectively) a new PNode is created, for which the position in the *NextNodeList*

Algorithm 11 UpdateMoreGeneralCounts

```
Require: PNode I, int freq
  active = I
  while active.hasNextNode do
    active = active.getNextNode
    if active.sharesPrefix == unknown then
      if checkSubsetcontainment(I, active) then
        active.increaseFrequency(freq)
        if isMFP(active) and ! isMFP(PNode) then
          addToMFPs(active)
        end if
        if isMFP(active) and isMFP(PNodeRep) then
          deleteFromMFP(active)
        end if
      end if
    end if
  end while
```

must be found. To do so, the first entry of the *headerTable* for item 3 is retrieved: $[1, 2]3$. Because its level (3) is higher than the level of the node to be inserted (2), the *nextNodeList* is followed until a node with equal or lower level. But because there is no next node, $NextNode([1, 2]3)$ links to [1]3. When inserting item 4, node [1]4 is of lower level. Thus, $NextNode([1, 2, 3]4)$ is updated to $[1, 3]4$ and $NextNode([1, 3]4)$ is set to [1]4. Each visited PNode is examined to determine whether or not it is a superset of the new PNode. This is the case for PNode $p = [1, 2]3$ (and $[1, 2, 3]4$), so that they are inserted in SsP along with their frequency (let this be 3). Imagine that SsP would already comprise a PNode p' which is a superset of p. Then p is only stored if $freq(p') - freq(p) > 0$. The rationale behind this is that only maximal patterns and their frequencies are stored and thus lead to a much smaller SsP. To further illustrate this rationale, consider the insertion of [1]4 of an AIST that already stores the patterns $[1, 2, 3], 4$, $[1, 3], 4$ and $[1, 2], 4$ all with frequency counts 2. This induces that the items 1, 2, 3 and 4 always occur together and that it is sufficient to store the maximal itemset without the loss of counts. Thus only $[1, 2, 3], 4$ is stored in the SsP. However, after the correct place for p is found, all frequencies of the patterns in SsP are summed up and added to the frequency of p. The frequency of $[1, 3]4$ thus equals 4.

UpdateMoreGeneralCounts

The next step in the insertion procedure is to increase the frequency of $[p]\iota$'s subsets (cf. Algorithm 11). By following the $nextNode$ links, one can decide whether or not another pattern in the AIST is a subset of $[p]\iota$: $[p']\iota \prec [p]\iota$. To speed up the subset check, a flag called $sharesPrefix$ is introduced. The intuition behind this flag is that due to the canonical ordering in the tree, subset-checks can be reduced to look-ups, if the parent of a PNode $[p']\iota$ was already visited during the insertion. It can take the values $\{unknown, true, false\}$ and indicates whether the node $[p']\iota$ was already visited during the insertion process. The subfunction *checkSubsetcontainment* returns true, if $[p']\iota$'s $sharesPrefix$ flag is $true$ and $false$, respectively. Then, Node $[p]\iota$ is updated with this result. If $[p']\iota$ was not yet visited (*unknown* is returned) a complete check for subset containment is performed, by subsequent parent comparisons.

Example: Reconsider the tree in Figure 6.2 where pattern $p = \{1, 3, 4\}$ was inserted. Subsequent to its own frequency adaptation, each subpattern's frequency must be updated as well. Here, the $sharesPrefix$-flag of the PNode []1 on the path to $[1, 3]4$ has already been set to *true*. When updating all subsets of pattern $[1, 3]4$, the first PNode considered is [1]4, because it is the first PNode after p in p's *nextNodeList*. The parent of [1]4 is PNode []1 and thus $sharesPrefix$ is set to *true*. This also implies that [1]4 is a subset of $[1, 2, 3]4$, because its prefix [1] is a subset and the last item, 4 is equal. Therefore, its frequency is increased by p's frequency. The second PNode in the list is [2]4 with parent []2. Here the flag $sharesPrefix$ is *unknown* and a full subset check must be performed. It will return false und thus [2]4's frequency is not increased. The last PNode to check is []4 with the root as father where $sharesPrefix$ is also set to *true*. It is therefore a subset of $[1, 3]4$, and again its frequency is increased.

6.5.2 Deleting MFPs

After a new pattern $[p]\iota$ was inserted in the AIST, all MFPs p_r have to be checked if they are still frequent and maximal. If there is a new pattern p'_r, which is more specific, p_r is only deleted from the list of MFPs. However, if the *minsup* constraint for p_r is not fulfilled anymore, it also has to be deleted from the MFPs list, and moreover, subsets of the pattern can still be frequent and are therefore examined concerning frequency and maximality. In general, a pattern of length l has l subsets of length $(l-1)$ that, in this case, must be checked one after another. Ideally, the enumeration of subsets follows a depth first search strategy so that if a frequent subset $[p']\iota$ is found, all further subsets of $[p']\iota$ can be pruned away from the search space. To ensure both, completeness and non-redundancy during the generation process, the search is implemented in the following way: All subsets of $[p]\iota$ are created by deleting the $IdxToDelete$th largest item of $[p]\iota$. If subset $[p']\iota$ is not frequent, all subsets of $[p']\iota$ are explored. Each created pattern is inserted in the AIST with frequency zero falling back to Algorithm 8. Frequency zero assures that no subsets are incorrectly incremented. In contrast, the count of all supersets is added to the pattern. Subsequently, the resulting node can be treated as any other node. However, this generation may also include candidates covering items which are not frequent anymore. This may lead to many unnecessary tests of subsets, because patterns containing infrequent one-patterns cannot be frequent. Thus, excluding even few items can drastically reduce the search space. In order to find such items, each entry in the header table is checked whether it is frequent or not. Then, the initial candidate for each MFP is the MFP itself, where all infrequent items are excluded. This guarantees that all maximal frequent subsets are found without considering useless subsets.

Example: To illustrate this strategy, imagine that a maximal pattern $[1, 3]4$ becomes infrequent, but two of its 2-subpatterns are frequent. The search starts from pattern $p = \{1, 3, 4\}$ (which is also the candidate, because all one-items are frequent) and creates the first pattern to be tested: $\{1, 3\}$ (the last item is removed). Because it is frequent, its subpatterns $\{1\}$ and $\{3\}$ are pruned away from the search space. The next pattern is $\{1, 4\}$ which turns out to be infrequent. Therefore, the next pattern is created: $\{4\}$, which is again frequent. Pattern $\{1\}$ is omitted here, because it would have been enumerated after $\{1, 3\}$. Then, the last pattern's ($\{3, 4\}$) frequency is calculated, and this is again frequent. Because $\{3, 4\}$ is now also maximal frequent, pattern $\{4\}$ is deleted from the MFPs list. As no more patterns are to be evaluated, search stops with a complete set of new MFPs.

6.6 Experiments

6.6.1 Data sets

The first data set is the diagnostic data introduced in Section 3.1.2. To derive different sized data sets, instances were randomly sampled of the whole data set with 170,375 instances.
Second, 18 data sets were generated by the IBM data set generator [2] which is implemented in the (T)ARTool [66]. They cover a different number of patterns ($P \in \{50, 100, 1000\}$), average transaction lengths ($AT \in \{5, 10, 15\}$) and average pattern length ($AP \in \{2, 4, 6\}$).[1] For each P, six data sets (DS) with parameters (P, AT, AP) were generated, assuring that AP is at most half of AT: DS1 = $(50, 5, 2)$, DS2 = $(50, 10, 2)$, DS3 = $(50, 10, 4)$, ..., DS18 = $(1000, 15, 6)$. Each data set holds 10,000 instances and consists of 100 attributes, resulting in differently sparse data sets, with different transaction and pattern lengths. These data sets were used to test the behavior

[1]The parameters were set following the suggestions in the original publication.

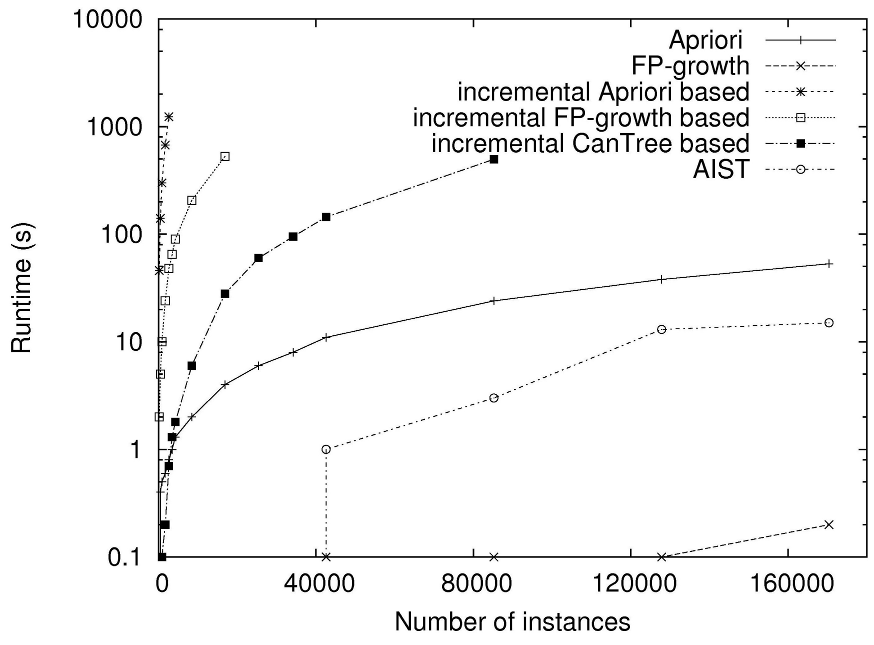


Figure 6.3: Runtime depending on the number of instances of the disease data set with a minimum support of 0.1

of the algorithm for different pattern and transaction lengths for different types of data sets. For scalability experiments, testing the performance for different sizes of input instances, data sets with up to 2M transactions ($P : 20, AT : 5, AP : 3$) were generated, including either $10(I10)$ or $30(I30)$ attributes. Because they were each generated from scratch, they exhibit different distributions of patterns. Note that in all these data sets, an instance is composed of several patterns and also called a transaction.

6.6.2 Empirical evaluation

To evaluate the proposed method, the runtime and the size of the AIST depending on the number of input instances, varying minimum support and different data set characteristics are displayed. Moreover, a comparison to three established methods for frequent itemset mining is drawn. Each of the following experiments was conducted on a 2.2 GHz machine with 1GB RAM. The AIST was implemented in Java.

A first evaluation compares the AIST to other pattern mining algorithms. The runtimes of AIST-based MFP mining, compared to the standard Apriori, FP-growth, LCM[2] (a batch MFP miner) approach and a version of CanTree are presented. A comparison against e.g., Moment, SWIM and

[2]Implementations from C. Borgelt and T. Uno, http://fimi.ua.ac.be/src/

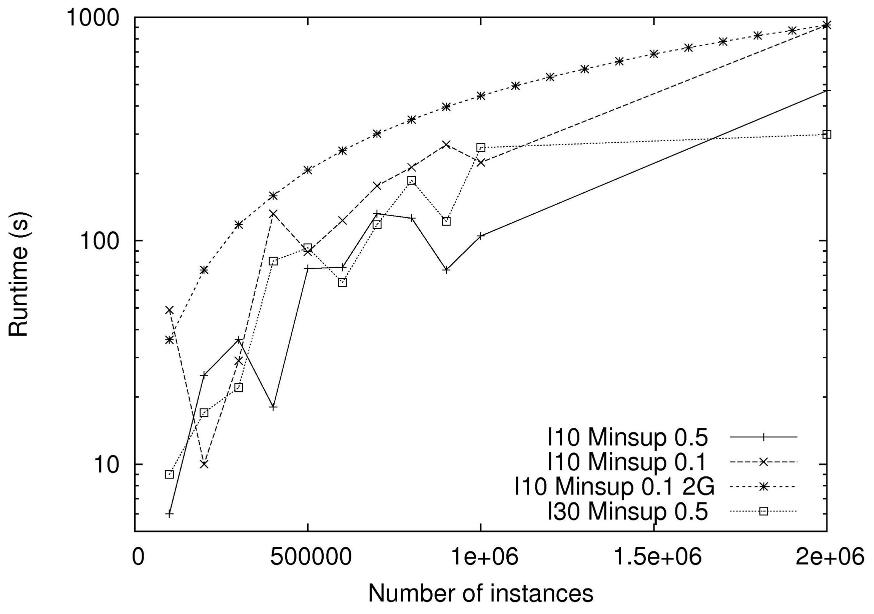


Figure 6.4: Runtime for different data set sizes

DISC is not conducted, because these algorithms were created for a different problem setting where instances may be observed more than once. Figure 6.3 presents the performance of miners for an increasing data set size. As we are particularly interested in an online setting, the right hand side of Figure 6.3 with more than 80,000 instances is most important. Note that runtimes below 0.1s are not displayed in the figure, because of the logarithmic scale. The runtimes show that, while A-priori is slower than the AIST approach, FP-growth and LCM (FP-growth is displayed exemplarily) are more effective than the AIST. However, these three methods only calculate the MFP after all instances were observed. When applying Apriori, FP-growth and LCM to the incremental case, by repeatedly mining the current set of MFPs, they are significantly slower than the proposed approach, despite their very efficient mining and search strategies (LCM is slightly better than FP-Growth). In fact, every batch algorithm that is applied repeatedly, will result in excessive runtimes: If the data set size is 10^l, the number of batch runs also grows exponentially with l, because it has to be repeated 10^l times. Next to that, the data set must be accessed for each run, which can result in additional memory problems if the set becomes large ($l \geq 6$). Thus, a 'repeated batch' approach is not appropriate large data sets. As a last algorithm for the runtime comparison, CanTree [55] was exemplarily chosen from the group of algorithms that incrementally mine updated databases. It outperforms several other algorithms and has a similar underlying data structure as the AIST. Thus, we interpreted the

Minimum support	0.1	0.08	0.06	0.04	0.02
Runtime (s)	~ 1	~ 1	2	7	>60

Table 6.1: Runtimes for different values of minimum support for 42,565 instances of the disease data set

problem setting as an initially empty database where as many updates take place as instances are in the data set (data stream setting). Although it is better than the FP-growth approach, it is still not faster than the AIST implementation. This is mainly due to the repeated creation of projected databases on which it mines the frequent patterns. This is indeed very efficient when only few updates take place, but slows down the mining process for many database updates.
The second evaluation targets the runtime for different numbers of minimum support. Table 6.1 gives the runtime, beginning at a minimum support of 0.1 (larger values always resulted in a runtime below one second). The runtime increases for smaller values for the minimum support, because more MFPs are found, which must be updated. This can be very expensive when large patterns become infrequent.
A third experiment examines the behavior of the algorithm for different data set types. Table 6.2 displays the runtime for the IBM data sets. The best runtimes are achieved for small transaction sizes consisting of few patterns (column 1, 7 and 13). In contrast, for great transactions that additionally consist of many small frequent patterns, the search may be very expensive (column 4, 10 and 16): If patterns in the data are small but the current candidate c is large, a worst case search time of $O(2^{|c|})$ may happen. However, long transactions with few, but large patterns can be mined more efficiently, because much less candidates are inspected. Additionally, these numbers show that denser data sets not necessarily lead to a longer runtime, but that the transaction and pattern lengths restrict the algorithm to either transactions of few large patterns or sparse data sets with small instances. Next, we address the algorithm's scalability. Figure 6.4 shows the runtime for several large data sets and different minimum support values. The two data set $I10$ $Minsup$ 0.1 and $I10$ $Minsup$ 0.1 $2G$ are both data sets with 10 attributes, where the minimum support is set to 0.1. The difference is that the curve $I10$ $Minsup$ 0.1 is composed of different data sets, (for each data set size a new data set was created), while the second curve displays the subsequent runtimes of the 2 million instances (one data set only). The first curve shows how the runtime varies between random data sets, while the second displays the runtime, when more and more instances of the same data set are taken into account. All data sets (except $I10$ $Minsup$ 0.1 $2G$) show that larger data sets may not necessarily result in longer runtimes. Again, this is due to the distribution of patterns in the data

Density	P	50						100					
	AT	5	10		15			5	10		15		
	AP	2	2	4	2	4	6	2	2	4	2	4	6
Runtime (s)		**0**	14	9	**98**	55	48	**0**	18	11	**188**	43	50

Density	P	1000					
	AT	5	10		15		
	AP	2	2	4	2	4	6
Runtime (s)		**1**	18	11	**43**	42	50

Table 6.2: Runtimes for the IBM data sets. Each block represents the data set created using a fixed P. The first line gives the transaction size of the data set, while the second line shows the pattern size (cf. Section 6.6.1). The maximal and minimal values per P-block are indicated in bold. P: Number of Patters, AT: transaction size, AP: pattern size

set. Large, but infrequent patterns as well as many MFPs may increase the runtime. Take curves $I30\ Minsup\ 0.5$ and $I10\ Minsup\ 0.5$ as an example: using more attributes does not necessarily induce a higher runtime (compare $Number\ of\ Instances = 10^5$ vs. $Number\ of\ Instances = 2 * 10^5$). The distribution of frequent patterns and the average size is crucial for the success of the algorithm. However, the runtime of one large data set for successive operation steps ($I10\ Minsup\ 0.1\ 2G$) shows the expected linear runtime for increasing data set sizes.
The last two evaluations examine first, the size of the AIST and second, the corresponding memory requirement for different data sizes and densities. Figure 6.5 gives the successive sizes of the AIST for the IBM data sets and shows the influence of the specific data set parameters AP, AT and P. Note that for a better visualization not every curve is displayed. Moreover, Figure 6.6 shows how the size of the AIST is influenced by different numbers of attributes in the data. Considering P (number of patterns), the size of the AIST grows with more patterns (cf. Figure 6.5). The lines of (5 50 2), (5 100 2) and (5 1000 2) clearly show this relation. This is of course trivial, the more patterns exists in the data, the more nodes must be present to capture them. The opposite is the case for the average pattern size (AP). The larger AP, the smaller the tree becomes: the search for larger patterns is quicker, because fewer candidate patterns have to be created and tested. The graphs (15 100 2), (15 100 4) and (15 100 6) show this relationship. The AIST of (15 100 6) is the smallest. The influence of AT is illustrated by lines (5 100 2), (10 100 2) and (15 100 2). The greater the average transaction size, the larger the AIST. The reason for this lies in the creation of data. A transaction consists of frequent patterns, the larger the predefined size for a transaction, the more patterns are in it. Then, the AIST has to separate

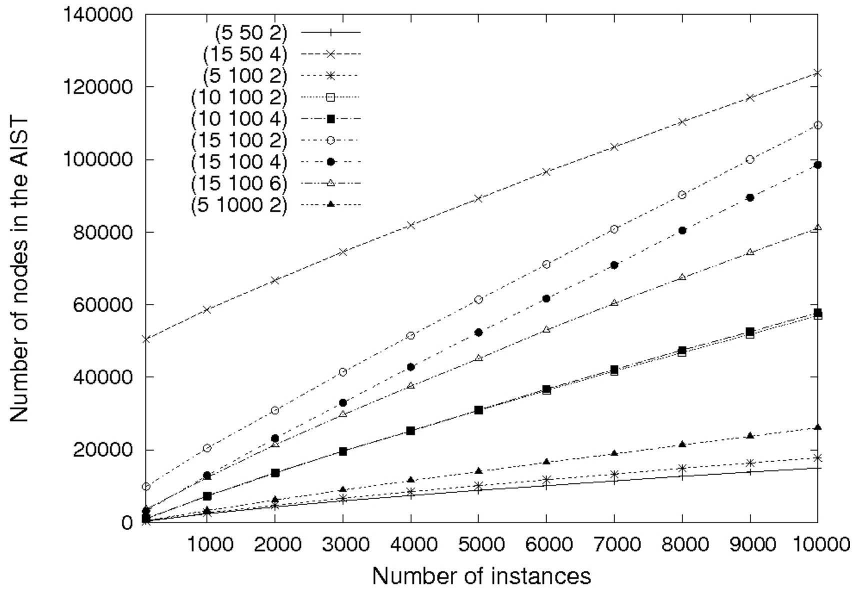


Figure 6.5: Number of nodes of the AIST for different data set parameters. The numbers in brackets correspond to (AT, P, AP) as described in Section 6.6.1. The first point for each line indicates the value for 100 instances.

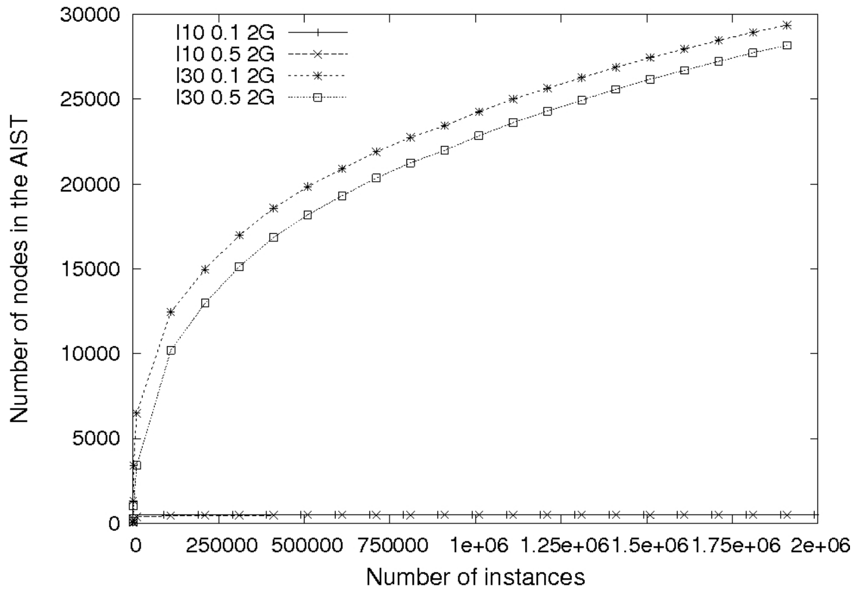


Figure 6.6: Number of nodes of the AIST for 2G data sets and different values of minimum support

these patterns again, which means that small subsets of large transactions have to be found. To do so, many nodes have to be created and evaluated. Now let us consider how the size of an AIST depends on the number of attributes in the data set. Each data set in Figure 6.5 has 100 attributes, while the data sets of Figure 6.6 contain 30 and 10 attributes. The ranges of the AIST sizes vary between $10^4 - 10^5$ for data sets with 100, $10^3 - 10^4$ for 30 and below 10^3 for 10 attributes, respectively. The explanation for this is that despite only a subset of possible patterns is evaluated, the patterns created for a data set vary stronger for many attributes, which increases the AIST. However, data sets with many attributes may nevertheless result in a smaller tree, if the transaction size or pattern size is appropriate. A further question is, whether the size of the AIST will saturate after a certain number of instances and how fast it is growing. At first glance, for small values of P the AIST is not growing extraordinarily for more than 7,000 patterns (cf. Figure 6.5 (5 50 2)). It seems that all important patterns are already incorporated into the tree and only few new patterns are added. Although this limit cannot be observed for the remaining curves, the slope for each curve is decreasing steadily. This also suggests that even for more difficult data sets the AIST will reach a steady level. If only few attributes are present in a data set, saturation is reached very quickly. Figure 6.6 gives an example for that, the final AIST for data set I10 is created after only a few hundred examples.
Considering the initial increase of the curve, one exceptional large one shows graph (15 50 2). It is the steepest 'start' for all presented data sets. In fact, (15 50 4) should be smaller than (15 100 4), because it has less patterns, but the same average transaction and pattern size. A possible explanation for this is that the more patterns were created for a data set, the higher the chance that these patterns overlap (they share items). Then the overlapping parts seem to be frequent maximal patterns in the beginning, because they occur frequently. However, later they will turn out to be infrequent or not maximal. Thus the slope of (15 100 4) and (15 1000 4) (not illustrated) is significantly greater than for (15 50 4). The size of the their AISTs will exceed the one of (15 50 4) after approximately $20{*}10^3$ and $15{*}10^3$ instances, respectively. In general, one can expect that the size of an AIST initially grows quicker for large transaction sizes and small pattern sizes.
Concerning the size of the AIST dependent on the minimum support, Figure 6.6 shows that the AIST increases for lower minimum supports (cf. $I30$ 0.5 $2G$ vs. $I30$ 0.1 $2G$). Candidates are greater if the minimum support is low, which again increases the search space. Then more candidates must be incorporated into the AIST. Although these findings give a hint for the size and the growth rate for an AIST, its final size is of course also dependent on the actual patterns in the data set, the minimum support and the order of the instances. A good example for this is given by (10 100 2) and (10 100 4). Their size is nearly equal, instead of (10 100 2) being larger. The reason

for this can be that most of the patterns of (10 100 2) are detected early or that a better ordering of the instances may lead to a more compact AIST. The last experiment adresses the memory requirement of the AIST (not illustrated). We computed the RAM usage for the AIST-task while processing the $I30$ 0.5 $2G$ data set. The time between RAM allocation increases during the process and the final RAM allocation is 50KB for the whole AIST. Altogether, these experiments lead to the conclusion that the AIST is an appropriate data structure for large frequent patterns and can be applied to different types of data sets. In particular for larger patterns, the savings can be marked. Consider a standard bottom up search strategy that needs to explore nearly the whole pattern space to find the MFPs. For data sets of 100 attributes about $1.27*10^{30}$ frequent patterns must be evaluated. Comparing this number to the sizes of the resulting AISTs, savings of several orders of magnitude can be observed. However, the fewer attributes, the less savings are achieved, for 30 attributes 99.99% of the search can be omitted, while it is only 50% for a data set of 10 attributes. These numbers may be further reduced, when the patterns are small ($|p| < \frac{1}{2} * |I|$). Then a bottom up search strategy may be more appropriate.

6.7 Conclusion

We introduced a data structure that allows for instance-incremental mining for maximal frequent patterns (MFPs). Our setting differs from related settings [18, 63, 20, 32] in that we assume that instances arrive one by one, are not stored, and MFPs of the complete datastream are desired. Combining an IST-like data structure with FP-tree elements in a data structure called AIST (augmented itemset tree), we ensure that the MFPs are up to date for each new instance at any point in time. Nodes (patterns) containing the same item are connected and thus the number of necessary subset checks, while updating the frequencies of observed patterns, is decreased. To evaluate the proposed algorithm, we examined the runtime and the size of the AIST depending on the number of input instances, minimum support values, data set characteristics and compared it to several alternative approaches. We conclude that this method can be applied to very large binary data sets when maximal frequent itemsets can be expected to be large. In future work, we plan to adapt and use the AIST for batch-incremental mining. Second, we want to apply the AIST in an MFP-based clustering (which gave us the main motivation for developing the data structure for this setting in the first place), where MFPs are the representers of clusters. Here, the successive and quick update of MFPs should be useful for efficiently extracting the main properties of clusters for very large binary data sets.

Chapter 7

Using Constraints on the Attribute Level for PRTA Induction (*CSPRTA*)

Like in other machine learning models, background knowledge should be incorporated into the learning process of PRTAs. This is especially useful in the domain of disease pattern analyses. There, physicians or other medical experts often want to concentrate on their individual area of expertise. Moreover, they are only interested in a specific question, e.g. how a certain subgroup of patients behaves or develops. In the domain of PRTA induction, a physician may want to examine how a specific group of patients corresponds to a treatment compared to another group. Therefore, the induction process must be provided with a definition of the group(s) of interest. Then, the resulting paths (or states of the groups) can be evaluated by the physician and differences in the developement of patients are displayed. As such definitions in the medical domain often rely on patient characteristic descriptions that correspond to the level of attributes, the constraints must also be defined and implemented at the attribute level. In this chapter, we first introduce constraints on the attribute level and then present an implementation for the induction of automata by an attribute constrained clustering (ACC). Last, we show how the constraints alter the resulting states of the automaton. Related work and further details about the scalability, extensibility and generality will be given in the next section, while this chaper focuses mainly on the applicability for the detection of automata.

7.1 Main Idea of Attribute Level Constraints

As described above, in the medical domain there is a strong need to describe groups of patients, which is of course done by their characteristics. Furthermore, such a description should be used to guide the expected results of a data evaluation step. Therefore, we propose to use a new type of constraint: attribute level constraints (cf. Section 8) that specify how instances having a certain characteristic should be treated in the PRTA induction process. Frequently, groups of patients having the same characteristics should be found, which induces that all patients with the specific characteristics should be put in the same cluster. Then, a physician can inspect the distribution of the remaining attributes in the resulting cluster. Thus, the first proposed constraint is the *must-link (ml)* constraint that 'collects' all patients that share a specific set of properties and forms a cluster with them. However, there is one problem: How to deal with patients that do not have the specified characteristics but are similar to this set of patients. In the case that only the patients that show the specific characteristics are to be combined, then this condition should also be expressed by a constraint. Therefore, the second proposed constraint is the *must-link-exclusive (mlx)* constraint that ensures that only patients that satisfy the given condition are grouped together. This is of course a very strict condition, but is necessary in the case when only the described group of patients is of interest. A formal description of these two constraints is given in Section 8.2.2 and 8.2.3, so that only an informal description of the problem setting is given here.

7.1.1 Problem Setting

Let $\mathcal{D} = \{(x_1^1, \ldots, x_m^1), \ldots, (x_1^n, \ldots, x_m^n)\}$ be a data set of binary instances ($x_j^i \in \{0,1\}$), m being the number of attributes and n the number of instances. Additionally, a set of attribute-level constraints Φ is provided. A constraint $\varphi \in \Phi$ is in general any formula in propositional logic ranging over the propositional variables x_1 to x_m. In the remainder of the chapter, the values 1 and 0 are interpreted as Boolean truth values, true and false. Although the propositional formulae could take any form in general (also normal forms, for instance), we restrict ourselves to conjunctions in the following. In a formula, an unnegated literal x_i means that the value of that variable has to be 1 in an instance, whereas a negated literal $\neg x_i$ means that the variable has to take the value 0. For each attribute-level constraint $\varphi \in \Phi$ and instance x^i, expression $\varphi(x^i)$ returns whether instance x^i fulfills constraint φ, i.e. the instance is a model [37] for the constraint

$$\varphi(x^i) = 1 \quad \Leftrightarrow \quad x^i \models \varphi \quad . \tag{7.1}$$

If this is the case, we say that instance x^i is in the *scope* of constraint φ. Function $\digamma(\Phi, \mathcal{C})$ returns whether a clustering $\mathcal{C}$ satisfies the specified constraint set Φ. The conditions in which the constraints are satisfied are given in the following sections for each specific constraint type. Moreover, a clustering can be scored with an objective function $f : \mathcal{C} \to \mathbb{R}$ according to its quality. The overall goal is to induce an automaton that includes k

states (that correspond to clusters) for the data set $\mathcal{D}$ such that the k states satisfy the provided attribute-level constraints, $\mathcal{F}(\Phi, \mathcal{C}) =$ true, and that f is maximized.

7.2 Implementation

This section shows how the two proposed constraints *ml* and *mlx* can be implemented in the PRTA induction. As there are already several methods proposed to derive a PRTA from data, this section focuses on the online induction method with a previous PTA construction (cf. Chapter 5).

7.2.1 Implementation of *must-link*

This section shows how the *ml* constraints can be used in the induction of PRTAs. Therefore, the algorithm *SPRTA* is adapted to incorporate constraints (*CSPRTA*), which is introduced in the following (cf. Algorithm 12). To incorporate constraints on the attribute level in the *SPRTA* algorithm,

Algorithm 12 $CSPRTA_{ml}$ (Histories H, double θ)

```
1:  PTA $pta$ = createPTA($H$)
2:  $clustering$ = initConstrAssignment($\Phi_{ml}$)
3:  for all $x_i \in pta$ do
4:    if $\exists \varphi_j \in \Phi_{ml} \mid x^i \vDash \varphi_j$ then
5:      assignInstanceToCluster($c_j$, $clustering[\varphi_j.belongsToCluster()]$)
6:    else
7:      $k = fNC(x_i, C)$
8:      if $k \neq -1$ then
9:        $C[k]$.addInstance($x_i$, $\theta$)
10:     else
11:       $k = C$.addNewCluster();
12:       $C[k]$.addInstance($x_i$, $\theta$)
13:     end if
14:   end if
15: end for
16: validateAssignments($pta$, $C$)
17: for all $C_j \in C$ do
18:   mergeAllInstancesOfCluster( $C_j$)
19: end for
```

two adaptations have to be made. First, after the PTA creation, there is one cluster created to which exactly one constraint belongs: the constraint is assigned to that cluster. The reason for this is that the *mlx* constraint does not allow incorporate instances that do not entail it. To avoid a merge of two states (where one of them does not entail the constraint) and a subsequent reallocation of the corresponding instances, the constraints are assigned to clusters right at the beginning of the induction process. As the implementation for the *ml* constraint should the same, this is also implemented in the same way for these constraints. This also ensures that all instances that entail the *ml* constraint are put in one cluster, but still allows that other instances are grouped with them. Additionally, each cluster is only assigned one constraint. The reason for this is that we assume that the user expects one cluster for each given constraint. Therefore, if a cluster had more than one constraint, then the final number of constrained clusters would be too small. The second adaptation takes place in the online clustering step. Then, each instance is evaluated as to whether it entails one of the given constraints. If this is the case, the instance is immediately placed in the cluster to which the constraint belongs. If an instance entails more than one constraint, it is placed in the first constrained cluster that is found. All instances that are not in the scope of a constraint are placed in the nearest cluster. However, in the last step, the validation of the assignments, only instances that do not entail a constraint are considered for reassignement. This ensures that each instance is placed to its corresponding constraint.

Algorithm 13 $CSIPRTA_{mlx}$ (Histories H, double θ)

```
1: PTA $pta$ = createPTA($H$)
2: $clustering$ = initConstrAssignment($\Phi_{ml}$)
3: for all $x_i \in pta$ do
4:   if $\exists \varphi_j \in \Phi_{mlx} \mid x^i \vDash \varphi_j$ then
5:     assignInstanceToCluster($c_j$, $clustering[\varphi_j.belongsToCluster()]$)
6:   else
7:     $k = fNC(x_i, C)$
8:     if $k \neq -1$ then
9:       while violatesConstraint($x^i$, clustering.get$\Phi_{mlx}$()) do
10:        $clustering = fNC(x^i, clustering, p{+}{+})$
11:      end while
12:      $C[k]$.addInstance($x_i$, $\theta$)
13:    else
14:      $k = C$.addNewCluster();
15:      $C[k]$.addInstance($x_i$, $\theta$)
16:    end if
17:  end if
18: end for
19: validateAssignments($pta$, $C$)
20: for all $C_j \in C$ do
21:   mergeAllInstancesOfCluster( $C_j$)
22: end for
```

7.2.2 Implementation of *must-link-exclusive*

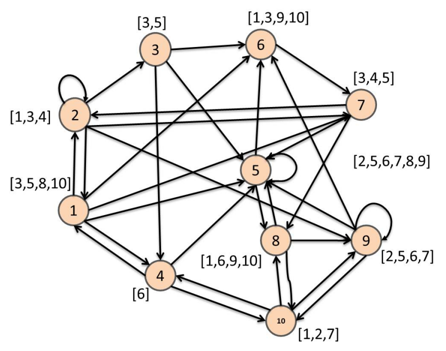
As another interesting constraint, the *mlx* should be incorporated into the *CSIPRTA* algorithm. The difference between the *mlx* and the *ml* constraint is that only instances that are in the scope of the *mlx* constraint may be, and must be, placed in the cluster. On the other hand, a cluster with an *ml* constraint can also have instances outside of its scope assigned to it. Such a constraint creates clusters of instances that share a specific amount of characteristics, which may be interesting to the user. Algorithm 13 shows how such constraints can be used in the induction of PRTAs. In addition to the adaptations of the *ml* constraint, there is another evaluation to satisfy all *mlx* constraints. This is shown in lines 9 to 10. If an instance should be placed in a *mlx*-constrained cluster, this instance must not *violate* the constraint, i.e. it must be in the scope of the constraint. If this is not the case, the instance must also not be placed in the corresponding cluster. Then, the nearest cluster in line is retrieved and again checked whether the instance violates the constraint. This is repeated until a cluster is found in which the instance may be placed. If there is no such a cluster, a new cluster is created and the instance is placed into this one. The remaining part of the algorithm corresponds to the *SIPRTA* algorithm.

7.3 Experiments

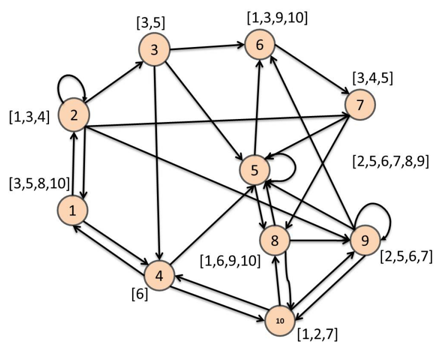
This section focuses on the results of the experiments when using such attribute level constraints for the induction of automata. Moreover, ways to define manageable constraints will be discussed. However, there is one algorithmic adaptation to show the power of constraints. Here, we do not make use of the *validateAssignments* which is mainly used to correct instances that we misplaced in 'wrong' clusters in the beginning of the clustering step, where the cluster-size condition (cf. Section 5.1.3) was not yet as strict as in the end. In contrast, when constraints are included, the user restricts the placement of instances a priori such that a misplacement cannot take place.

7.3.1 Synthetic Constraints

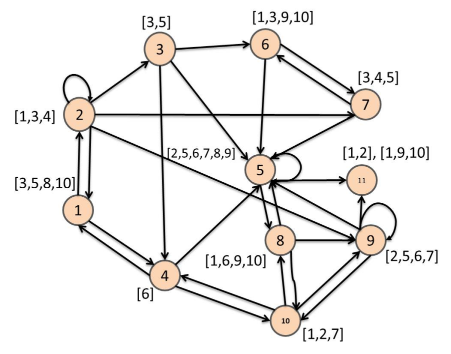
The first experiment addresses the behavior of the synthetic automaton that includes constraints. To show the applicability of the attribute constraints, we first created an automaton without constraints. This automaton is illustrated in Figure 7.1a. It shows that although the algorithm is capable of identifying the correct maximum frequent patterns per state, there are many wrong transitions and some states are blurred due to the inclusion of events that should in fact be place in other states. One example for this problem is state 3 which combines some states from state 1. This is ascribed to the initial phase of the clustering, where event $[3, 5, 8, 10]$ shares two items with



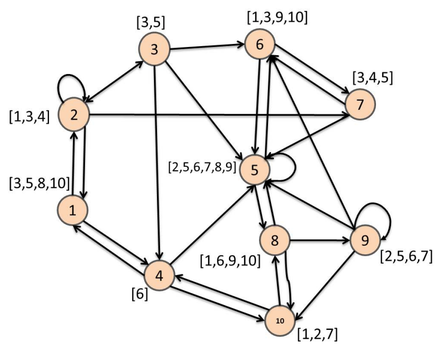
(a) Result for the unconstrained SPRTA algorithm without the validate assignment procedure. The representer (MFP) for each state is shown in brackets.



(b) Result of the CSPRTA algorithm for the synthetic data set and constraint mle_1.



(c) Result of the CSPRTA algorithm for the synthetic data set and constraints mle_1 and mle_2.



(d) Result of the CSPRTA algorithm for the synthetic data set and constraints mle_1, mle_2, mle_3 and mle_4.

Figure 7.1: Resulting automata by a stepwise inclusion of the constraints

events [3,5] that are placed in a small cluster. Then $[3, 5, 8, 10]$ may therefore be assigned to the same cluster. The same is true for state 7. Nevertheless, attribute constraints can be used to address this problem. Therefore, we first introduce a constraint that collects all events $[3, 5, 8, 10]$ into one state: $mle_1 = (3 \wedge 5 \wedge 8 \wedge 10)$ and thus separates states 3 and 1. Figure 7.1b shows that the inclusion of this single constraint erases four erroneous transitions and also makes the distinction of profiles better. But still, there is one problem with state 6 – one transition to state 5 is missing (as well as the transition to state 7). Moreover, the distinction between state 9 and state 10 is difficult when considering the profiles, which is due to the fact that there are some events clustered to state 9 that should have been with state 5. To fix these two problems we introduce three more constraints:

- $mle_2 = (1 \wedge 3 \wedge 9 \wedge 10)$
- $mle_3 = (2 \wedge 5 \wedge 6 \wedge 7 \wedge 8 \wedge 9)$

- $mle_4 = (2 \wedge 5 \wedge 6 \wedge 7)$.

They are included in the CSPRTA algorithm in two steps. Figure 7.1c shows that this was quite a good choice, because the missing transitions from state 6 appear again as well as their correct direction. However, using the *mlx* constraint also comes with a drawback here: an additional state appeared that comprises events $\langle 1, 2,^* \rangle$ and $\langle 1,^*, 9, 10\rangle$ which are deviations of states 10 and 6. [1,9,10] cannot be included in state 6 because of the *mlx* constraint and probably by accident [1,2] is also included in this state. However, the frequency counts of this node as well as the transition probabilities would suggest to the user that this node is an exception. Now mlx_3 and mlx_4 should be included in the model, because state 9 holds some events of state 10. Note that the single inclusion of mlx_4 would not result in a correct model, because then, states 9 and 5 would be combined into a single one as the profile of state 9 is a subset of state 5. Therefore, a constraint that also combines the events of state 5 is needed. Figure 7.1d shows the result of this operation. The structure of the automaton is nearly correct, but still there are three remaining transitions that are wrong. One experiment to test whether it is possible to leave out, e.g. mlx_2 showed that this is not possible because then states 8 and 6 would be placed in one cluster. Summarizing these experiments, one can see that constraints may improve the final model. Interestingly, they mainly do not influence the resulting profiles or states' representers but rather the transitions, by avoiding erroneous assignments of events during the beginning phase of the SPRTA clustering step. Second, literals that can express that an attribute is not present should be used to constrain 'small' states. This would be very helpful to identify states having only one or two attributes in their profile, like e.g., state 4. If constraints would be used that only include few positive literals ('small' constraints) then too many events would be clustered together and the final result would be misleading. Therefore, the creation of constraints must be done very carefully. Then, for such small automata it is not useful to incorporate must link constraints, although they are capable of summarizing all necessary instances: In an early timepoint of the clustering process, there may be other nodes that should not be put in that cluster, but are, because they are still just similar enough. Finally, there are still some transitions that are not yet correct. This shows that constraints may guide the induction process, but are not capable to guarantee correct results.

7.3.2 Yeast Constraints

This section shows how the yeast automaton can be constrained. First, Figure 7.2 shows the resulting automaton, when SPRTA is run without the *validateAssignments* procedure. Although the cyclic structure is of course visible, first note that two states are highly similar (state 3 and state 5). Secondly, there is an additional transition from state 5 to state 3 (dashed

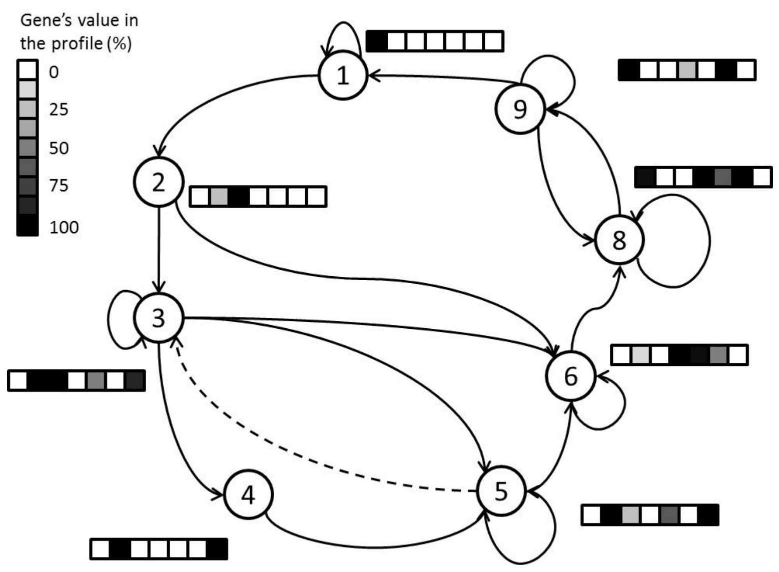


Figure 7.2: Resulting automaton when no constraints are used. The dashed transition destroys the unique 'direction' of the process.

transition). The main problem that arises is that the overall 'flow direction' of the automaton is destroyed here. While cycles of degree one (a node is connected to itsself) show that a cell can stay in a certain stage for a while, this transition means that the cell can 'return' to a stage, which is not true. Besides, the short cut from state 2 to state 6 also makes no sense because it skips the evident coexpression of gene 2 and gene 7. This is due to the fact that the algorithm is not able to distinguish the events that share a subset of genes 2, 3, 5 and 7. Essentially, states 3 and 5 share the same itemsets, so that such wrong transitions occur. However, the situtation is more complicated, as genes 3 and 5 are coexpressed in timepoints 23 and 24 (cf. Figure 3.2), although the remaining timepoints indicate that gene 5 should follow gene 3 and that they are rarely coexpressed. The same problems can be found for states 8 and 9. Therefore, we propose to constrain the automaton using two constraints: The first one should make the distinction between states 3 to 6 easier, while the second one should introduce knowledge about the gene expression sequence of genes 1, 4 and 6:

- $mle_1 = (1 \wedge 4 \wedge 6)$
- $mle_2 = (2 \wedge 3 \wedge 7)$

Running CSPRTA with these two constraints results in an automaton that also has eight states, but is more reasonable in the way the transitions are located. First, there are no more backward transitions and second, the states' profiles are much easier to distinguish. State 5 now shows that genes

	genes						
	1	2	3	4	5	6	7
Mfp 1	●						
Mfp 2			●				
Mfp 3u		●	●		●		●
Mfp 3c		●	●				●
Mfp 4		●					●
Mfp 5		●			●		●
Mfp 6				●	●	●	
Mfp 7	●			●	●	●	
Mfp 8	●					●	

Table 7.1: Maximal frequent patterns for the yeast data set; with and without constraints. Note that Mfp4 was identified for the constrained automaton (c), while Mfp3 was induced in the unconstrained (u) one.

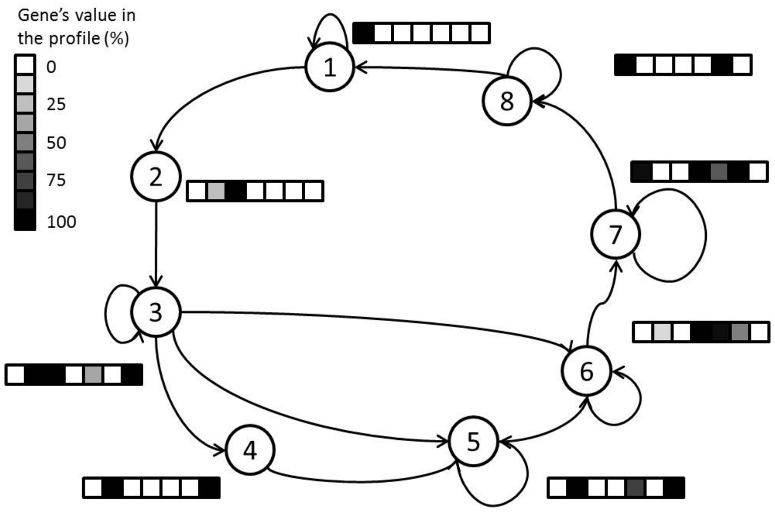


Figure 7.3: Result of the constrained yeast automaton

Element	Name	Interpretation
WBC	ratio of white blood cells	
HGB	Haemoglobine	
HCT	ratio of solid blood components	
RBC	ratio of red blood cells	
PLT	ratio of blood plates	
CRP	C-reactive protein	inflammation indicator, risk factor for heart attacks (HA)
LDH	Lactat-Dehydrogenase	organ destruction indicator
CRE	Creatinine	bladder problems indicator
GOT	Aspartat-Aminotransferase	liver problems, acute oxygen undersupply, heart attack
GPT	Alanin-Aminotransferase	liver problems
UN	blood urea nitrogen	protein degradation, (HA) ind.

Table 7.2: Elements of the blood count and their interpretation

2,5 and 7 often occur together after genes 2,3 and 7 occured together. State 4 indicates that gene 3 is not necessarily replaced immediately by gene 5 but that there may also be a timespan, where neither gene 3 nor gene 5 is active. Then, states 7 and 8 also show that genes 1,4 and 6 frequently occur together before gene 6 is switched off. Table 7.1 displays the maximal frequent patterns of both automata. It is very interesting that the SPRTA approach and the CSPRTA approach identify nearly the same representers per cluster, but still CSPRTA manages to identify the transitions better.

7.3.3 Hepatitis Constraints

The Hepatitis data set is a rather hard one for the given algorithm of automata induction. This is due to the fact that it is dense, i.e. each instance consists of many attributes. When running the induction algorithm without constraints, one also sees that there are many clusters having several maximum frequent patterns as representers. Therefore, the constraints should focus on the combination of interesting similar patterns to first reduce the number of resulting clusters and to focus on a special kind of patients. The attributes that are used in this data set are again summarized in Table 7.2. One interesting question for a physician would be to compare patients with a bad blood count to ones having a normal blood count. The frequency distribution of the values shows that there are quite a few patients that have a normal blood count (normal values of WBC, HGB, HCT, RBC and PLT), but there are also patients that have values that are too low, e.g. HGB. In contrast, excessively high values values are rare, cf. for example HGB and HCT, there are only 2 cases where they are too high. Therefore, we define a state

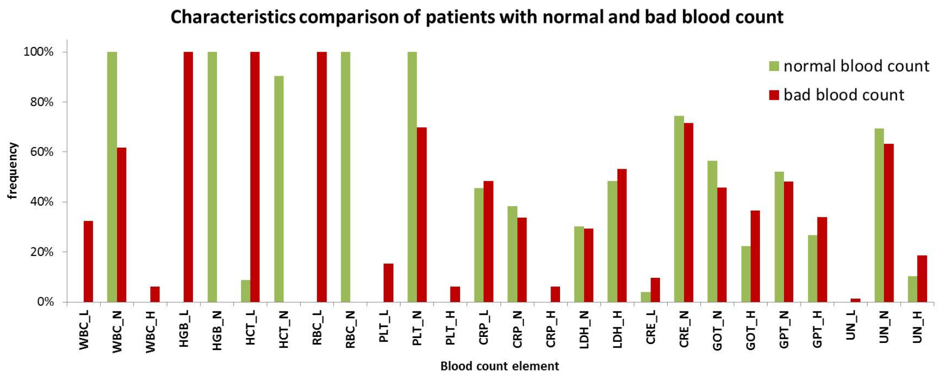


Figure 7.4: Comparison of patients having a normal (group 1) and a bad blood count (group 2). Red bars indicate the frequency of the blood count elements for group 2, while the green bars show the distribution for group 1.

'bad blood count' (BBC) that summarizes patients with low values for HGB, HCT and RBC (PLT and WBC were left out because of their low frequency) and a state 'normal blood count' (NBC): $mle_1 = (HGB_L \wedge HCT_L \wedge RBC_L)$, $mle_2 = (HGB_N \wedge HCT_N \wedge RBC_N \wedge WBC_N)$. Figure 7.4 shows the result of this comparison analysis, i.e. the profiles of the resulting constrained states are shown. Both states contain many events (NBC: 544, BBC: 709) and are highly connected to other states (34 and 35 outgoing and 33 / 38 incoming transitions). However, patients with BBC also show abnormal blood count values for the remaining elements. Notably, they are the only ones were high (low) values for CRP (UN) can be observed. Moreover, BBC-patients also suffer from a high GOT-value more frequently than NBC-patients, which shows an acute oxygen undersupply, which may, e.g. cause a heart attack. This leads to another interesting question: how do patients look that have a high risk of a heart attack (the GOT and UN values are high). Therefore, we designed another constraint that captures patients with these properties: $mle_3 = (GOT_H \wedge UN_H)$. The resulting state combines 136 events and has 33 (34) out (incoming) transitions. These patients also show a high value of LDH that also indicates some kind of organ damage. Interestingly, the CRP-value is probably low, which shows that there are no current inflammations.

Although these analyses give a good overview of the comorbidities of the patients, another interesting analysis would be to observe their progression. However, this data set remains unsuited for this problem, because the resulting states are all highly connected among each other so that no obvious paths can be seen. Nevertheless, this data set gives a very good impression of how medically interesting constraints can be formed and inspected by using the CSPRTA methodology.

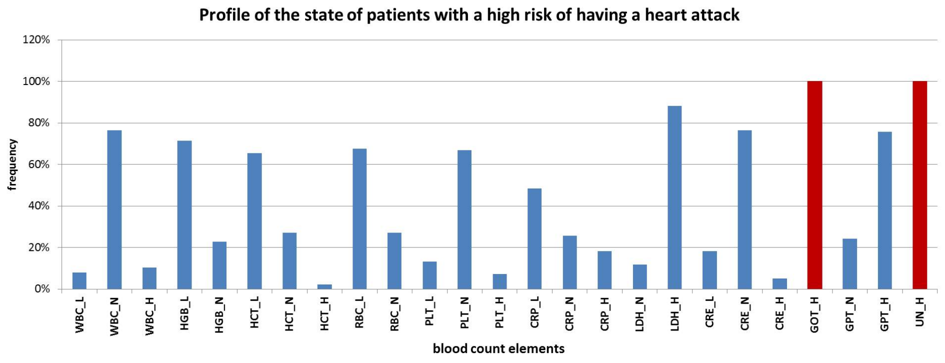


Figure 7.5: Profile of patients having a high risk of heart attack (UN and CRP are high and colored in red)

7.4 Conclusion

In this chapter we showed how the two attribute level constraints *must-link* and *must-link-exclusive* can be incorporated and used in the SPRTA environment. Therefore, we adopted the implementation details presented in Chapter 8 to fit the SPRTA algorithm. Most importantly, we showed in the experimental part, how to design constraints, their benefits and shortcomings on three different types of data. Using very specific constraints, i.e. constraints that consist of many literals, seems to lead to understandable results. Including only a small number of literals instead, may blur the resulting automaton. Besides, using *must-link-exclusive* constraints is easier than the *must-link* constraints, as the user can directly form expected states. Moreover, attribute constraints seem not to work for the correction of the states' profiles, but work well for the correction of transitional errors. Altogether, including attribute level constraints may be well suited for the evaluation of specific groups of events, but still the constraints must be carefully designed.

Chapter 8

Attribute Constrained Clustering

Clustering is a frequently used method to analyze and segment data sets and for many applications, users wish to include their knowledge about the expected result. Including such background knowledge in the form of constraints may improve the clustering result or even drastically reduce the runtime. This is why researchers addressed the problem of defining and evaluating different types of constraints [100] in various cluster algorithms in the past decade. The first idea of such constraints was to restrict the instances that must (*must-link*) or may not (*cannot-link*) be grouped into a cluster. This type of constraint is called instance-level constraint. However, background knowledge is not only present on the instance level. There also is domain knowledge that can be expressed without even knowing specific instances of a given data set. Therefore, we introduce a new type of constraints: constraints on the attribute level, i.e. the *properties* of the instances in a cluster are described instead of the *relationships* between the instances themselves. We call this type of constrained clustering *attribute-constrained clustering (ACC) [77]*.[1] This type of constraint has one additional benefit compared to the known instance-level constraints: the representation of background knowledge is much more compact when given on the attribute level and thus addresses one important factor of constrained mining [24]. Although the approach is not restricted to it, we focus on constraints that specify how instances with given attribute characteristics must be combined. In summary, the contributions of this chapter are as follows:

- First, we introduce *must-Link* and *must-Link-Excl* constraints on the attribute level in Section 8.2.
- Second, the direct incorporation into a popular clustering algorithm (k-Medoids) is presented (cf. Sections 8.2.2 and 8.2.3).

[1]This is an extension of previous work [11].

- Moreover, constraints on the attribute level are shown to be more compact (Section 8.3.3) than on the instance level and experimentally demonstrated to be useful when the background knowledge can be expressed by instance characteristics (Section 8.3).
- Last, the relation to the well known instance-level constraints is established for each type of attribute-level constraint (cf. Sections 8.2.2 and 8.2.3).

8.1 Related Work

The first clustering constraints in the literature were so-called *must-link* and *cannot-Link* constraints, which were incorporated into several algorithms, e.g. k-Means [100] and divisive clustering [50]. One main goal was to specify the types of constraints and to show how heuristics can be used to find a near-optimal clustering solution that satisfies all constraints. This was investigated, e.g. in the context of the k-Means algorithm [24, 25]. Especially the use of *must-Link* and *cannot-Link* constraints as well as cluster distance constraints were evaluated as to whether there exists a partition that satisfies the given constraints and if it is actually possible to find it. Moreover, the scalability of *must-Link* and *cannot-Link* constraints and the modification of clustering approaches using the nearest-representative property were studied [89]. Although it is acknowledged that attribute-level constraints are important [101], they have not yet been explored extensively. One approach applied one type of attribute-level constraint to co-clustering [71, 70]. The proposed algorithm uses local bi-sets that are clustered to a final solution. In the clustering procedure, a k-Means-like process evaluates local anti-patterns for their correctness. The proposed constraint is not defined in general, but for specific attributes only, and may only be extended to pairs of attributes. In particular, the constraint can only be applied if the corresponding attribute is ordered, like a time stamp for the instance. Moreover, the attribute must be numeric and thus, the constraint may be defined on an interval or non-interval scale. A similar constraint setting is used to also define range-constraints on attributes in a cluster [23]. Again, the constraint is applied on a single numeric attribute and the constraints are not associated, i.e. they are independent of each other. To sum up, work in this area so far introduces constraints on the attribute level, but only links pairs of attributes at most, and not sets in general. Additionally, the constraints were incorporated into co-clustering, while this chapter aims for regular clustering, as we cannot guarantee to find optimal co-clusters. Another type of constraints are cluster-level constraints [64]. The basic idea is to select clusters for a clustering out of a set of predefined possible clusters. The constraint then can define, e.g., which clusters must be incorporated in the clustering or how large, complete or disjoint clusters may become. A last approach to drive the clustering process is to define that patterns in general (e.g., frequent itemsets) should be found in the final clusters [85]. Although this approach also describes characteristics of clusters, no data set specific knowledge can be provided to the mining algorithm. In summary, attribute-level constraints are restricted in their expressiveness so far. In this chapter, we extend attribute-level constraints to a general form such that each attribute may be involved in a constraint. Additionally, we specify problem settings that benefit from such constraints.

8.2 Constraints on the Attribute Level

This section first introduces the necessary notation. Then, the proposed constraints are formally defined including a running example. Subsequently, variations of the k-Medoids algorithm that incorporate the attribute-level constraints directly are presented.

8.2.1 Problem Description

Let $\mathcal{D} = \{(x_1^1, \ldots, x_m^1), \ldots, (x_1^n, \ldots, x_m^n)\}$ be a data set of binary instances ($x_j^i \in \{0, 1\}$), m being the number of attributes and n the number of instances. Additionally, a set of attribute-level constraints Φ is provided. A constraint $\varphi \in \Phi$ is in general any formula in propositional logic ranging over the propositional variables x_1 to x_m. In the remainder of the chapter, the values 1 and 0 are interpreted as Boolean truth values, true and false. Although the propositional formulae could take any form in general (also normal forms, for instance), we restrict ourselves to conjunctions in the following. In a formula, an unnegated literal x_i means that the value of that variable has to be 1 in an instance, whereas a negated literal $\neg x_i$ means that the variable has to take the value 0. For each attribute-level constraint $\varphi \in \Phi$ and instance x^i, expression $\varphi(x^i)$ returns whether instance x^i fulfills constraint φ, i.e. the instance is a model [37] for the constraint

$$\varphi(x^i) = 1 \quad \Leftrightarrow \quad x^i \models \varphi \ . \tag{8.1}$$

If this is the case, we say that instance x^i is in the *scope* of constraint φ. The task is, to group the instances $x^i \in \mathcal{D}$ into a clustering that satisfies the given attribute-level constraints. A clustering $\mathcal{C}$ consists of k clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$, where each cluster is a set of instances $C_i \subseteq \mathcal{D}$. Function $\digamma(\Phi, \mathcal{C})$ returns whether a clustering $\mathcal{C}$ satisfies the specified constraint set Φ. The conditions in which the constraints are satisfied, are given in the following sections for each specific constraint type. Moreover, a clustering can be scored with an objective function $f : \mathcal{C} \rightarrow \mathbb{R}$ according to its quality. The precise definition of this function depends on the application domain. As an example one could consider the intra or inter cluster distance or the silhouette coefficient [48]. Although constraints can be used in any clustering scheme, this chapter presents how they can be included in the k-Medoids [48] algorithm. K-Medoids was chosen because it is a standard

Algorithm 14 k-Medoids (Dataset D, int k)

```
1: medoidsChange = true
2: clustering = initializeClusterMedoid($k, D$)
3: while medoidsChange do
4:     for each $I_i \in D$ do
5:         clustering = assignToNearestCluster($I_i$, clustering)
6:     end for
7:     medoidsChange = calculateNewMedoids(clustering)
8: end while
```

clustering algorithm that is fast, easy to modify and because it was frequently considered in the field of constrained mining. Algorithm 14 shows the corresponding pseudocode. Summarizing, the overall goal is to find k clusters for the data set $\mathcal{D}$ such that they satisfy the provided attribute-level constraints, $\mathcal{F}(\Phi, \mathcal{C}) =$ true, and that f is maximized.

8.2.2 Must-Link

The first constraint, *must-Link* (*ml*), describes which instances must be clustered together due to their attribute characteristics. It is defined via the logical formula given in Equation (8.2).

$$\varphi = ml(x_1 \wedge \ldots \wedge x_m) \tag{8.2}$$

A clustering $\mathcal{C}$ satisfies a set of *must-Link* constraints Φ_{ml}, if and only if all instances that are in the scope of a specific constraint $\varphi_i \in \Phi_{ml}$ are grouped into one cluster:

$$\begin{aligned} &\mathcal{F}(\Phi_{ml}, \mathcal{C}) = true \Leftrightarrow \\ &\forall \varphi_i \in \Phi_{ml} \;\; \exists C_l \in \mathcal{C} \;\; \forall x^k \in \mathcal{D} : x^k \models \varphi_i \rightarrow x^k \in C_l \end{aligned} \tag{8.3}$$

The *must-Link* constraint specifies that instances that are in the scope of a constraint φ_i must be grouped in the same cluster C_l. Additionally, instances that are not in the scope of the constraint but are nearest to it, may also be grouped in that cluster. So, each cluster C_l contains the instances satisfying the constraint and additionally the closest instances:

$$C_l = \{x^k \mid x^k \models \varphi_i\} \cup \{x^j \mid \min_{C \in \mathcal{C}} d(x^j, C) = C_l\}$$

Then, the *must-Link* φ_i is *related* to that cluster C_l. This relation is the key point to include attribute-level constraints in the clustering process. Each

instance can be checked whether it is in the scope of the *ml* constraint related to the clustering. An instance is in the scope of a *must-Link* constraint, if it is a model for the constraint (cf. Formula 8.1), i.e. it has the necessary attribute setting.
Note that this type of constraint can also be related to the domain of instance-level constraints. More precisely, every attribute-level constraint φ_i induces, for pairs of instances x^k and x^j within the scope of the constraint, a set of instance-level must-link constraints $\{mustLink(x^k, x^j) \mid x^k \models \varphi_i \wedge x^j \models \varphi_i\}$. However, the *ml* constraint is not transformed into an instance level constraint but directly incorporated into the clustering process. Algorithm 15 shows the two main modifications necessary for the incorporation of *must-Link* constraints in the *k*-Medoids algorithm. The first adaptation ensures that no initial medoid is in the scope of more than one *must-Link* constraint (line 2), which guarantees that each constraint is only related to one cluster. The second modification takes place in the clustering step. If an instances is in the scope of exactly one *ml* constraint φ_j (line 6), which is already assigned to a cluster, then the instance will immediately be placed in the corresponding cluster. If several constraints apply, the one with the nearest corresponding medoid is chosen for assignment. If the instances are

Algorithm 15 *ml*-*k*-Medoids (data set $\mathcal{D}$, int k, *ml* constraints Φ_{ml})

```
1: $medoidsChange$ = true
2: ($clustering$, $assignedConstraints$) =
   initializeClusterMedoid($k, \mathcal{D}, \Phi_{ml}$)
3: while $medoidsChange$ do
4:   for all $x^i \in \mathcal{D}$ do
5:     for all $\varphi_j \in assignedConstraints$ do
6:       if $x^i \vDash \varphi_j$ then
7:         assignToConstrainedCluster($x^i$, $clustering$, $\varphi_j$)
8:         belongstoCluster($x^i$) = true
9:       end if
10:    end for
11:    if ! belongsToCluster($x^i$) then
12:      $clustering$ = assignToNearestCluster($x^i$, $clustering$)
13:      for all $\varphi_k \in unAssignedConstraints$ do
14:        if $x^i \vDash \varphi_k$ then
15:          assignConstraintToCluster($\varphi_k$, $clustering$)
16:        end if
17:      end for
18:    end if
19:  end for
20:  $medoidsChange$ = calculateNewMedoids($clustering$)
21: end while
```

not in the scope of a related constraint (line 12), the usual assignment is conducted. Last, each constraint that is not yet assigned to a cluster is inspected whether the instance is in its scope. If this is the case, the constraint is related to that cluster. In the following iterations, all instances that are in the scope of this constraint are then grouped into that cluster. Consider the following example: Let the database $\mathcal{D}$ comprise the following instances: $\mathcal{D} = \{x^1, \ldots, x^5\}$, where $x^1 = (1,1,0,1,0)$, $x^2 = (1,1,0,0,0)$, $x^3 = (0,1,0,0,0)$, $x^4 = (0,0,1,0,1)$ and $x^5 = (0,0,0,0,1)$. These instances are to be grouped into two clusters ($k = 2$) and one *must-Link* constraint is provided: $\varphi_1 = ml(x_1 \wedge x_2)$. Let instances x^1 and x^4 be the initial medoids of the clustering. As instance x^1 is already in the scope of the *must-Link* constraint, it is related to cluster 1. Due to the constraint, instance x^2 is also grouped into cluster 1. The remaining instances are assigned without considering the constraint. Instance x^3 belongs to cluster 1, instance x^5 to cluster 2. The resulting clustering $\mathcal{C}$ is thus $\mathcal{C} = \{\{x^1, x^2, x^3\}, \{x^4, x^5\}\}$. This constraint can be used to group instances that share one or several properties, no matter how the remaining attributes look like.

8.2.3 Must-Link-Excl

The second constraint, *must-Link-Excl* (*mlx*), is a modification of the *must-Link* constraint in a way that it not only defines which instances must be grouped together but moreover, which instances must not belong to this group. Equation 8.4 shows when a clustering $\mathcal{C}$ satisfies such a constraint set Φ_{mlx}. This is the case, if only instances that are in the scope of a *mlx*-constraint $\varphi_i \in \Phi_{mlx}$ are combined into a cluster.

$$\begin{aligned}
&\mathcal{F}(\Phi_{mlx}, \mathcal{C}) = true \Leftrightarrow \\
&\forall \varphi_i \in \Phi_{mlx} \;\; \exists C_l \in \mathcal{C} \;:\; \forall x^k \in C_l : x^k \models \varphi_i \wedge \\
&\nexists C_j \in \mathcal{C} : C_j \neq C_l : \exists x^m \in C_j : x^m \models \varphi_i
\end{aligned} \tag{8.4}$$

Again, this constraint can also be expressed by a set of instance-level constraints, more precisely, a set of *must-Link* and *cannot-Link* constraints:

$$\begin{aligned}
&\{mustLink(x^k, x^j) \mid x^k \vDash \varphi_i \wedge x^j \vDash \varphi_i\} \cup \\
&\{cannotLink(x^k, x^j) \mid x^k \vDash \varphi_i \wedge x^j \nvDash \varphi_i\}
\end{aligned}$$

The first part is again a *must-Link* constraint which specifies that instances which possess the given characteristics must be grouped together. However, for all instances that are not in the scope of the constraint, a *cannot-Link* constraint is induced. In this case the instance cannot be grouped to the cluster, it *violates* the constraint. Note, that although each attribute-level constraint can be transformed into an instance-level constraint, the opposite is not the case.

Again, the attribute-level constraint *mlx* is used in the clustering process:

all instances which are in the scope of the constraint are clustered together. In other other words, each cluster C_l contains exclusively the instances satisfied by the constraint φ_i: $C_l = \{x^k \mid x^k \models \varphi_i\}$. The scope of a *mlx* constraint is defined as an *ml* constraint. Algorithm 16 shows how to include the *mlx* attribute-level constraint in the k-Medoids algorithm. The

Algorithm 16 *mlx*-k-Medoids (data set $\mathcal{D}$, int k, *mlx* constraints Φ_{mlx})

```
1: $medoidsChange$ = true
2: $clustering$ = initializeClusterMedoid($k, \mathcal{D}, \Phi_{mlx}$)
3: **while** $medoidsChange$ **do**
4:    **for all** $x^i \in \mathcal{D}$ **do**
5:       $p = 1$
6:       **if** $\exists \varphi_j \in \Phi_{mlx} \mid x^i \models \varphi_j$ **then**
7:          assignInstanceToCluster($c_j$, $clustering[\varphi_j.belongsToCluster()]$)
8:       **else**
9:          $clustering$ = findNearestCluster($x^i$, $clustering$, $p$)
10:         **while** violatesConstraint($x^i$, clustering.get$\Phi_{mlx}$() ) **do**
11:            $clustering$ = findNearestCluster($x^i$, $clustering$, $p$++)
12:         **end while**
13:      **end if**
14:   **end for**
15:   $medoidsChange$ = calculateNewMedoids($clustering$)
16: **end while**
```

initialization is dependent on the given constraints. As each cluster with a related *must-Link-Excl* constraint can only include instances that are in its scope, each *mlx* constraint must belong to a separate cluster. Thus, during the initialization, each constraint is related to a cluster that consists of a medoid that also must be in the scope of the constraint. Then, each remaining instance (from line 3) is evaluated whether it is in the scope of a *mlx* constraint φ_j. If this is the case, it is assigned to the corresponding cluster. The main adaptation to include *must-Link-Excl* constraints is shown in line 9 to line 12, where the assigment of an instance to a cluster is not only dependent on the distance to its medoid but also if its assignment would *violate* the constraint. While this is the case, the second (or further) nearest cluster is chosen for assignment. If no such an assigment can be found, the algorithm stops. To illustrate this type of constraint, again consider database $\mathcal{D} = \{x^1, \ldots, x^5\}$ from the previous example. The clustering size is set to two ($k = 2$) and the provided constraint is $\varphi_1 = mlx(x_5 \wedge \neg x_3)$. Let instance x^1 and x^5 be the initial medoids of the clustering. Instance x^5 is in the scope of the *must-Link-Excl* constraint, therefore this constraint is related to cluster two. Instance x^4, which is similar to instance x^5, cannot be grouped in cluster two because it violates the constraint ($x_3^4 = 1$). Thus it must be assigned to cluster one. Last, all other examples are assigned to the clusters without considering the constraints because they are not in its scope. Instance x^3 and x^2 belong to cluster 1. The resulting clustering $\mathcal{C}$ is thus $\mathcal{C} = \{\{x^1, x^2, x^3, x^4\}, \{x^5\}\}$.

8.2.4 Convergence of Attribute Constrained Clustering

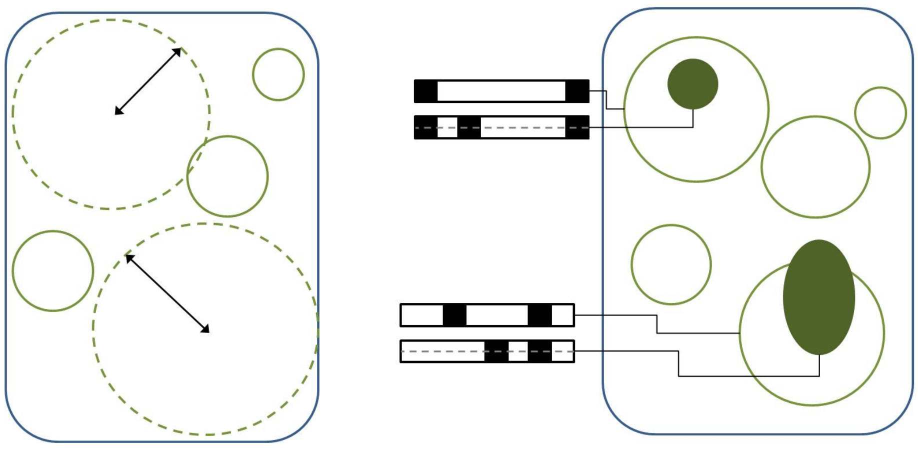
Another question when using this type of attributes is whether the proposed algorithms terminate. As k-Medoids is proven to converge (condition 1) an evaluation of the inclusion of the *ml* or *mlx* constraints is necessary. This can be shown by contradiction: If there is an instance that is in the scope of a constraint φ_i, φ_i is related to the cluster to which the instance is assigned. If there is no such instance, the constraint is not considered for the clustering process. Then, all instances that are also in the scope of φ_i are as well assigned to this cluster (in the same iteration or at most one iteration later). Thus for all instances in the scope of φ_i the algorithm always finds a solution (even in at most two iterations). Moreover, the assignment of theses instances remains fixed for the rest of the clustering process. The remaining instances are clustered via the standard k-Medoids procedure. If there was no possibility to find a solution for the remaining instances, then the standard k-Medoids would not terminate. However, this is a contradiction to condition 1 and thus, the algorithms must terminate.

8.2.5 Extension of Attribute Constrained Clustering

Although the constraints in this work are defined and applied on binary data only, an extension to non-binary data is straightforward. To do so, literals x_i mean that the value of that variable is in relation to a constant in an instance, whereas a negated literal $\neg x_i$ means that the variable is not in relation with that constant: $x_l \Leftrightarrow x_l^i \bigodot c$, where $c \in \Re$ and $\bigodot \in \{<, \leqq, =, \geqq, >\}$. *must-Link* and *must-Link-Excl* can then be used as described above.

8.3 Experiments

This section first introduces the synthetic datasets that were created for the experiments. Second, an evaluation measure is presented and last, the experimental results are shown. For each constraint, the dependency on four parameters is evaluated: the number of attributes an instance consists of (*numAttr*), the number of attributes that are used in the constraint ($numFixedAttr$), the number of constraints in the data set ($numConstraints$) and finally, the number of instances in such a data set ($numInstances$). As this part of the thesis is focused on binary instances, the Hamming distance was chosen for the clustering process. It captures the proportion of dissimilarity between two instances x and y (cf. Equation 8.5), where n_{01} (n_{10}) is



(a) Data *must-Link* (b) Data *must-Link-Excl*

Figure 8.1: Synthetic data set idea - 8.1a: The three circles represent unconstrained clusters, while the two dashed circles denote constrained clusters. The *ml*-constrained clusters have a much larger diameter than the unconstrained clusters. Cluster overlaps are allowed in the given setting. 8.1b: The constrained clusters (filled) may overlap with the unconstrained clusters so that a separation of the instances becomes harder. The boxes represent exemplary medoids, a dashed line indicates a *mlx* constraint – they are similar but not equal.

the number of attributes that are equal to 1 (0) in instance x and equal to 0 (1) in instance y.

$$d(x,y) = \frac{n_{01} + n_{10}}{numAttr} \tag{8.5}$$

8.3.1 Data Sets

To evaluate the proposed constraints, synthetic data sets were created. The basic idea for the *must-Link* constraint is that it can connect instances which are very far apart but nevertheless belong to the same cluster. Then, a *must-Link* constraint can help to find their connection by a small attribute description. To show the use of a *must-Link-Excl* constraint, overlapping clusters are an interesting case. Many standard approaches cannot separate them, but including a *must-Link-Excl* constraint allows specifying which instances must belong to another cluster. Figure 8.1 illustrates these ideas. Clusters with *must-Link* constraints (Fig. 8.1a) are larger (big diameter) but are held together by the constraint. In contrast, unconstrained clusters are more compact. The data sets for *must-Link-Excl* constraints consist of overlapping clusters (cf. Figure 8.1b) that cannot be separated with an unconstrained clustering. For each data set a predefined clustering is assumed that consists of k clusters, of which *numConstraints* clusters are

Parameter	Abbrv	Default	Min	Max
numAttr	A	100	20	1000
numFixedAttr	F	4	1	60
numConstraints	C	4	1	20
numInstances	I	1000	40	1000

Table 8.1: Overview of the chosen parameter setting. For each parameter a default value is given and their ranges, respectively.

constrained. For each unconstrained cluster a medoid of length *numAttr* is created that is at least *interMedoidDist* apart from the other medoids. The parameter *prob* gives the probability of ones ($x_i^j = 1$) in the medoid. Second, *numConstraints* constrained medoids are created. The parameter *numFixedAttr* defines how many literals are included in the constraint. Next, instances for the unconstrained clusters are generated, by varying the medoid of the corresponding cluster so that they are no more than *smallDist* apart from their medoid. For constrained clusters, they must not be more than *bigDist* apart and moreover, must be in the scope of the constraint that is related to that cluster. Altogether, *numInstances* instances are induced, where a cluster contains on average *numInstances* (resp. k) instances. For the *must-Link-Excl* constrained clusters, the medoids are not created independently like it is the case for *must-Link* data sets. Instead, the constrained cluster medoids are derived from the unconstrained cluster medoids such that the resulting medoids are similar and the clusters overlap. To examine the quality of the clustering depending on the different ranges of the parameters, data sets with different parameter settings were generated. Table 8.3.1 summarizes these parameters' settings with their default values and the corresponding ranges. Throughout the value of the parameters *prob* (0.3), *smallDist* (0.1), *bigDist* (0.5), *interMedoidDist* (0.02) and k (20) were left fixed. Altogether, for each constraint (*ml*, *mlx*) and parameter value, 20 datasets were created, to take into account the variance among data sets.

8.3.2 Evaluation of Constrained Clustering

To evaluate the resulting clustering, the Adjusted Rand Index [46] (*ARI*) is measured. It is an adjusted version of the Rand Index [50] (*RI*) as described in section 3.2. A second quality parameter is the change in runtime and iterations, respectively, when the constraints are incorporated into the clustering process. Due to the ordering dependency and the random initialization procedure of the k-Medoids algorithm, each experiment was repeated 10 times to eliminate incidental effects. Altogether, this gives 200 test results for each parameter value and constraint.

8.3.3 Constraint Specification Costs

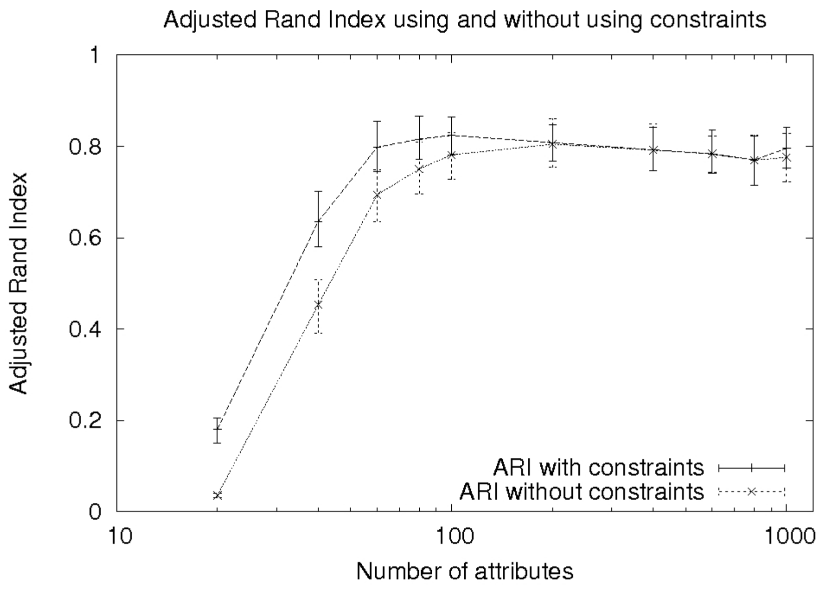
To illustrate the specification savings of attribute-level constraint compared to the standard instance-level constraints, assume the standard data set setting with four constraints and 1000 instances. On average, each cluster contains 50 instances so that for a clustering with instance-level *must-link* constraints at least 200 constraints had to be provided (4 constrained clusters * 50 instances) to make sure that all instances of the constrained clusters are grouped appropriately. Comparing this to the number of attribute-level constraints (4), the constraint specification compression is evident. Moreover, during the clustering process with instance-level constraints, each instance must be compared against all instances of each cluster, whether there applies an instance-level constraint. In the worst case, this results in $O(n^2)$ checks (let n be the number of instances). Comparing this to attribute constrained clustering, only $O(|C| * n)$ scope checks have to be performed (let $|C|$ be the number of clusters), as only the related attribute-level constraint is tested against each instance. Especially for large datasets and few clusters, using attribute constraints can thus result in large savings of runtime.

8.3.4 Results *must-Link* and *must-Link-Excl*

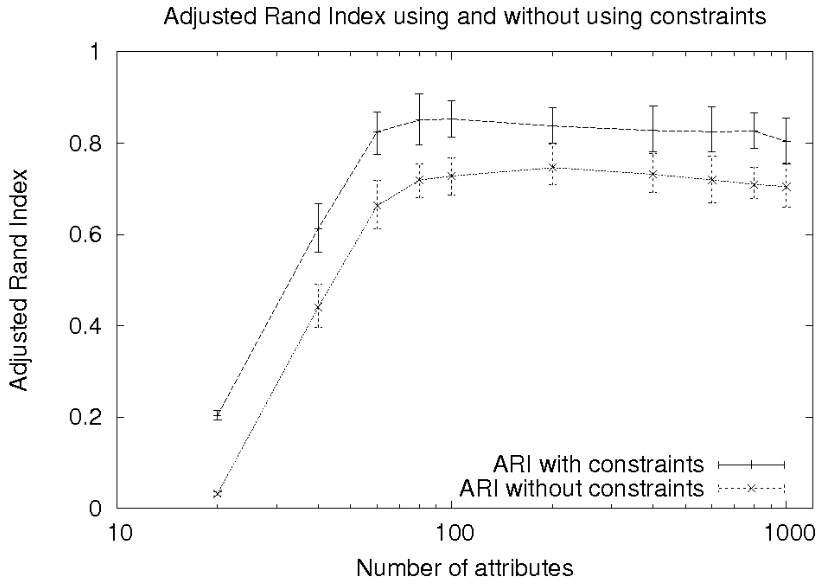
This section presents the resulting *ARI* and runtimes depending on the four varied parameters. For each evaluation, two figures are given. The first compares the *ARI*, when *must-Link* constraints are used for the clustering or not respectively (left figure), while the other one shows the improvement of the *ARI* for *must-Link-Excl* constraints. For every parameter value an error bar (Q25-quantile, mean, Q75-quantile) is shown. Below each figure the parameter settings for I, A, C, F and k are given. To judge the effect of the constrained clustering on the runtime, a subsequent table gives the runtime and the number of iterations (rounds) that were needed for the clustering with or without using the constraints.

Results *numAttr*

Figure 8.2 shows the *ARI* depending on the instance size. The x-axis gives the number of attributes an instance consists of, which is varied between 20 and 1000. The clustering process is more difficult for small instances, no matter whether constraints are used or not. For higher dimensions, the clustering results become better but do not exceed an *ARI* of 0.83 (when using *ml*-constraints) and 0.80 (without *ml*-constraints), respectively. Using *mlx* constraints leads to better results than without constraints, for all instance dimensions. Considering *ml* constraints, only in small dimensions information can be gained from constraints. Comparing the *ml* and *mlx* constraints, a higher information gain can be observed for the *mlx* constraints (the difference of the *ARI*s is higher). Table 8.3.4 shows the runtime and the number



(a) Comparison *must-Link*



(b) Comparison *must-Link-Excl*

Figure 8.2: Resulting *ARI* for different instance sizes (number of attributes) (I = 1000, A = -, C = 4 , F = 4, k = 20)

of rounds that were needed for the clustering. The larger the instances, the longer needs the clustering. This is mainly due to the longer instance comparison time, which of course increases for higher dimension. In fact, the clustering becomes easier for higher dimension for *ml* constraints, be-

A	20	40	60	80	100	200	400	600	800	1000
t *ml*	0.44	0.91	0.98	1.1	1.19	1.97	4.38	6.33	8.43	10.10
t	0.38	0.84	0.97	1.09	1.21	2.1	4.44	6.69	9.03	11.43
R *ml*	1.92	4.29	4.09	4.13	3.92	3.65	3.82	3.63	3.55	3.48
R	1.22	3.97	4.32	4.26	4.24	4.02	3.88	3.81	3.75	3.71
t *mlx*	0.67	1.07	1.19	1.27	1.32	1.89	3.3	4.7	6.31	7.71
t	0.65	1.08	1.19	1.34	1.41	1.81	2.94	4.12	5.19	6.8
R *mlx*	2	4.04	4.22	4.09	3.91	3.81	3.54	3.67	3.67	3.54
R	1.3	3.88	4.08	4.11	3.99	3.75	3.57	3.62	3.36	3.55

Table 8.2: Resulting runtime (s) and number of iterations (R) for different *numAttr*

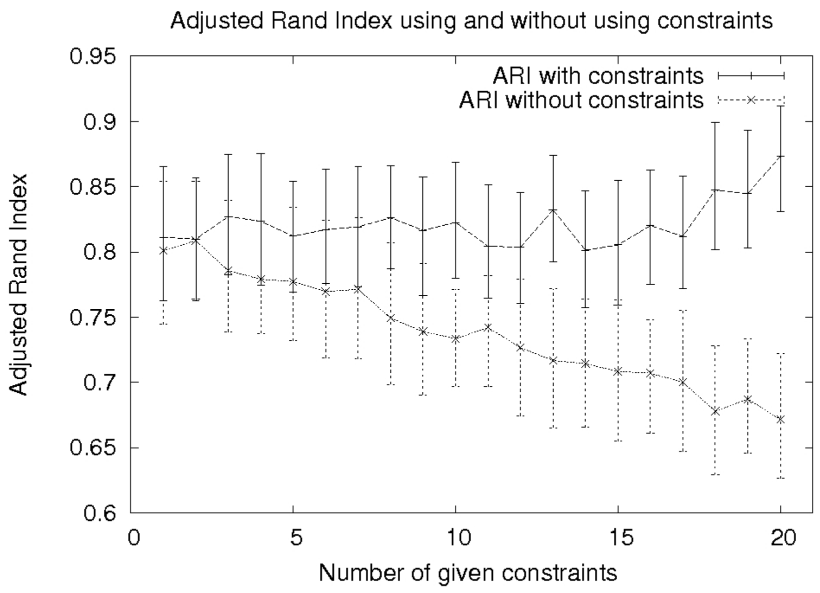
cause instances can be distinguished better. Thus, the larger the instances' dimensions, the more a clustering can benefit from *ml* constraints in terms of runtime. In contrast, *mlx* constraints lead to a slightly higher runtime. An explanation for this is that the initial assignment for an instance may be revised because it violates a constraint. Thus, additional computations have to be conducted. The number of rounds needed is decreased in higher dimension for both constraint types.

Results *numConstraints*

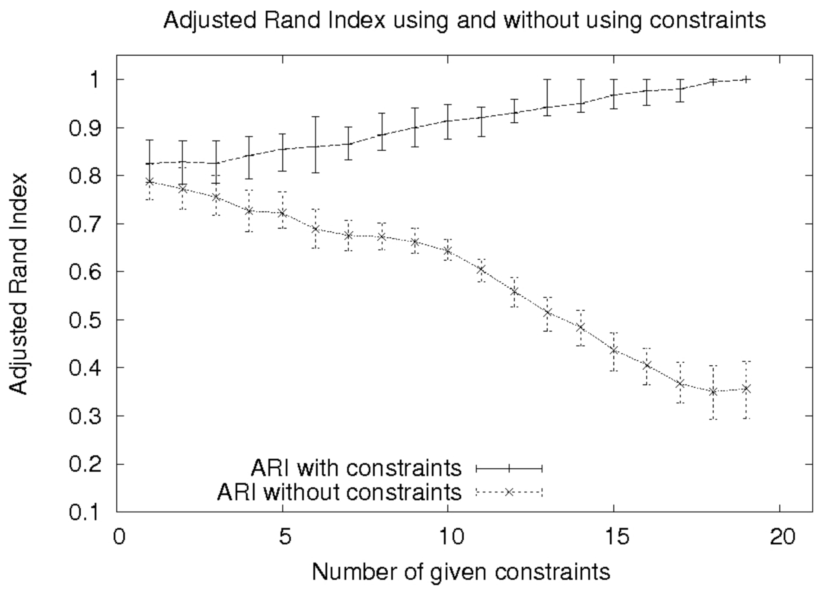
Figure 8.3 shows how the number of constraints that are used in the clustering influence the resulting *ARI*. If only few constraints are given, k-Medoids still gives a good solution. This is a result of the data creation process. Unconstrained clusters are quite dense so that the grouping of the instances is straightforward. The more constraints are included, the more separated the instances become. Then, the cluster assignment without constraints is more and more difficult. In contrast, for a data set that contains very scattered instances ($numFixedAttr \rightarrow 20$), the constraints are of much greater value to the clustering. The highest *ARI*-gain is observed for a large number of constraints. Concerning runtime and number of rounds, Table 8.3.4 shows that the runtime is not heavily affected by the inclusion of more constraints. In contrast, fewer rounds are needed to finish the clustering. The time that is saved by fewer rounds is consumed to process the constraints. Using the constraints is beneficial for both types of constraints, and the more constraints are used, the more savings can be achieved.

Results *numFixedAttr*

Figure 8.4 shows how the number of attribute that are included in a constraint influences the *ARI*. During the data creation process, this number



(a) Comparison *must-Link*



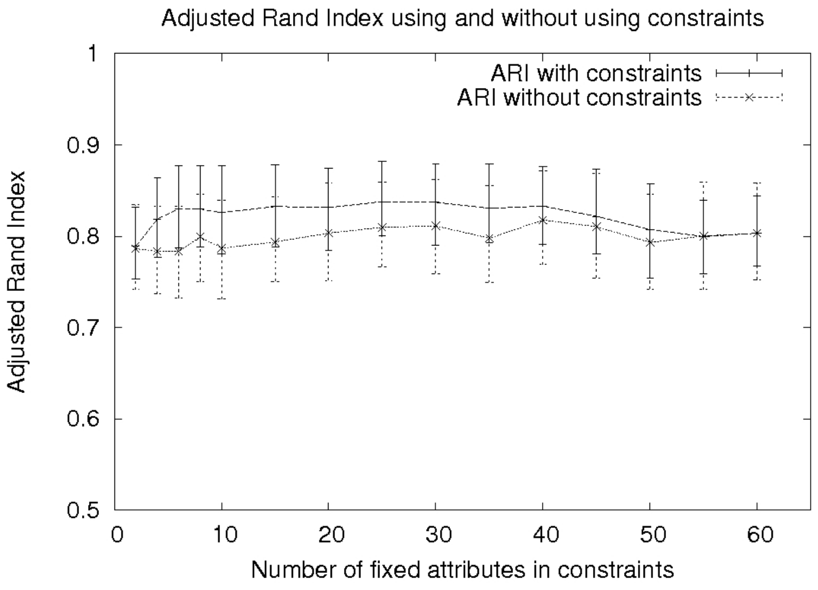
(b) Comparison *must-Link-Excl*

Figure 8.3: Resulting *ARI* for clusterings with a different number of constraints (I = 1000, A = 100, C = - , F = 4, k = 20)

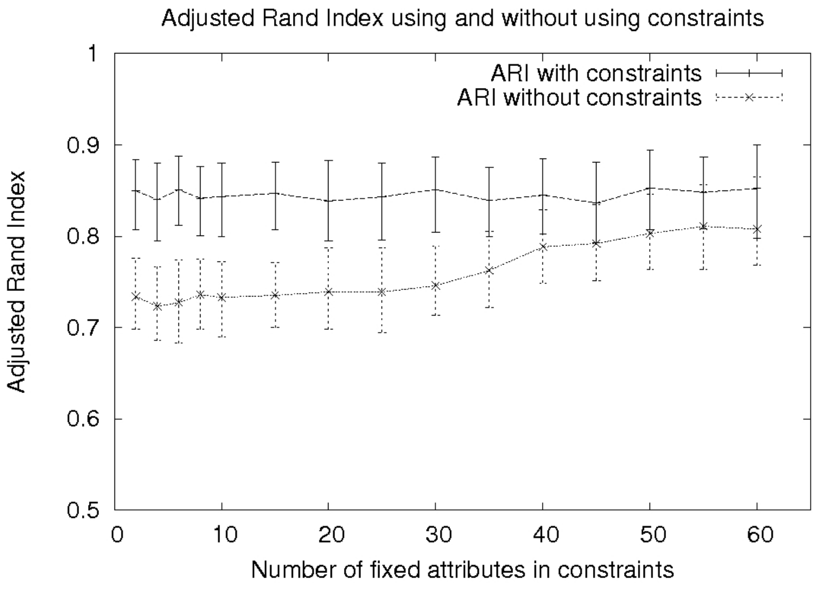
C	1	2	4	5	7	8	10	11	13	14	16	17	19	20
t *ml*	1.16	1.31	1.21	1.21	1.18	1.19	1.2	1.17	1.12	1.13	0.99	1	0.89	0.84
t	1.23	1.28	1.24	1.27	1.23	1.28	1.29	1.23	1.24	1.2	1.23	1.23	1.18	1.15
r *ml*	4.11	3.94	3.9	3.83	3.79	3.67	3.52	3.54	3.42	3.29	3.06	3.02	2.81	2.73
r	4.22	4.33	4.24	4.42	4.22	4.37	4.29	4.11	4.1	4.09	4.14	4.16	4	3.86
t *mlx*	1.38	1.36	1.41	1.37	1.39	1.42	1.32	1.21	1.18	1.18	1.18	1.14	1.09	0.99
t	1.41	1.41	1.48	1.44	1.45	1.45	1.46	1.46	1.56	1.59	1.61	1.61	1.61	1.43
r *mlx*	4.1	4	3.95	3.92	3.76	3.63	3.54	3.46	3.28	3.12	2.89	2.64	2.44	2
r	4.08	4	4.03	3.92	3.77	3.76	3.67	3.6	3.44	3.45	3.39	3.14	3.15	3.06

Table 8.3: Resulting runtime (s) and number of iterations for different *numConstraints*

specified the similarity of the instances within a cluster, because if an attribute is used for a constraint, its value will always be equal for all instances in the cluster. Overall, the constraints provide some additional information for the clustering, which is evident especially for *mlx* constraints. The best improvement can be observed when only few attributes are included in the constraints. Then, the higher variability in the instances is a problem for the unconstrained *k*-Medoids clustering. That is why the most improvement over a standard approach is to be found for small numbers of *numFixedAttr*. The *ARI*-gain decreases for higher values of *numFixedAttr*, because then the instances become more and more similar so that *k*-Medoids is also able to induce the correct clustering. How the runtime and number of rounds is affected by the inclusion of more attributes in the constraints is presented in Table 8.3.4. For *must-Link* constraints, the runtime increases with higher values of *numFixedAttr*, no matter whether the constraints are considered or not. Here, the unconstrained clustering is even comparable in the runtime. Although the runtime does not increase much for *must-Link-Excl* constraints, the runtime is also comparable to the case when no constraints are considered. The number of rounds is the same for all values of *numFixedAttr*, when using constraints, but slightly decreased for the standard approach. This can be explained by the fact that the separation



(a) Comparison *must-Link*



(b) Comparison *must-Link-Excl*

Figure 8.4: Resulting *ARI* for a different number of attributes that are included in a constraint (I = 1000, A = 100, C = 4 , F = -, k = 20)

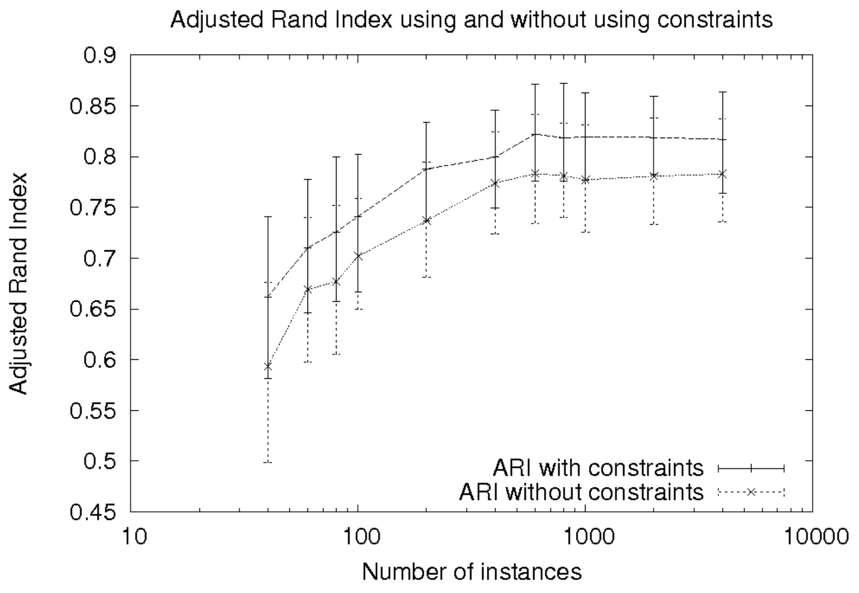
F	2	4	6	8	10	15	20	25	30	35	40	45	50	55	60
t *ml*	1.17	1.22	1.21	1.23	1.23	1.21	1.28	1.23	1.24	1.26	1.25	1.29	1.37	1.37	1.39
t	1.19	1.23	1.24	1.22	1.23	1.25	1.24	1.24	1.25	1.27	1.26	1.3	1.3	1.32	1.34
r *ml*	3.85	3.86	3.81	3.94	3.98	3.81	3.96	3.92	3.9	3.92	3.85	3.78	3.93	3.89	4.03
r	4.28	4.23	4.26	4.17	4.22	4.21	4.12	4.09	4.1	4.1	4.05	4.05	4	4.09	4.1
t *mlx*	1.36	1.32	1.34	1.38	1.37	1.34	1.34	1.34	1.36	1.39	1.36	1.4	1.42	1.36	1.4
t	1.42	1.37	1.4	1.37	1.38	1.4	1.39	1.43	1.42	1.48	1.45	1.43	1.43	1.43	1.43
r *mlx*	3.85	3.99	4	3.91	3.97	3.94	3.87	3.88	3.9	3.89	3.88	3.84	4.03	3.9	3.92
r	3.97	3.94	3.96	3.91	3.89	3.96	3.92	4.01	3.96	4.07	4.16	4	4.01	4.12	4.08

Table 8.4: Resulting runtime (s) and number of iterations for different *numFixedAttr*

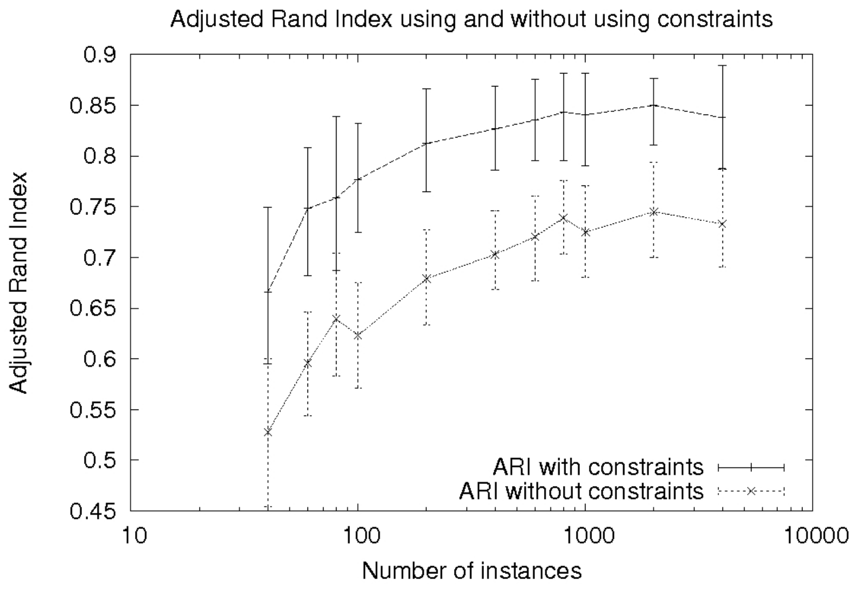
of the instances becomes easier, because more fixed attributes mean that the resulting instances are more similar. Thus, the standard clustering converges earlier. However, in most cases the constrained clustering needs fewer rounds than the standard clustering approach.

Results *numInstances*

Figure 8.5 shows the *ARI* for differently sized data sets. The data sets include from 40 to 4000 instances each. The inclusion of both types of constraints is beneficial. Throughout the parameter values, an average improvement of 4.2% (*ml*) and 12% (*mlx*) can be observed. The biggest improvement was achieved for small data sets (*ml*: 6.8% and *mlx*: 15% respectively) but no general trend can be inferred from these numbers. Table 8.3.4 shows that larger datasets increase the runtime and iterations. The inclusion of constraints is beneficial for the runtime and rounds. Both are lower when compared to the standard clustering approach.



(a) Comparison *must-Link*



(b) Comparison *must-Link-Excl*

Figure 8.5: Resulting *ARI* for a different number of instances (I = -, A = 100, C = 4 , F = 4, k = 20)

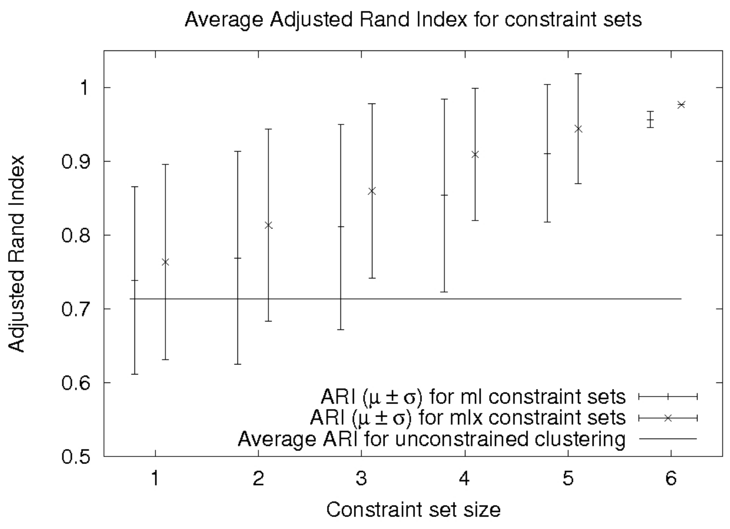
I	40	60	100	200	400	600	800	1000	2000	4000
t *ml*	0.1	0.15	0.2	0.3	0.58	0.69	0.89	1.2	3.48	13.72
t	0.11	0.14	0.2	0.34	0.64	0.75	1	1.29	3.69	14.74
r *ml*	2.71	3.12	3.44	3.52	3.75	3.84	3.85	3.94	4.01	4.02
r	2.81	3.32	3.56	3.9	4.05	4.22	4.37	4.3	4.3	4.26
t *mlx*	0.1	0.14	0.19	0.31	0.62	0.84	1.07	1.31	3.29	11.87
t	0.1	0.14	0.16	0.31	0.6	0.82	1.09	1.37	3.75	14.34
r *mlx*	2.71	3.16	3.35	3.62	3.69	3.77	3.98	3.88	3.91	3.86
r	2.84	3.17	3.41	3.67	3.67	3.87	3.94	3.85	4.08	3.88

Table 8.5: Resulting runtime (s) in second and number of iterations for different $numInstances$

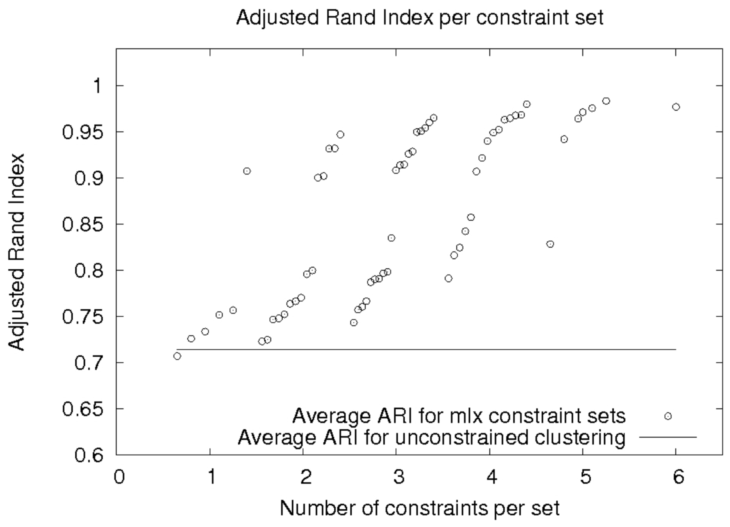
Results on the Real-World Data Set

A last experiment shows the applicability of the constraints on a real-world data set. The UCI zoo data set[2] was chosen for this experiment. It contains 101 binary instances that describe animals. Additionally, the corresponding cluster membership (the biological class) is included. Six constraints were created using biological background knowledge about the similarities of animals and the given attributes: $ml_1(milk)$, $ml_2(feathers)$, $ml_3(fins \wedge eggs)$, $ml_4(4Legs \wedge toothed \wedge eggs\ )$, $ml_5(6Legs \wedge breathes)$, $ml_6(\neg backbone \wedge \neg breathes)$. The same constraints were also created for the mlx type. Then, sets of constraints were created, including one to six constraints each, in order to show the individual contribution for each constraint as well as the benefit of their combinations. For each possible combination, a separate constraint set was constructed and then used for the clustering process. Again, the clustering was repeated 50 times for each constraint set. Figure 8.6a shows the mean ARI for each constraint set size as well as the baseline result when no constraints are considered. The leftmost point(s) (above 1) give the average ARI when the constraint set consists of only one constraint, while the point(s) above 2 show the average ARI for the combination of two constraints each. The results indicate that the more constraints are applied, the better is the clustering quality. However, not every constraint or constraint combination performs equally well. Figure 8.6b gives the mean ARI for each mlx constraint and their combinations. Although the general trend shows that the inclusion of such constraints is beneficial, there exist constraints (and combinations) that do not improve the result notably. Especially the inclusion of constraints mlx_3 and mlx_6 only leads to small improvements. Most significantly, the constraint mlx_6 alone even decreases the ARI compared to the baseline (leftmost point in

[2] $http://archive.ics.uci.edu/ml/datasets/Zoo$



(a) Mean *ARI* for constraint sets



(b) *ARI* for individual *mlx* constraint sets

Figure 8.6: *ARI* for the Zoo data set. (I = 101, A = 23, C = 1-6 , F = 1-3, k = 7)

Figure 8.6b). This shows that such constraints (as every other type of constraints) must be defined very carefully, but may significantly improve the clustering performance.

8.4 Conclusion

In this chapter we transferred the notions of the instance-level constraints *must-Link* and *cannot-Link* to the attribute level, where the *cannot-Link* effectively becomes a new type of constraint: *must-Link-Excl.* An adaptation of the well-known k-Medoids algorithm was presented that is able to incorporate the provided constraints. Each constraint was evaluated concerning several parameters. The results indicate that it is not only possible to define constraints on the attribute level, but also that they may be very beneficial in the clustering process. Moreover, we discussed on which types of clustering problems the constraints should be applied and how much the specification costs can be reduced compared to standard instance-level constraints. For future work, we would like to consider the combination of the presented constraints, how this combination will affect the runtime and how a probably NP-hard problem (like for other constraint clusterers [26]) can be avoided. Moreover, we would like to examine how constraints that express, e.g. the amount of shared attributes between clusters, can be formalized and incorporated into the constrained mining process.

Chapter 9

An Online Approach for PRTA Induction (*OPRTA*)

The recent creation of massive time labeled data sets (or event sequences [69]) in all areas of sciences and industry increases the need for scalable methods that induce models for any kind of underlying processes. Especially models that automatically induce the main elements of such processes (e.g., events) are very interesting for time-cause analyses. Important application areas include, amongst others, the analysis of disease progressions and disease patterns [67]. Probabilistic real time automata are a new graphical model to represent such processes. They automatically induce states and transitions based only on the data set. Moreover, they enable insights into the temporal relationships [97] and probabilities of change from one state to the next. The induction of PRTAs is currently based on the state merging method [78], which is considered state of the art in automata induction. However, this also includes the creation of a prefix tree acceptor (PTA) and a subsequent clustering step. For massive data sets both of these steps can become very time and memory consuming. Using an online approach is a straightforward solution for this problem and can be solved easily for the clustering step. However, how to achieve the PTA creation is still unknown. We tackle this problem by transforming both steps into online versions [82]. Each instance of the data set is first converted into a PRTA, which is then merged with a pre-existing one. The merge step is based on a clustering that uses maximum frequent patterns as representatives. We chose this clustering strategy to identify very homogeneous states, i.e. states that combine events having the same co-occurring characteristics. Moreover, this enables the incremental creation of PRTAs for data streams, which may arise, e.g. in sensor networks. The method is also able to deal with concept drift in such data streams, i.e. the PRTA is able to detect and to model a shift in the underlying process by the creation (or deletion) of states and transitions. Similar to existing work [61] this is done if a new event type is detected or when a state becomes out of date. We show that this method produces stable and informative automata that can be computed in reasonable time. In summary, our contribution is threefold:

1. We propose an online method for PRTA induction (*OPRTA*) that can be used for massive data sets.
2. We present adaptations of the method for data sets with concept drift.
3. We show three real world applications of the method along with the insights that can be gained from the resulting model.

This chapter first defines the problem setting (Section 9.1), the model and explains the online approach, which is extensively evaluated in Section 9.2.

9.1 Online Induction of PRTAs Based on MFP Clustering

This section first introduces the problem, which is followed by an informal description of the solution. This is then formalized in Section 9.1.3. Last, Section 9.1.4 explains how the proposed method can be used when concept drift is present.

9.1.1 Problem Setting

The task is to model a timed language model of data set D. Let D be a database of histories H_i: $D = \{h_1, \ldots, h_n\}$. A *history* h_i is a sequence of timed events $h_i = (\vec{e}_1, t_1)\ (\vec{e}_2, t_2) \ldots (\vec{e}_l, t_l)$. The event sequence is ordered corresponding to the time label (t_j) of the events. Note that the time labels need not necessarily form equal intervals, thus a varying amount of time can pass between successive events. An event e_i is a binary vector $\vec{e_i} = (a_{i1}, a_{i2}, \ldots, a_{ij})$ consisting of j attributes, where a_{ij} is equal to one if the attribute was observed in this event.[1] Data set D is to be modeled by a probabilistic real time automaton (PRTA) A PRTA is a directed graph and defined as follows.

Definition
A PRTA Γ is a tuple $\Gamma = (Q, \sum, T, S, F)$, where

- Q is a finite set of states
- Σ is a finite set of events
- T is a finite set of transitions
- $S = Q$ is the set of start states
- $F = Q$ is the set of final states

[1]This is similar to the notation of itemsets and thus an event can also be regarded as an itemset.

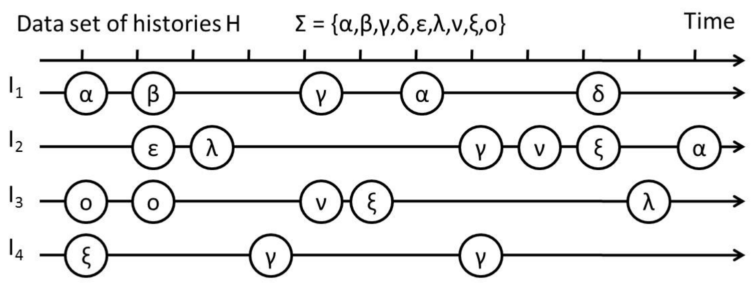


Figure 9.1: Example data set

A state $q_i \in Q$ is a pair $\langle E_i, \vec{f_i}\rangle$ where E_i is its set of events ($E_i = \{\vec{e_k} : \vec{e_k} \in C_i\}$) and $\vec{f_i}$ is an attribute vector called its profile. Σ are all events $\vec{e}$ that are observed in the input data. A transition $t \in T$ is a tuple $\langle q, q', T_L, \phi, p\rangle$ where $q, q' \in Q$ are the source and target states, $T_L = \Delta(E_i, E_j)$ and ϕ is a delay guard defined by an interval $[t_1, t_2]$ with $t_1, t_2 \in \mathbb{N}$. p defines a probability $p \in [0, 1]$ that this transition occurs. A PRTA that is also able to model concept drift (denoted by $PRTA_C$) is defined as tuple $\Gamma = (Q, \sum, T, S, F, Z)$, where Z gives for each state and transition the number of timesteps that have elapsed since their last observation: $Z = \{(o \subseteq Q \cup T, k \in \mathbb{N}^+)\}$.

9.1.2 Overview of the Approach

In this section, the general procedure of how to induce PRTAs online is given. The main idea of the online induction is that each new history is added incrementally to the existing PRTA. The histories $h_i \in H$ are observed one by one. First, each history h_i is transformed into a 'path'-PRTA, where there is only one predecessor and one successor for each node (exactly the events that occurred in the history). This PRTA is then collapsed into a PRTA Γ, i.e. all nodes $n_l \in Q$ sharing the same profile are merged into one state. Then, each state n_l of Γ is compared to the states Q' of a pre-existing PRTA Γ' that is built from all previous histories $h_l (l < i)$. If node n_l *belongs* to state $q_j \in Q'$, then node n_l is merged with state q_j. The function fNN that identifies such a state q_j is described in Section 9.1.3. Such a merge also induces that all incoming transitions from state q_j are incorporated into Γ'. If the sucessor node n_{l+1} of n_l of the PRTA Γ is also merged with a state q_{j+1} of the PRTA, then the transition t_n is linked between q_j and q_{j+1}. If there is already a transition t_s, then it is updated with the information of t_n. However, if there is no state q_j of Γ' that state n_l belongs to, n_l is only added to Q'. This also induces that there may be no transition from n_l to the remaining states of Q'. To clarify this procedure, consider the following example. Figure 9.1 shows a set of histories consisting of events $\{\alpha, \beta, \gamma, \delta, \epsilon, \lambda, \nu, \xi, o\}$. As the histories are observed one by one, history h_1 is considered first. Figure 9.2 illustrates this process. First, the

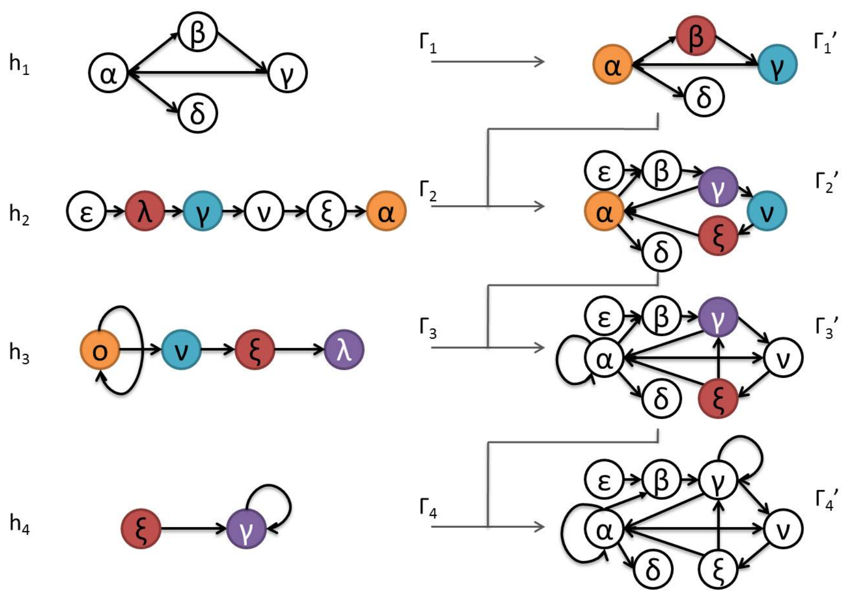


Figure 9.2: Example for the online induction of a PRTA. Left: Subsequent collapsed histories (Γ). Right: Resulting automaton (Γ′) when merged with new collapsed history (Γ)

history h_1 is transformed into a PRTA Γ_1 (the figure does not illustrate the transformation of the history in the path automaton), which is followed by the collapse step: all nodes of h_1 having the same profile $\vec{f}$ are merged, which results in Γ_1. The last step is to find similar nodes in the pre-existing PRTA. As there is no existing automaton for the first instance, Γ_1 is equal to Γ_1'. Then, the processing of h_1 is finished and history h_2 can be incorporated into the PRTA. Again, the compression is the first step, but cannot be conducted as there are no identical nodes in the history. Thus, Γ_2 remains a sequence of states. Then, each state of Γ_2 is compared to Γ_1' whether there is a state q_j it belongs to. This is true for the states that correspond to the events λ, γ and α: $\alpha \in \Gamma_2$ belongs to state $\alpha \in \Gamma_1'$, because they have the same profile, which is also true for $\gamma \in \Gamma_2$ and $\gamma \in \Gamma_1'$; $\lambda \in \Gamma_2$ belongs to state $\beta \in \Gamma_1'$ because they have a similar profile. (In Figure 9.2, the corresponding states of Γ_2 and Γ_1' are shaded equally.) The last step is then to merge the states of Γ_2 with the states of Γ_1', resulting in an updated PRTA Γ_2': all states in Γ_2 that do not have a corresponding state in Γ_1' are added to Γ_1', including all their transitions. The others are merged including their transitions. E.g., transition $t' = \langle \beta, \gamma, T_L', \phi', p' \rangle$ of the PRTA Γ_1' and transition $t = \langle \lambda, \gamma, T_L, \phi, p \rangle$ of the PRTA Γ_2 are merged to a final transition $t'' = \langle \beta, \gamma, T_L'', \phi'', p'' \rangle$ in the PRTA Γ_2'. The state β then contains two events β and λ, but is still labeled with β, because we assume that $\lambda \prec \beta$. Moreover, the delay guard ϕ and the probability p of the transitions are adjusted accordingly. Then the third and fourth history is processed in the same manner. The final automaton is given on the lower right side of the figure. It consists of seven states and 12 transitions. Note that the initial nine events are collapsed into seven states because β and λ as well as α and o are merged into one state due to their similarity. In the next section, we will explain formally how the transformation of the history in a PRTA is done and how states of Γ and Γ' are compared.

9.1.3 Creating the PRTA Online

The PRTA is created online by first converting the history into a PRTA Γ (compression step) and then merging the states of Γ with a PRTA Γ' that models all previous histories. Let dataset D comprise a set of histories H: $D = \{h_1, \ldots, h_n\}$ as described in Section 9.1.1. Each history h_i is converted into a PRTA Γ in two steps: first, for each event $\vec{e_j}$ a state $q_{ij} = \langle \{\vec{e_j}\}, \vec{e_j} \rangle$ is created $Q_\Gamma = \bigcup q_{ij}$. Then, each state q_{ij} is connected to its successor via a transition:

$$\begin{aligned}
&\forall(\vec{e_j}, \vec{e}_{j+1}) : isSuccessorIn(h_i, \vec{e_j}, \vec{e}_{j+1})\\
&\exists\ t_j \in T_\Gamma :\\
&\langle q_{ij}, q_{ij+1}, \Delta(E_{ij}, E_{ij+1}), [t_{j+1} - t_j, t_{j+1} - t_j], 1.0\rangle
\end{aligned} \tag{9.1}$$

, which creates the set of transitions in Γ: $T_\Gamma = \bigcup t_j$. In a next step, this PRTA Γ is collapsed, i.e. all states sharing the same event are merged into one state. For simplicity, let the new state q' be described by the set of states it consists of $q' = \{q_i, \ldots, q_j\}$. Then, each state of the compressed PRTA can be described as follows:

$$\begin{aligned}
\forall q' \in Q_\Gamma : &q_i, q_j \in q' \rightarrow q_i.E = q_j.E \wedge\\
&q_i \in q'_k, q_j \in q'_l, k \neq l \rightarrow q_i.E \neq q_j.E
\end{aligned} \tag{9.2}$$

Including the compressed PRTA Γ into the PRTA that models all previous histories $\Gamma' = (Q', \Sigma', T', S', F')$ gives the resulting PRTA Γ'. First, all observed events are added to the alphabet of Γ' and second, all states and transitions are merged if appropriate Γ': $\Gamma' = (merge(Q', Q), \Sigma' \cap \Sigma, merge(T', T), S', F')$. Function fNN identifies which two states q_i and q_j may be merged and is explained in more detail in Section 9.1.3. Function $merge(Q', Q)$ that defines how a merged state is created, is defined in Equation 9.3

$$\begin{aligned}
&merge(Q', Q) =\\
&\forall(q_i, q_j) : q_i \in Q', q_j \in Q, q_i.E = q_j.E : merge(q_i, q_j),
\end{aligned} \tag{9.3}$$

, where the method $merge(q_i, q_j)$ combines all profiles $\vec{f_i}, \vec{f_j}$ of the states q_i, q_j into one single profile $\vec{f_k}$ by their weighted mean:

$$\vec{f_k} = \frac{1}{\sum_{q_i \in Q \cap Q'} |E_i|} \sum_{q_i \in Q \cap Q'} |E_i| \times \vec{f_i}$$

The merge of the states also induces the merge of transitions, if appropriate. Last, the automaton can also be applied to data sets with concept drift. Here, elements of the automaton may become out of date. Therefore, the list Z is updated for each event in the history h_i: let $z \in Z$ be (k, q), where k is a timepoint and q is a set of states:

$$\begin{aligned} &\forall\ (\vec{e}_{ij}, t_{ij}) \in h_i : \\ &\exists\ z = (k, q) : k = t_i \wedge \nexists\ z' = (k', q') : \vec{e}_{ij} \in q', k' > k \\ &\rightarrow z = k, q \cup \vec{e}_{ij} \end{aligned} \tag{9.4}$$

Thus, the most recent timepoint is kept for each event and the corresponding states. Last, all states that exceed the minimum time constraint are deleted from the PRTA Γ''.

$$\begin{aligned} &\forall\ q \in Q'' : \\ &\exists\ z = (k, q) : q_i \in q \wedge k \geq minTime \\ &\rightarrow Q = Q \backslash q_i,\ T = T \backslash t : \\ &t = \langle q_i, q_j, T_L, \phi, p \rangle \vee t = \langle q_j, q_i, T_L, \phi, p \rangle \end{aligned} \tag{9.5}$$

Algorithm 17 summarizes this procedure: For each history, a PRTA is created and compressed. Then, each state of this PRTA may be merged with a state of the pre-existing PRTA. Last, if concept drift is taken into account, the timelist is updated and outdated states and transitions are deleted from the PRTA.

The time complexity of this approach is a sum of the collapse step and the clustering step. In the worst case $O(n^2)$ comparisons must be performed to check whether a state of history h_i can be merged with another state of history h_i. Next, for each state of Γ, the most similar state in Γ' is retrieved, which costs $O(n * m)$. Therefore, the total complexity of the algorithm is $O(n^2 + n * m)$.

Function fNN

Function fNN identifies whether there is a state q that is similar to a state q_{ij} in history h_i. This is the case if q_{ij} covers a large fraction of the frequent patterns of state q. The identification of frequent shared properties (corresponding to the minimum support threshold ms) is achieved by comparing q to the set of maximal frequent patterns (MFPs) of all nodes in q_{ij}. This set is updated incrementally via the AIST data structure (cf. Chapter 6)[76]. Function fNN computes the fraction of shared properties between q and q_{ij} of history h_i. If the maximal fraction exceeds a minimum overlap threshold, q_{ij} is merged with state q.

Algorithm 17 InducePRTA (D, $minsup \in [0,1]$, $minTime \in \mathbb{N}$, $ConceptDrift \in \{0,1\}$)

```
1:  $\Gamma' = (Q = \{\}, \Sigma\{\}, T = \{\}, S = \{\}, F = \{\}, Z = \{\})$
2:  for all $h_i \in \mathcal{H}$ do
3:    PRTA $\Gamma$ = createPRTA($h_i$)                    // cf. Equation 9.1
4:    $\Gamma$ = compressPRTA ($\Gamma$)                       // cf. Equation 9.2
5:    for all $q_{ij} \in h_i$ do
6:      State $q$ = fNN($q_{ij}$, $Q$, $minsup$) // cf. Sec. 9.1.3
7:      if $q \neq \{\}$ then
8:        $\Gamma'$ = merge($\Gamma'$,$q, q_{ij}$)               // cf. Equation 9.3
9:      end if
10:   end for
11:   if $ConceptDrift$ then
12:     $Z$ = updateTimeList($h_i$)                   // cf. Equation 9.4
13:     $\Gamma'$ = deleteOldNodes($Z$, $minTime$)           // cf. Eq. 9.5
14:   end if
15: end for
```

9.1.4 Adaptation For an Unbounded Data Set With Concept Drift

The presented algorithm can be adapted for unbounded data streams with concept drift. Concept drift means that the underlying true concept (here assumed as an automaton) changes as time goes by and thus, elements of the automaton become out of date. In the following section the characteristics of such a data stream setting are described. Moreover, the definition of when an element of an automaton is out of date will be introduced.

The Repeated World Data Stream Setting

The basic question is how a change of a concept can be observed, where one essential element is how time is monitored. Often, processes are recorded in successive time intervals. Then, a data bag is created with these measurements. In the current problem setting, such a bag is a set of histories which is additionally labeled with a time stamp. Such measurements are repeated in equal intervals, which explain the term *repeated world setting.* As an example, consider the monitoring of a multivariate process of a company within one week, where at the end of the week the set of histories is collected and transferred. Then, an updated model can be learned with this data bag. In the repeated world setting, concept drift is observed, when elements of the automaton do no longer occur in recent bags but only in older ones. Of course, there may also be new elements of an automaton. This may be the case, e.g. if a process changed within the company (maybe due to a change in the personnel). Then some states or transitions may dis-

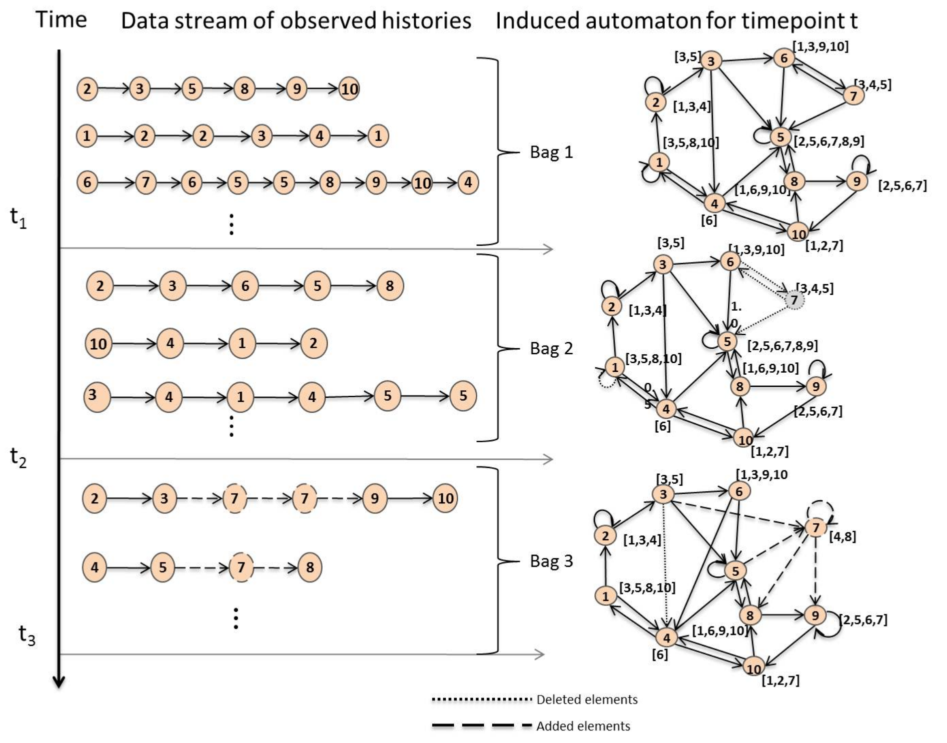

Time
Data stream of observed histories
Induced automaton for timepoint t
Bag 1
Bag 2
Bag 3
t_1
t_2
t_3
Deleted elements
Added elements

Figure 9.3: Illustration of the data stream (repeated world) setting. Left: data stream illustration (bags). Right: Resulting automata for the data stream. Added and deleted elements of the automaton are drawn with dashed lines.

appear, while others arise. Thus, the change of the automaton reflects how whole processes change over time. In the repeated world, the updates of the automaton are based on the new bags of histories (left part of Figure 9.3). The first important property is that each history in each bag may begin at timepoint t_0, i.e. the first event was observed at timepoint t_0. This also suggests that there exist two timelines. One is the ordering of the events (horizontal timeline) within one history, the second timeline shows in which order the bags arrive (vertical timeline). This also defines the time threshold o when an element is out of date. This is the case, when an event was last observed too long ago. As an example consider state 7. This state does occur in bag one but not in bag two. Thus, it is out of date when bag 3 is observed (o being set to 1) and is therefore deleted. This shows that the time to evaluate whether one state is out of date is dependent on the time of the bag and not the timestamp in the history.

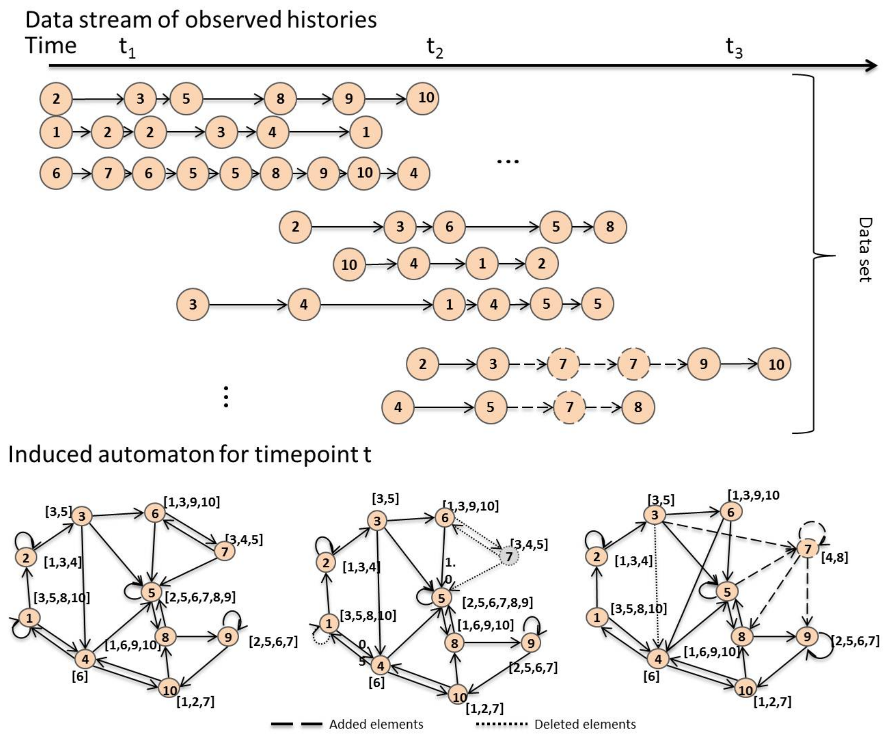

Data stream of observed histories
Time
t1
t2
t3
Data set
Induced automaton for timepoint t
Added elements
Deleted elements

Figure 9.4: Continuous world setting

The Continuous World Data Stream Setting

However, a change of the concept can be observed in another way, when one set of histories is monitored continuously. There, the set of histories is known up-front but it is unknown, which history will be updated with the next data set and how many new events a history will provide. Such a world is then called *continuous world*. As an example for such a world consider the ongoing recording of fixed population of patients, where each patient record (and so his history) is extended with each physicians visit. In such a world, a change of the concept is observed, if there are elements of the automaton that have not been observed for a long time within all histories. The last observation is the last absolute time point of any history for this element. Note that the set of updates of the histories may also vary in its length. It may contain only the observations for the next timepoint in line, but it also can contain updates over several timepoints. Figure 9.4 shows the continuous world setting. In the continuous world, time is measured absolutely, i.e. the time stamp for each event is equal to its last observation within the whole data set. As the data stream progresses, updates for histories are delivered (e.g., for H_3), or new histories are started (e.g., for H_7). For example, the last observation for state 7 of the automaton is t_1. As more and more data is recorded, state 7 is not observed anymore and thus is out of date at time point t_2. Therefore, it is removed from the automaton. The same is true for transition $q_1 \rightarrow q_1$ (not illustrated in the histories). As time progresses, a new state 7^* (indicated by dashed circles) is observed and also included in the data set. Note that these are updates and thus there is only one time line and the time stamp always increase.

Algorithmic Adjustments

To adapt the presented algorithm for concept drift, each element (states and transitions) of the automaton must be annotated with a label that shows when this element was observed the last time. If such an element then becomes out of date, it must be deleted from the automaton. How this is achieved in detail is presented in the following.

Repeated World Adjustments Beginning with the automaton that is (also incrementally) created with a first bag of histories, the algorithm receives updates of histories (bags) as input. Thus, each update comes along with a label that defines the current time stamp. In the repeated world, histories are loaded in bags and processed like in the standard incremental setting. Additionally, each element of the automaton is assigned a timestamp that defines when it was observed the last time: A PRTA that is also able to model concept drift is defined as tuple $\Gamma = (Q, \sum, T, S, F, Z)$ (cf. Section 9.1.1), where Z is a mapping that gives for each state and transition the number of timesteps that have elapsed since their last observation: $Z = \{(o \in Q \cup T, k \in \mathbb{N}^+)\}$. The timepoint is the timestamp of the bag and not the timestamp of the history because the change of the whole process shall be explored. Each history update is integrated as described above into the existing automaton. Additionally, the last observation timepoint of the elements (state or transition) which are covered during the integration process is updated to the current bag timepoint. This is achieved by a simple hash structure. After all histories of the bag are processed, outdated elements are deleted from the automaton. This is the case if the element's time stamp is too old compared to the current bag's time stamp. Note that the threshold o, when an element is out of date, is specified by the user.

Continuous World Adjustments Beginning with the automaton that is created (also incrementally) with a first set of histories, the algorithm receives updates of specific histories as input: For each history the set of newly observed events is given as input. This includes that the last event of the history is the first event of the history-update so that the starting point for each history can be deduced. Each history-update is integrated as described before into the existing automaton. Additionally, each element (state or transition) that is covered during the integration process is updated to the most recent element observation time point. After each history update was processed, outdated elements are again deleted from the automaton. However, if the data source cannot store the last event of each history, the implementation of this setting may be expensive. Then, this assignment must be done within the automaton's structure. Therefore, we do not concentrate on this setting in the experimental section but only on the repeated world setting.

9.2 Experiments

This section presents stability results, the performance on real world data sets and finally shows how the algorithms works on a data set with concept drift.

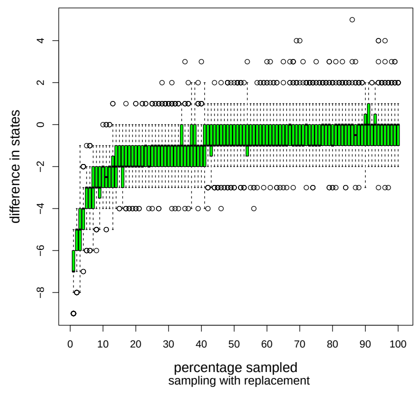
9.2.1 Performance on Stream Setting Without Concept Drift

This section addresses the algorithm's stability, which will be tested on the synthetic data set and then shown on real world data sets. For the stability experiments, first, a bootstrap analysis presents how often the true underlying structure of the automaton is rediscovered. Second, we show how the size of the histories influences these results. Last, we present how the runtime of the algorithm and the final number of states depends on the three standard parameters input data size, history length and minimum support. To show the applicability of such a model, we applied the algorithm on several real world data sets. Note that we will not address the quality of the clustering because this was already evaluated in Chapter 5.

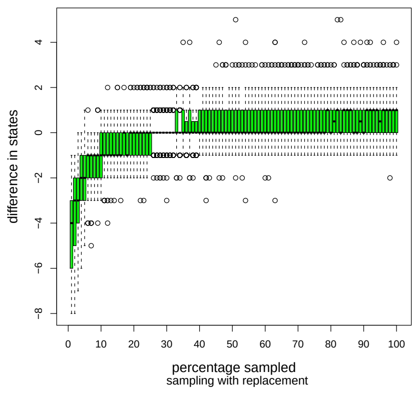
Stability Analysis

This section shows the results of the experiment in a bootstrap evaluation. The task is to extract the correct structure of a predefined automaton (cf. Section 3.1.1). To compare the final automaton to the predefined one, the measures Δ_{States}, L_{States} and the F-Measure, introduced in Section 3.2, are taken into account.

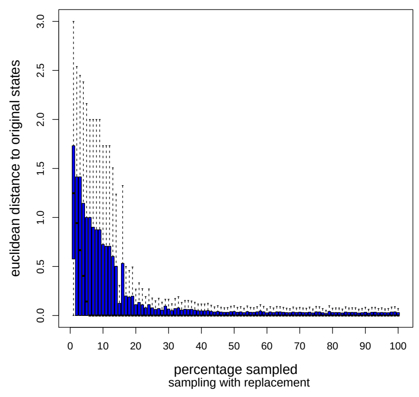
Rediscovery of a Known Automaton The results of the bootstrap analysis for the synthetic data set indicate that the proposed solution is able to find the correct number of states and transitions although it tends to create to many states (cf. Figure 9.5b). Additionally, the distance from the induced states to their closest original is also very small (cf. Figure 9.5d). Comparing these results to the PRTA induction with a previous PTA-creation 5 (Figures 9.5a to 9.5e), a minimal performance decrease can be observed, which is owed to the fact that not all events are observable right



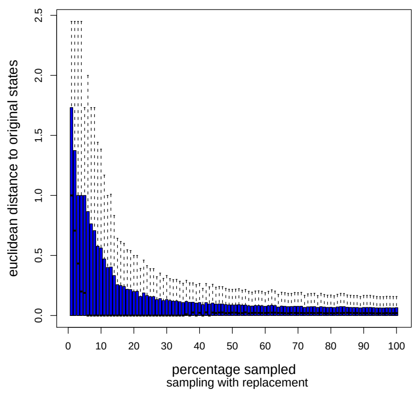
(a) Δ_{States}



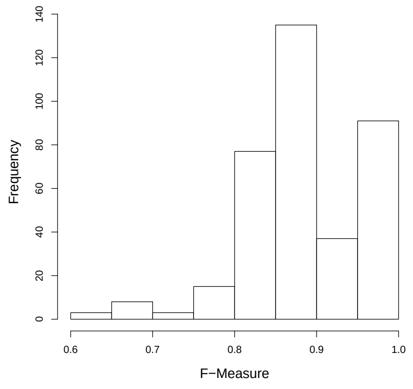
(b) Δ_{States}



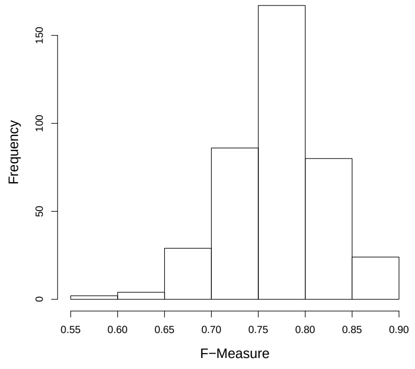
(c) L_{States}



(d) L_{States}



(e) F-Measure of learned transitions



(f) F-Measure of learned transitions

Figure 9.5: Accuracy of the induced PRTA compared to an approach using a PTA [76]. Figures 9.5a - 9.5e show the results when including a previous PTA construction, Figures 5.7b - 5.7f show the results for the proposed approach.

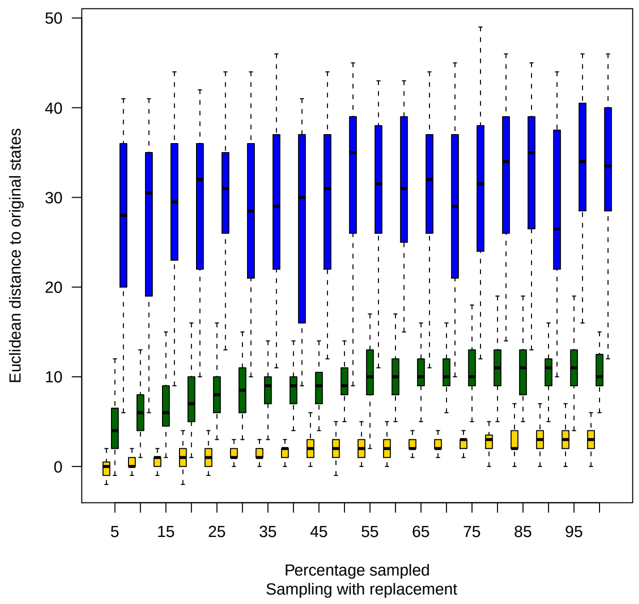
History length ($ds = 10000$)			Data set size ($hl = 100$)		
hl	runtime	#states	ds	runtime	#states
10	3.9	21.5	100	0.96	12.5
50	10.8	23.6	1000	2.96	18.9
100	21.1	25.6	5000	10.17	22.7
500	137.8	28.0	10000	20.96	25.6
1000	419.5	30.3	50000	191.76	26.5
			100000	725.27	29.9

Table 9.1: Runtime (sec) and number of states for different history lengths and data set sizes

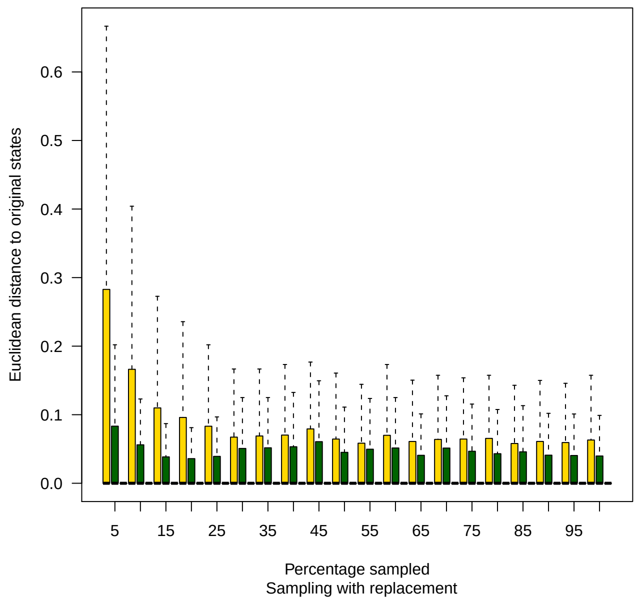
at the beginning and thus, the online approach is introducing errors at the beginning. Overall, the accuracy loss compared to an approach that uses a PTA creation are almost negligible. In contrast, the online approach is able to include more information than the batch approach and may thus produce much more specific profiles and more accurate transition probabilities. A further investigation concerning the RR of a state shows that all states are learned well except state 7 (not illustrated). This may be due to the small transition probability from state 6 to state 7 and its high similarity to state 2, so that the events of this state are only rarely observed, or often misplaced in a wrong cluster.

Influence of the History Length We repeated the bootstrap analysis to evaluate the influence of the history length (hl) on the number of final states and their profiles' accuracy. Therefore, data sets having different history lengths (100, 1000, 10000) have been used. Figure 9.6 illustrates that the longer the histories, the better the induced profiles match those of the original automaton. However, this comes along with many more states (cf. Figure 9.6a). This can be explained by more exceptions that are present in the data for longer histories.

Number of States and Runtime Depending on History Length, Minimum Support and Data Set Size The next evaluation addresses the algorithm's dependency on its main parameters: data set size ds, minimum support ms and history length (hl). We calculate the runtime and final number of states for 10 different data sets of the synthetic automaton for each value of hl, ds and ms. The standard parameter values are $ms = 0.5$, $hl = 100$ an $ds = 10000$. Table 9.1 gives the final average number of states and runtime (in sec) for different values of hl and ds, while Figure 9.7 illustrates the behavior for various minimum supports over all data sets. Table 9.1 shows that the actual runtime is linearly dependent on hl and also nearly linear with increasing ds. For both parameters, the number of final states



(a) Δ_{States} for all history lengths



(b) L_{States} for all history lengths

Figure 9.6: Stability of the online algorithm for the proposed approach ((yellow, green, blue): hl = (100, 1000, 10000)) Difference of states (Top) and Euclidean distance (Bottom) for the induced automaton to the true automaton structure.

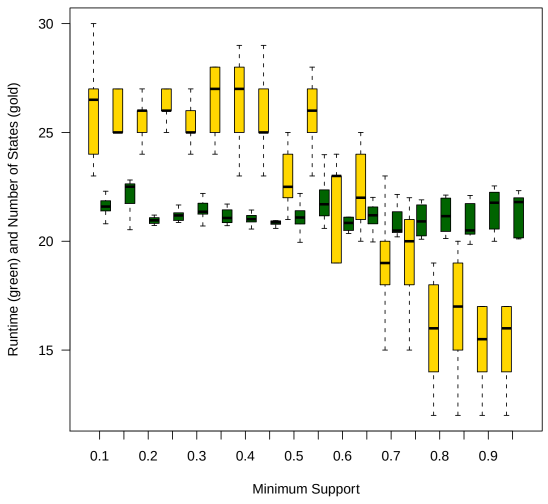


Figure 9.7: Number of final states (yellow) in the automaton and the corresponding average runtime in sec (green) for a varying minimum support.

increases, which is to be expected as the more data is present, the more exceptions (and thus new patterns) occur. For such exceptions new states are created. However, the final number of states is quite small compared to the number of events in the data set (10,000,000), which shows that the algorithm can compress the patterns in the data. Figure 9.7 illustrates that the runtime is quite stable with different numbers of ms, while the number of final states decreases. This can be explained by the fact that if the minimum support increases, more and more instances in a cluster must cover the representative pattern. This can only be achieved if the representative pattern is small. Then, more instances cover this pattern and are added to the cluster. Therefore, fewer clusters are found.

Runtime Comparison of the Approach To illustrate the advantages of the genuine online approach, we compared its runtime (cf. Table 9.1) to the approach that first creates a PTA and subsequently runs the proposed online clustering (cf. Chapter 5)[76]. Such an approach needs 2,6 s on average ($hl = 100$) for $ds = 100$, 333 s for $ds = 1000$ and already 9 hours for $ds = 5000$. This extreme difference is mainly due to the merging procedure, which takes very long. However, even the creation of the histories, the PTA and the subsequent clustering lasts approximately 17 s, which is still above the runtime of the online approach.

Hepatitis data set								
ds	50	100	200	400	600	900	1K	1236
s	6	11	18	37	54	75	81	90

Disease group data set							
ds	100	1K	5K	10K	50K	100K	147656
s	1	1	2	2	8	24	51

Table 9.2: Runtime (sec) on the Hepatitis data set for different data set sizes (*ds*)

Performance on the Real World Data Sets

This section presents results of the proposed method for real world medical and biological data sets. Therefore, the Hepatitis, disease group and yeast data set have been used and their results are now discussed consecutively. First, the runtime of the approach is tested on the Hepatitis data set. Table 9.2 gives the results and shows that even for such a 'hard' (because it has a lot of attributes and long histories) data set, the runtime is reasonably low. Note that although the number of instances is only 1236, this Hepatitis data set includes 52,520 single events, due to the long histories. The main time consuming step here is the clustering and not the collapse method. The resulting automaton can be used to foresee how the blood values will develop based on the current blood values.
The same evaluation was run for the first disease group data set (P10), for which the runtime is also shown in Table 9.2 (lower part). Here, the algorithm is faster as there are less attributes and shorter histories given. Again, the runtime is reasonable for the given application. Interestingly, the algorithm identifies a stable number of states (52) for data sets with more than 5000 histories. When applying our approach on the second disease group data set (P01), the resulting automaton identifies 181 states within a total runtime of about 2 minutes. This shows that even with a larger number of attributes, the approach is still fast. Moreover, the number of states does also not increase for data sets larger than 5000 histories. The stable number of states for both data sets suggests that all important patterns in the data have been found. Although the final number of transitions increases for larger data sets, the ratio of new transitions decreases. We thus can expect that even the number of transitions will be fixed at a certain data set size. PRTA learning based on DIANA clustering [76] is not included in this comparison as even for the smallest data set it runs out of memory and needs longer than the online approach on the largest data set.
Regarding the structural properties of the resulting automaton for data set $P10$, we observe that most states have short representatives (MFPs): mostly, they consist of one to three DGs. The full profile then comprises the repre-

sentative DGs and may additionally have exceptional DGs. 20% of the states cover more than 50 events, while about the same amount captures less than 5 events. There are no states with no in/outgoing transitions except two states, which shows that different disease phases of different persons have been well combined in the automaton. Thus, the automaton is a generalization of the individual patient histories. There are also several hubs, which combine events that share very frequent DGs. Moreover, plenty of exceptional DGs are found in the profiles of the hubs. Such a state reflects very frequent disease phases along with all possible comorbidities. However, the additional diseases may be regarded as random side effects as their do not occur as often as the main disease. Overall, many states focus on the DGs hypertension, diabetes or heart problems, i.e. these DGs have a frequency of more than 90% in the resulting profile. This makes sense in a way as these are also the most common diseases, which are then represented along with their accompanying diseases. The transitions in this automaton reflect the probability of a change in the disease status, i.e. whether new diseases arise or others are cured or not coded anymore. For the most hubs there is no one main transition, but several having a moderate probability. Non-hub states, in contrast, are associated with such a main transition but this can still be regarded as a random effect, because there are not so many patient histories included. Overall, the automaton shows which disease patterns exist in this specific population and how probable transitions from one to another disease status can be.
The last experiment addresses the biological knowledge that can be inferred from such an automaton. Figure 9.8 is the final automaton for the yeast data set. The states' profiles show which gene is active at which timepoint and therefore, the metabolism of the cell. The overall structure of the automaton shows that the cell in this experiment undergoes a cycle in the gene expression. This is known to be the cell cycle. Besides, a known resting phase is modeled by the delay guard ($\varphi = [1, 2]$) on transition 1 to 2 that shows that the cell can either step forward to the very next cell stage or wait. All these experiments show that a PRTA can correctly identify the stages of life in a population or of individuals, annotate them with important properties and can thus reflect the dynamics in such systems.

9.2.2 Performance on Stream Setting With Concept Drift

Finally, the approach is tested in a setting with concept drift. Therefore, we created a data set that is based on three different underlying automata that are derived from the one shown in Figure 3.1. This illustrates the repeated world (cf. Section 9.1.4) setting, where after a certain time span the underlying process changes, i.e. different automata should be found. The first automaton is the standard automaton. For the second one (the second concept), state 7 was deleted as well as transitions $1 \rightarrow 1$ and $3 \rightarrow 4$. The

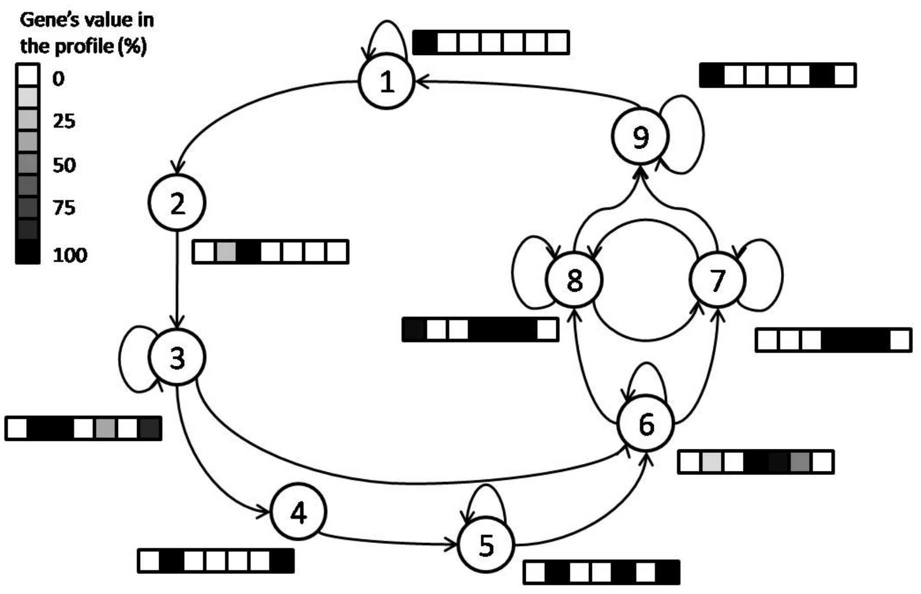


Figure 9.8: Result of the approach on the yeast data set. The black and white squares represent the values of the genes in the state's profile. The darker, the higher its values.

ds	100	500	1000	5000	10000	50000	100000
s	1	2.3	3.4	11	23	202	745

Table 9.3: Mean runtime (in seconds) for the data set including concept drift for different data set sizes (*ds*)

second derived automaton (concept three) includes a state 7^* with profile [4,8] and additional transitions. Using these concepts, the first bag of the data set is created by using the original automaton, the second and the third bag with the first and second derived automaton (cf. Figure 9.3, right). Then, the algorithm was applied with $o = 1$ (cf. Section 9.1.4). The concept changes are well identified for small error ratios (not illustrated). As for larger error ratios the clustering is also harder, the quality of the inferred automata decreases. Table 9.3 shows how the search for concept drift affects the runtime. The runtime is only slightly longer than for the approach that does not account for concept drift. We thus can conclude that the approach may also be valuable for problems with concept drift.

9.3 Conclusion

In this chapter, we introduced a method to induce PRTAs online. Therefore, we leave out the PTA construction step and instead collapse the histories before merging them with a pre-existing PRTA. Still, this merging is based on an online clustering method. Besides, we discuss how this approach can handle data streams with concept drift. The experiments on synthetic and real world data sets showed that the method is stable, scalable and can capture patterns from the application domain. Most importantly, we discussed the application of this method for the induction of PRTAs on health care data. The resulting automaton reflects the progression of diseases within a whole population along with the probabilities of health status changes. For future work, we first want to improve the accuracy of transitions, for which up to now the false positive rate is still too high. Therefore, we consider a hypothesis test that defines whether a transition having a very small probability is more likely noise or an exception in the data. Finally, we want to apply the automaton on large gene expression data sets that cover expression ratios over a long time period in order to discover different cell stages on a more detailed level.

Chapter 10

Summary and Outlook

In the last chapter of this thesis, the main contributions are summarized and an outlook on possible future research is given.

10.1 Summary

This thesis is concerned with the induction of process models for multivariate time series, e.g. the progression of diseases within a population. There, a model is desired for the development of the health status (combination of diseases) of each individual. The main problems are that the model should be easy to understand, e.g. by a graphical representation, it should be able to model process cycles, give probabilities of generalized events, and incorporate time. Therefore, we have introduced a new type of model to mine patterns in processes. Until now, processes were mainly described by models like HMMs or Petri Nets, which address important aspects of process mining. However, some features of each model, e.g. the ability to model time (cf. Chapter 2), have not yet been included in the corresponding other model types.

1. Our first contribution is an approach to identify a probabilistic graphical model (a probabilistic real time automaton [PRTA]) that (a) is able to automatically identify multi-variate events in a given process log, (b) allows for the inclusion of time-dependent queries, (c) can represent cycles if they are to be found in the data and, (d) is able to identify (at least up to a certain level) events in the presence of noise.

2. Secondly, this approach was enhanced to be able to handle large data sets. This was done by first,

   - introducing a data structure (augmented itemset tree [AIST]) to efficiently mine maximal frequent patterns in data streams. The main feature of this approach is that (a) it focuses on mining online sparse datasets (having nevertheless many features) with (b) large frequent patterns and (c) for which the set of maximal frequent patterns shall be extracted after each new instance.

- Then, this data structure was applied within the PRTA-framework (scalable PRTA (SPRTA)), in particular to handle the special dataset characteristics, which are, sparseness, high dimensionality and magnitude. Using the AIST for the induction of PRTAs instead of the previously used DIANA-based approach, resulted in faster runtimes, better discriminable profiles and even in a better capability to identify predefined model structures.

3. Third, we investigated how to include user constraints into a PRTA. The challenge here was to find a way to express specific types of constraints and incorporate them into the PRTA framework. This had not been described nor used before.

   - Our first step was to provide a language to describe constraints at the attribute level for clustering, i.e. the user may describe the properties of the resulting clusters. In contrast to previous constraints, this type of condition may cause lower specification cost compared to using, e.g. instance-level constraints.
   - Having introduced and evaluated the new type of constraint, we presented a method to incorporate them in the SPRTA framework. We showed how they can be used next to the maximal frequent pattern based clustering procedure (constrained SPRTA (CSPRTA)). Experiments using such constraints show their applicability in the domain of biology and medicine.

4. Last, we extended the SPRTA-approach to handle data streams of process logs. The first problem was to eliminate the step of the PTA-creation, upon which the clustering depends. Second, we addressed concept drift, which occurs frequently in data streams. Both aspects have been included in the SPRTA approach which resulted in the online PRTA (OPRTA) method.

We hope PRTAs cannot only be applied in the presented applications but may also be adopted to fit other application problems. Certainly, there are ways to incorporate some of the PRTA's features into HMMs or Petri nets so that an even better understanding and modeling of processes can be achieved. Then, each of these models will be able to model one aspect of process mining best so that a combination of all of these models may reflect every aspect of the process.

10.2 Outlook

Although the initial PRTA-induction method was improved in several aspects, there is ample room for further research. In this work, we have addressed a specific problem setting of the medical domain with a specific data set. This data set is very large, has many attributes, but is nevertheless sparse. For such a data set the SPRTA is very well adapted. However, for denser datasets (one could think of a population which is very sick), the SPRTA framework is not appropriate because the space of frequent patterns cannot be examined in an efficient manner. For denser data sets, one may thus either adopt the AIST framework or incorporate another frequent pattern miner.
Besides, if the AIST data structure should be applied to other problems that need a much smaller minimum frequency threshold, it must also be improved as it is currently not able to handle thresholds smaller than 0.08 efficiently. Using closed sets may be a first idea to tackle this problem.
Regarding the inclusion of background knowledge, there may of course also exist constraints that still cannot be expressed in the given language. That may be the case if a constraint cannot be expressed by basic logical expressions but needs a higher level of logic (e.g. including functions). Consider, for example, that all instances with a specific number of attributes must be put in one cluster. Besides, another natural extension in this field would be to provide a constraint that disallows an instance being clustered in a group because of a combined condition regarding its characteristics.
In general, the (S|O)PRTA framework should be further tested on other domains, to validate its applicability. We think that there are still quite a few improvements possible to increase the PRTA's expressive power. Therefore, we already thought about learning relations on the transitions that may summarize or even generalize the events that are annotated on the transitions. Logical learning would certainly fit very well for this problem.
Additionally, we have already started to include a statistical test to separate noise from exceptions in the data. This hypothesis test was incorporated into the PRTA-framework by Verwer *et al.* [96]. First results are encouraging and show that it may be possible to detect exceptions and keep them in the profiles, rather than to discard them. Finally, we hope that this type of model complements the set of existing process models and finds many applications.

Chapter 11

Bibliography

[1] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data 1993*, pages 207–216. ACM, 1993.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[3] H. Ahonen, H. Mannila, and E. Nikunen. Forming grammars for structured documents: an application of grammatical inference. In *ICGI '94: Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 153–167. Springer-Verlag, 1994.

[4] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[5] D. Arlia and M. Coppola. Experiments in parallel clustering with DBSCAN. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 326–331. Springer-Verlag, 2001.

[6] L. E. Baum, T. Petrie, G. Soules, and N. Weiss. A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. *The Annals of Mathematical Statistics*, 41(1):164–171, 1970.

[7] J. Besson, C. Robardet, and J.-F. Boulicaut. Constraint-based mining of formal concepts in transactional data. In *Proceedings of the Advances in Knowledge Discovery and Data Mining, 8th Pacific-Asia Conference, PAKDD 2004*, pages 615–624. Springer-Verlag, 2004.

[8] S. Blachon, R. Pensa, J. Besson, C. Robardet, J.-F. Boulicaut, and O. Gandrillon. Clustering formal concepts to discover biologically relevant knowledge from gene expression data. *In Silico Biology*, 7(4-5):467–83, 2007.

[9] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of nfa. In *Proceedings of the 21st international jont conference on Artifical intelligence*, IJCAI'09, pages 1004–1009, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[10] W. Boomsma, J. Kent, K. Mardia, C. Taylor, and T. Hamelryck. Graphical models and directional statistics capture protein structure. *Interdisciplinary Statistics and Bioinformatics*, pages 91–94, 2006.

[11] E. Brändle. Bachelor thesis: A framework for attribute-constrained clustering, 2010.

[12] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*, pages 139–152, London, UK, 1994. Springer-Verlag.

[13] R. C. Carrasco and J. Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO (Theoretical Informatics and Applications)*, 33:1–20, 1999.

[14] J. Castro and R. Gavaldà. Towards feasible PAC-learning of probabilistic deterministic finite automata. In *Proceedings of the 9th international colloquium on Grammatical Inference: Algorithms and Applications*, ICGI '08, pages 163–174, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] L. Cerf, J. Besson, C. Robardet, and J.-F. Boulicaut. Closed patterns meet n-ary relations. *ACM Transactions on Knowledge Discovery from Data*, 3:1–36, 2009.

[16] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of the Twelfth International Conference on Data Engineering (ICDE)*, pages 106–114. IEEE Computer Society, 1996.

[17] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support. In *IDEAS '03: Proceedings of the 7th International Database Engineering and Applications Symposium 2003*, pages 111–116. IEEE Computer Society, 2003.

[18] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *Proceedings of the fourth IEEE International Conference on Data Mining*, pages 59–66. IEEE Computer Society, 2004.

[19] K. C. Chipman and A. K. Singh. Using stochastic causal trees to augment Bayesian networks for modeling eQTL datasets. *BMC Bioinformatics*, 12:7, 2011.

[20] D.-Y. Chiu, Y.-H. Wu, and A. Chen. Efficient frequent sequence mining by a dynamic strategy switching algorithm. *The VLDB Journal*, 18:303–327, 2009.

[21] A. Clark. Towards general algorithms for grammatical inference. In *Algorithmic Learning Theory*, pages 11–30, 2010.

[22] A. Clark and F. Thollard. PAC-learnability of probabilistic deterministic finite state automata. *Journal of Machine Learning Research*, 5:473–497, Dec. 2004.

[23] B.-R. Dai, C.-R. Lin, and M.-S. Chen. Constrained data clustering by depth control and progressive constraint relaxation. *The VLDB Journal*, 16(2):201–217, 2007.

[24] I. Davidson and S. Ravi. The complexity of non-hierarchical clustering with instance and cluster level constraints. *Data Mining and Knowledge Discovery*, 14:25–61, 2007.

[25] I. Davidson and S. S. Ravi. Clustering with constraints: Feasibility issues and the k-means algorithm. In *Proceedings of the fifth SIAM Data Mining Conference*, pages 138–149, 2005.

[26] I. Davidson and S. S. Ravi. Towards efficient and improved hierarchical clustering with instance and cluster level constraints. Technical report, State University of New York, Albany, 2005.

[27] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, September 2005.

[28] A. K. A. de Medeiros, B. F. van Dongen, W. M. P. van der Aalst, and A. J. M. M. Weijters. Process mining: Extending the α-algorithm to mine short loops. In *Beta Working Paper Series Eindhoven University of Technology*, 2004.

[29] C. Dima. Real-time automata. *Journal of Automata, Languages and Combinatorics*, 6(1):3–24, 2001.

[30] P. Dupont, F. Denis, and Y. Esposito. Links between probabilistic automata and hidden Markov models: probability distributions, learning models and induction algorithms. *Pattern Recognition*, 38(9):1349–1371, Sept. 2005.

[31] S. Džeroski, V. Gjorgjioski, I. Slavkov, and J. Struyf. Analysis of time series data with predictive clustering trees. In *Proceedings of the 5th international conference on knowledge discovery in inductive databases*, KDID'06, pages 63–80. Springer-Verlag, 2007.

[32] A. Floratou, S. Tata, and J. M. Patel. Efficient and accurate discovery of patterns in sequence datasets. In *ICDE 2010: Proceedings of the 26th International Conference on Data Engineering*, pages 461–472. IEEE Computer Society, 2010.

[33] B. J. Frey and N. Jojic. A comparison of algorithms for inference and learning in probabilistic graphical models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27:1392 – 1416, 2005.

[34] N. Friedman, I. Nachman, and D. Peér. Learning bayesian network structure from massive datasets: The "sparse candidate" algorithm. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 206–215, 1999.

[35] M. Fukuzaki, M. Seki, H. Kashima, and J. Sese. Finding itemset-sharing patterns in a large itemset-associated graph. In *Proceedings of the 14th Pacific-Asia conference on Advances in Knowledge Discovery and Data (2)*, volume 6119 of *Lecture Notes in Computer Science*, pages 147–159, 2010.

[36] R. Gavaldà, P. W. Keller, J. Pineau, and D. Precup. PAC-learning of Markov Models with hidden state. In *17th European Conference on Machine Learning 2006 (ECML 2006)*, pages 150–161, 2006.

[37] H. Gensler. *Introduction to Logic*. Routledge, 2001.

[38] Z. Ghahramani. Learning dynamic Bayesian networks. In *Adaptive Processing of Sequences and Data Structures*, pages 168–197. Springer-Verlag, 1998.

[39] A. Hafez, J. Deogun, and V. V. Raghavan. The item-set tree: A data structure for data mining. In *DaWaK '99: Data Warehousing and Knowledge Discovery*, pages 183–192. Springer, 1999.

[40] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2006.

[41] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD international conference on management of data*, pages 1–12. ACM, 2000.

[42] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer, 2 edition, 2009.

[43] C. D. L. Higuera and J. Oncina. Learning probabilistic finite automata. In *Grammatical Inference: Algorithms and Applications, Proceedings of the International Conference on Grammatical Inference (ICGI 04), volume 3264 of LNAI*, pages 175–186. Springer-Verlag, 2004.

[44] C. D. L. Higuera, J. Oncina, and E. Vidal. Identification of DFA: data-dependent vs. data-independent algorithms. In *Proceedings of the 3rd International Colloquium on Grammatical Inference: Learning Syntax from Sentences*, pages 313–325. Springer-Verlag, 1996.

[45] J. A. Hoeting, D. Madigan, A. E. Raftery, and C. T. Volinsky. Bayesian model averaging: A tutorial. *Statistical science*, 14(4):382–417, 1999.

[46] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, (1):193–218, 1985.

[47] J. Kalbfleisch. *Probability and Statistical Inference: Vol. 2: Statistical Inference.* Springer, 1985.

[48] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data. An introduction to cluster analysis.* Wiley, 1990.

[49] K. Kersting, T. Raiko, S. Kramer, and L. De Raedt. Towards discovering structural signatures of protein folds based on logical Hidden Markov Models. In R. Altman, A. Dunker, L. Hunter, T. Jung, and T. Klein, editors, *Proceedings of the Pacific Symposium on Biocomputing (PSB-03)*, pages 192–203. World Scientific, 2003.

[50] H. A. Kestler, J. M. Kraus, G. Palm, and F. Schwenker. On the effects of constraints in semi-supervised hierarchical clustering. In *Artificial Neural Networks in Pattern Recognition*, pages 57–66. Springer, 2006.

[51] R. Kindermann and J. L. Shell. *Markov Random Fields and Their Applications (Contemporary Mathematics; V.1).* American Mathematical Society, 1980.

[52] R. Küffner, T. Petri, L. Windhager, and R. Zimmer. Petri nets with Fuzzy Logic (PNFL): Reverse engineering and parametrization. *PLoS One*, 5:e12807, 09 2010.

[53] D. Lee and W. Lee. Finding maximal frequent itemsets over online data streams adaptively. In *Proceedings of the Fifth IEEE International Conference on Data (ICDM 2005)*, pages 266–273, 2005.

[54] H.-S. Lee. Incremental association mining based on maximal itemsets. In R. Khosla, R. J. Howlett, and L. C. Jain, editors, *KES (1)*, volume 3681 of *Lecture Notes in Computer Science*, pages 365–371. Springer, 2005.

[55] C. K.-S. Leung, Q. I. Khan, Z. Li, and T. Hoque. Cantree: a canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311, 2007.

[56] E. Levin and R. Pieraccini. Dynamic planar warping for optical character recognition. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 149–152. IEEE Computer Society, 1992.

[57] H. Li, Z. Wei, and J. Maris. A hidden Markov random field model for genome-wide association studies. *Biostatistics*, 11(1):139–150, Jan. 2010.

[58] H.-F. Li, S.-Y. Lee, and M.-K. Shan. Online mining (recently) maximal frequent itemsets over data streams. In *Proceedings of the 15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications*, RIDE '05, pages 11–18, 2005.

[59] W. Lian, D. W. Cheung, and S. M. Yiu. Maintenance of maximal frequent itemsets in large databases. In *Proceedings of the 2007 ACM Symposium on Applied Computing*, SAC '07, pages 388–392. ACM, 2007.

[60] H.-A. Loeliger. An Introduction to factor graphs. *IEEE Signal Processing Magazine*, 21(1):28–41, Jan. 2004.

[61] M. M. Masud, T. Al-Khateeb, L. Khan, C. C. Aggarwal, J. Gao, J. Han, and B. M. Thuraisingham. Detecting recurring and novel classes in concept-drifting data streams. In D. J. Cook, J. Pei, W. Wang, O. R. Zaïane, and X. Wu, editors, *11th IEEE International Conference on Data Mining, ICDM 2011*, pages 1176–1181, 2011.

[62] B. Merialdo, J. Jiten, and B. Huet. Multi-dimensional dependency-tree hidden Markov models. In *ICASSP 2006, 31st IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2006.

[63] B. Mozafari, H. Thakkar, and C. Zaniolo. Verifying and Mining Frequent Patterns from Large Windows over Data Streams. In *ICDE '08:*

Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, pages 179–188. IEEE Computer Society, 2008.

[64] M. Mueller and S. Kramer. Integer linear programming models for constrained clustering. In B. Pfahringer, G. Holmes, and A. Hoffmann, editors, *Discovery Science*, volume 6332 of *Lecture Notes in Computer Science*, pages 159–173. Springer, 2010.

[65] H. Nam, K. Lee, and D. Lee. Identification of temporal association rules from time-series microarray data sets. *BMC Bioinformatics*, 10(Suppl 3):S6, 2009.

[66] A. Omari, R. Langer, and S. Conrad. Tartool: A temporal dataset generator for market basket analysis. In *Advanced Data Mining and Applications*, volume 5139 of *Lecture Notes in Computer Science*, pages 400–410. Springer, 2008.

[67] D. Patnaik, P. Butler, N. Ramakrishnan, L. Parida, B. J. Keller, and D. A. Hanauer. Experiences with mining temporal event sequences from electronic medical records: initial successes and some challenges. In C. Apté, J. Ghosh, and P. Smyth, editors, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2011)*, pages 360–368, 2011.

[68] J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference.* Morgan Kaufmann Publishers Inc., 1988.

[69] H.-K. Peng, P. Wu, J. Zhu, and J. Y. Zhang. Helix: Unsupervised grammar induction for structured activity recognition. In D. J. Cook, J. Pei, W. Wang, O. R. Zaïane, and X. Wu, editors, *11th IEEE International Conference on Data Mining, ICDM 2011*, pages 1194–1199, 2011.

[70] R. Pensa, J.-F. Boulicaut, F. Cordero, and M. Atzori. Co-clustering numerical data under user-defined constraints. *Statistical Analysis and Data Mining*, 3(1):38–55, 2010.

[71] R. Pensa, C. Robardet, and J. F. Boulicaut. Constraint-driven co-clustering of 0/1 data. In S. Basu, I. Davidson, and K. Wagstaff, editors, *Constrained Clustering: Advances in Algorithms, Theory and Applications*, pages 123–148. Chapman & Hall/CRC Press, 2008.

[72] L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. In *Proceedings of the IEEE Readings in speech recognition*, volume 77, pages 257–286. Morgan Kaufmann Publishers Inc., 1989.

[73] D. Ron, Y. Singer, and N. Tishby. On the learnability and usage of acyclic probabilistic finite automata. In *COLT '95 Proceedings of the eighth annual conference on computational learning theory*, pages 31–40. ACM Press, 1995.

[74] M. Rowicka, A. Kudlicki, B. P. Tu, and Z. Otwinowski. High-resolution timing of cell cycle-regulated gene expression. *Proceedings of the National Academy of Sciences of the United States of America*, 104(43):16892–16897, 2007.

[75] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases*, pages 432–444. Morgan Kaufmann, 1995.

[76] J. Schmidt, S. Ansorge, and S. Kramer. Scalable induction of probabilistic real-time automata using maximum frequent pattern based clustering. In *Proceedings of the twelfth SIAM International Conference on Data Mining*, SIAM DM'12, pages 272–283. Omnipress, 2012.

[77] J. Schmidt, E. Braendle, and S. Kramer. Clustering with attribute-level constraints. In *Proceedings of the 2011 IEEE International Conference on Data Mining*, pages 1206 – 1211. Springer-Verlag, 2011.

[78] J. Schmidt, A. Ghorbani, A. Hapfelmeier, and S. Kramer. Learning probabilistic real time automata from multi attribute event logs. *Intelligent Data Analysis - Special Issue*, 7(1), 2013.

[79] J. Schmidt, A. Hapfelmeier, W.-D. Schmidt, and U. Wollina. Improving wound score classification with limited remission spectra. *International Wound Journal*, 9(2):189–98, 2012.

[80] J. Schmidt and S. Kramer. The augmented itemset tree: a data structure for online maximum frequent pattern mining. In *Proceedings of the 14th International Conference on Discovery Science*, DS'11, pages 277–291. Springer-Verlag, 2011.

[81] J. Schmidt and S. Kramer. Learning Probabilistc Real Time Automata From Multi-Attribute Event Logs. TUM Technical Report TUM-I1117, 2011.

[82] J. Schmidt and S. Kramer. Online induction of probabilistic real time automata. In B. Goethals and G. Webb, editors, *Proceedings of the IEEE International Conference on Data Mining, 2012*, pages 625–634. IEEE Computer Society, 2012.

[83] M. Seeland, T. Girschick, F. Buchwald, and S. Kramer. Online structural graph clustering using frequent subgraph mining. In J. Balcazar, F. Bonchi, A. Gionis, and M. Sebag, editors, *Proceedings of the European Conference of Machine Learning and Principles and Practice of Knowledge Discovery in Databases*, volume 3, pages 213–228, 2010.

[84] J. Sempere and P. García. A characterization of even linear languages and its application to the learning problem. In R. Carrasco and J. Oncina, editors, *Grammatical Inference and Applications*, volume 862 of *Lecture Notes in Computer Science*, pages 38–44. Springer Berlin/Heidelberg, 1994.

[85] J. Sese and S. Morishita. Itemset classified clustering. In *Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, PKDD '04, pages 398–409. Springer, 2004.

[86] S. Siddiqi, G. J. Gordon, and A. Moore. Fast state discovery for HMM model selection and learning. In *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*, pages 40–47, 2007.

[87] M. Stoelinga. An introduction to probabilistic automata. *Bulletin of the European Association for Theoretical Computer Science*, 78:176–198, 2002.

[88] F. Thollard, P. Dupont, and C. d. l. Higuera. Probabilistic DFA inference using Kullback-Leibler divergence and minimality. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 975–982. Morgan Kaufmann Publishers Inc., 2000.

[89] A. K. H. Tung, R. T. Ng, L. V. S. Lakshmanan, and J. Han. Constraint-based clustering in large databases. In *Proceedings of the 8th International Conference on Database Theory*, pages 405–419. Springer, 2001.

[90] P. Valtchev, R. Missaoui, and R. Godin. A framework for incremental generation of closed itemsets. *Discrete Applied Mathematics*, 156:924 – 949, 2008.

[91] P. Valtchev, R. Missaoui, R. Godin, and M. Meridji. Generating frequent itemsets incrementally: two novel approaches based on galois lattice theory. *Journal of Experimental & Theoretical Artificial Intelligence*, 14(2-3):115–142, 2002.

[92] W. van der Aalst, B. van Dongen, C. Günther, R. S. Mans, A. K. A. de Medeiros, A. Rozinat, V. Rubin, M. Song, H. M. W. E. Verbeek,

and A. J. M. M. Weijters. ProM 4.0: Comprehensive support for real process analysis. In J. Kleijn and A. Yakovlev, editors, *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 484–494. Springer, 2007.

[93] W. van der Aalst, B. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2):237 – 267, 2003.

[94] W. van der Aalst, T. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16:1128–1142, September 2004.

[95] S. Verwer and M. De Weerdt. An algorithm for learning real-time automata. In M. van Someren, S. Katrenko, and P. Adriaans, editors, *Proceedings of the Sixteenth Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, pages 128–135, 2007.

[96] S. Verwer, M. de Weerdt, and C. Witteveen. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In J. M. Sempere and P. García, editors, *ICGI'10 Proceedings of the 10th international colloquium conference on grammatical inference: theoretical results and applications*, Lecture Notes in Computer Science, pages 203–216, 2010.

[97] S. Verwer, M. de Weerdt, and C. Witteveen. The efficiency of identifying timed automata and the power of clocks. *Special Issue: 3rd International Conference on Language and Automata Theory and Applications (LATA 2009)*, 209(3):606–625, 2011.

[98] S. E. Verwer, M. M. de Weerdt, and C. Witteveen. Identifying an automaton model for timed data. In Y. Saeys, E. Tsiporkova, B. D. Baets, and Y. van de Peer, editors, *Proceedings of the Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, pages 57–64, 2006.

[99] S. E. Verwer, M. M. de Weerdt, and C. Witteveen. Efficiently learning simple timed automata. In W. Bridewell, T. Calders, A. K. de Medeiros, S. Kramer, M. Pechenizkiy, and L. Todorovski, editors, *Proceedings of the Second International Workshop on the Induction of Process Models at ECML PKDD*, pages 61–68, 2008.

[100] K. Wagstaff, C. Cardie, S. Rogers, and S. Schroedl. Constrained k-means clustering with background knowledge. In *Proceedings of 18th International Conference on Machine Learning (ICML-01)*, pages 577–584. Morgan Kaufmann, 2001.

[101] K. L. Wagstaff. Value, cost, and sharing: open issues in constrained clustering. In *KDID'06: Proceedings of the 5th international conference on knowledge discovery in inductive databases*, pages 1–10. Springer, 2007.

[102] L. Wen, J. Wang, and J. Sun. Detecting implicit dependencies between tasks from event logs. In X. Zhou, J. Li, H. Shen, M. Kitsuregawa, and Y. Zhang, editors, *Frontiers of WWW Research and Development - APWeb 2006*, volume 3841 of *Lecture Notes in Computer Science*, pages 591–603. Springer, 2006.

[103] L. Wen, J. Wang, and J. Sun. Mining invisible tasks from event logs. In G. Dong, X. Lin, W. Wang, Y. Yang, and J. Yu, editors, *Advances in Data and Web Management*, volume 4505 of *Lecture Notes in Computer Science*, pages 358–365. Springer, 2007.

[104] L. Windhager and R. Zimmer. Intuitive modeling of dynamic systems with Petri nets and fuzzy logic. In *German Conference on Bioinformatics*, pages 106–115, 2008.

[105] H. Yao, C. Butz, and H. Hamilton. Causal discovery. In *The Data Mining and Knowledge Discovery Handbook*, volume 9, pages 945–955. Springer, 2005.

[106] M. Young-Lai, F. W. Tompa, and R. Mooney. Stochastic grammatical inference of text database structure. *Machine Learning*, 40:111–137, August 2000.