University of Economics, Prague
Faculty of Informatics and Statistics
Department of Information and Knowledge Engineering

Finding Optimal Decision Trees

PhD Thesis

PhD student : Petr Máša
Supervisor : Doc. RNDr. Jiří Ivánek, CSc.
Study plan : Informatics

For citations, please use the following reference: *Máša, P.: Finding Optimal Decision Trees, PhD Thesis, University of Economics-FIS, Prague, 2006*

Prague, 2006

Prohlášení

Prohlašuji, že doktorskou práci na téma "Finding Optimal Decision Trees" jsem vypracoval samostatně. Použitou literaturu a podkladové materiály uvádím v přiloženém seznamu literatury.

V Praze dne 25. června 2006

Acknowledgements

I would like to thank to many people. Without them, this work would not have been realized. It is almost impossible to name all of them. Nevertheless, I would like to thank to some people in particular.

First, I would like to thank to my supervisor, Doc. RNDr. Jiří Ivánek, CSc. for tutoring and for help during my entire PhD study, including this thesis. He was giving me very valuable advices, freedom to my interests, patience to combine my doctoral study with my job and has provided me support and advices I needed. Many suggestions allowed improving this thesis, including readability and understandibility.

Next, I would like to thank to Ing. Tomáš Kočka, Ph.D. for kindness and endless help with this research any time. This began with discovering problem, which is solved in this thesis. His advices have shown the direction of this research. His large support allowed this work to be realized.

Last mentioned but not least are my friends and parents. I would like to thank them for their patience, understanding and support.

Abstract

Decision trees are widely used technique to describe the data. They are also used for predictions. We may find that the same distribution can be described by one or more decision trees. Usually, we are interested in the simplest tree (we will call it the optimal tree).

This thesis proposes the extension to the post-pruning phase of the standard decision tree algorithm, which allows more prunes. We will study theoretical and practical properties of this extended algorithm. In theory, we have described a class of distributions, for which we have proven that the optimal tree is always found by the algorithm.

In practical tests, we have studied whether this algorithm is able to reconstruct a known tree from the data. We were interested mainly whether there is an improvement in number of correct reconstructions at least for data which are large enough. This fact was confirmed by a set of tests.

So we can expect that with a growing number of records in a dataset we will achieve more true reconstructions of decision trees from data by the algorithm. A similar result was recently published for Bayesian networks. Our proposed algorithm is polynomial in the number of leaves of the tree which is the result of the growing phase, comparing to the exponential complexity of a trivial exhaustive search (brute force search) algorithm.

The proposed algorithm was tested on both simulated and real data and it was compared over the traditional CART algorithm. Our algorithm achieves better results for both real and simulated data.

The structure of PhD thesis is the following. Chapter 1 is a brief introduction to data analysis and data mining in a wider context. Chapter 2 is a brief introduction to decision trees. Chapter 3 describes the problem solved in more detail, goals of this thesis and the related work. Note that chapters 1 to 3 describe also the state of the art in the research area. Chapter 4 is the core chapter of PhD thesis. It includes formal definitions, the restriction on

the set of distributions, theorems, the algorithm itself and the proof that this algorithm finds the optimal decision tree under certain assumptions. Chapter 5 shows the comparison of the proposed algorithm with the CART algorithm on both simulated and real data. Finally, Chapter 6 concludes this thesis and briefly summarizes results achieved.
Author's original results include Chapters 4 and 5.

Key words: Decision Tree, Optimality, Smallest Decision Tree, Restructuring, Algorithm.

Abstrakt

Rozhodovací stromy jsou rozšířenou technikou pro popis dat. Používají se často také pro predikce. Zajímavým problémem je, že konkrétní distribuce může být popsána jedním či více rozhodovacími stromy. Obvykle nás zajímá co nejjednodušší rozhodovací strom (který budeme nazývat též optimální rozhodovací strom).

Tato práce navrhuje rozšíření prořezávací fáze algoritmů pro rozhodovací stromy tak, aby umožňovala více prořezávání. V práci byly zkoumány teoretické i praktické vlastnosti tohoto rozšířeného algoritmu. Jako hlavní teoretický výsledek bylo dokázáno, že pro jistou třídu distribucí nalezne algoritmus optimální rozhodovací strom (tj. nejmenší rozhodovací strom, který reprezentuje danou distribuci). V praktických testech bylo zkoumáno, jak je schopen algoritmus rekonstruovat známý strom z dat. Zajímalo nás, zdali dosáhne naše rozšíření zlepšení v počtu správně rekonstruovaných stromů zejména v případě, že data jsou dostatečně velká (z hlediska počtu záznamů). Tato domněnka byla potvrzena praktickými testy.

Obdobný výsledek byl před několika lety dokázán pro Bayesovské sítě. Algoritmus navržený v této disertační práci je polynomiální v počtu listů stromu, který je výstupem hladového algoritmu pro růst stromů, což je vylepšení oproti jednoduchému algoritmu prohledávání všech možných stromů, který je exponenciální.

Struktura disertační práce je následující. Kapitola 1 obsahuje stručný úvod do analýzy dat a data miningu v širším pojetí. Kapitola 2 je stručný úvod do problematiky rozhodovacích stromů. Kapitola 3 popisuje problém řešený v této disertační práci, cíle disertace a uvádí další práce, které řešily obdobné problémy.

Poznamenejme, že kapitoly 1 až 3 popisují zároveň stav výzkumu v dané oblasti. Kapitola 4 je jádrem disertační práce. Obsahuje formální definice, omezení, potřebná tvrzení (matematické věty včetně důkazů) a samotný algoritmus včetně důkazu, že tento algoritmus za vymezených podmínek nalezne optimální rozhodovací strom. Kapitola 5 srovnává navržený algoritmus se standardním CART algoritmem na simulovaných a reálných datech. Závěrečná kapitola 6 shrnuje přínosy práce a stručně též dosažené výsledky.

Vlastním přínosem autora k této problematice jsou uvedená tvrzení a algoritmus, tedy zejména kapitoly 4 a 5.

Klíčová slova: rozhodovací strom, optimalita, nejmenší rozhodovací strom, změna vnitřní struktury, algoritmus.

Abstrakt in Deutsch

Entscheidungsbäume sind erweiterte Technik für Datenbeschreibung. Sie werden oft auch für Prediktionen benutzt. Das interessante Problem ist, dass eine konkrete Distribution durch einen oder mehrere Entscheidungsbäume beschrieben werden kann. Es interessiert uns gewöhnlich den einfachsten Entscheidungsbaum (der auch als optimaler Entscheidungsbaum genannt wird).

Diese Arbeit schlägt Erweiterung der durchgeschnittenen Phase der Algorithmen für Entscheidungsbäume so vor, damit sie mehr Durchschneidung ermöglicht. In der Arbeit wurden teoretische und praktische Eigenschaften dieses erweiterten Algorithmus geprüft. Als haupttheoretisches Ergebniss wurde bewiesen, dass für bestimmte Distributionsklasse der Algorithmus den optimalen Entscheidungsbaum findet (das heisst den kleinsten Entscheidungsbaum, der die entsprechende Distribution repräsentiert). In praktischen Testen wurde untersucht, wie der Algorithmus fähig ist, einen bekannten Baum von Daten zu rekonstruieren. Es hat uns interessiert, ob unsere Erweiterung Verbesserung der Zahl richtig rekonstruierten Bäume erreicht vor allem im Fall, dass die Daten ausreichend gross sind (aus der Sicht der Zahl der Anmerkungen). Diese Vermutung wurde durch praktische Teste bestätigt.

Analogisches Ergebniss wurde vor einigen Jahren für Bayesnetze bewiesen. Der in dieser Disertationsarbeit vorgeschlagene Algorithmus ist polynomial in der Zahl der Blätter des Baums, der der Ausgang des hungrigen Algorithmus für Wachstum der Bäume ist, was eine Verbesserung im Vergleich zum einfachen Algorithmus der Durchsuchung aller möglichen Bäume ist, der exponential ist.

Diese Struktur der Disertationsarbeit ist folgendes. Das Kapitel 1 beinhaltet kurzfassende Einführung in Datenanalysis und Datenmining in breiterer Auffassung. Das Kapitel 2 ist kurzfassende Einführung in Problematik der Entscheidungsbäume. Das Kapitel 3 beschreibt das in der Disertationsarbeit gelöste Problem, die Ziele der Disertation und stellt andere Arbeiten vor, die analogische Probleme gelöst haben. Bemerken wir, dass die Kapitel 1 bis 3 zugleich den Forschungszustand in gegebenem Gebiet beschreiben. Das Kapitel 4 ist der Kern der Disertationsarbeit. Es beinhaltet formale Definitionen, Einschränkungen, nötige Behauptungen (mathematische Sätze einschliesslich Beweise) und den eigentlichen Algorithmus einschliesslich Beweise, dass dieser Algorithmus unter begrenzten Bedingungen den optimalen

Entscheidungsbaum findet. Das Kapitel 5 vergleicht den vorgeschlagenen Algorithmus mit Standard CART Algorithmus für simulierte und reale Daten. Das letzte Kapitel 6 fasst Beiträge der Arbeit und kurz auch erreichte Ergebnisse zusammen.

Der eigene Beitrag des Authors zu dieser Problematik sind eingeführte Behauptungen und Algorithmus, also vor allem die Kapitel 4 und 5.

Schlüsselwörter: Entscheidungsbaum, Optimalität, kleinster Entscheidungsbaum, Innenstrukturveränderung, Algorithmus

Contents

Chapter 1

Data Analysis and Data Mining

In this chapter, we will introduce what the data analysis is, its purpose and its typical applications. We will focus on the data mining, which is a set of techniques to mine knowledge from the data. Data mining has many applications. We will focus here on applications which are typical in business. Basic task types and major methods of the data mining will be enumerated and briefly described in this chapter. To be more specific, we will also introduce basic variable types and what format of data is entering into the data analysis. This chapter is about the data analysis in general. The chapter include topics like why we need the data analysis, where to get the data, where the data analysis can help, and what type of data usually goes into the data mining task.

1.1 Data analysis and its purpose

In many areas, including business and research, large amounts of data are collected. Analyses of these data can bring new business opportunities (e.g. discovering why our customers leaving our company, or discovering which customers buy some particular product). These can result in the increased profit, or the increased value of customers for the company, or the other business goal. In the medical research, discovered results can lead into, for example, introducing new curing methods.

1.1.1 Data analysis

In today's world, most of business companies already analyse the data. But some of them analyse its data only very briefly. The reason is that contemporary managers have their 'best practises', which were successful in the past and they want to apply them to present and future. But world is being changed. The successful data analysis can lead into focusing on most valuable customers and retaining them. The required level of data analysis depends on competitors. When the data analysis in one company is better, then this company can choose which customers from market they want, which products to sell them, which campaigns are cost effective and so on. Conversely, when the data analysis in this company is slower than competitive's one, then this company is starting to lose most valuable customers. In many cases, without the data analysis, the company is not able to recognise the value of customers which are leaving, so this company is not able to imagine the size of the problem.

1.1.2 Data sources

The data can be treated as a corporate asset. Many of business decisions are supported by the data. For the data analysis, the high quality and integrated data are required. To ensure that the data required by business users are available, up to date and in required quality, many of companies have started the data stewardship programme. From a view of a business user/data analyst, following major points are required for successful results of his or her job:

1. To know what data are available, what is the periodicity of their update.
2. The data required by analyses are available.
3. The data should have corresponding history (depends on tasks solved).
4. The data are in required quality.
5. The data are pre-processed and transformed into the form which is suitable for analyses.

The data in companies usually resides in two places. The first one is called legacy systems (or production systems, or transaction systems), which are

designed to ensure company (item-level) data access and core functions (for example, a bank has actual balances available, which have to be used when the withdrawal is made and then the actual balance is updated). The second place is called the data warehouse. Business users usually get data from the data warehouse. The data warehouse integrates data from various sources (legacy systems), cleans it and transforms it into the form required by business users. It also reduces the load of legacy systems (the only need is in predefined time give the information for the data warehouse). Most of analyses are based on data from the data warehouse, and therefore no additional load for legacy systems is required.
But the data warehouse is not the only source for data analysis. Typical sources for the data analysis include also:

1. External sources.
    (a) The information about Corporate Customers (from some registry, surveys, market analyses, market monitoring, . . .)
    (b) Some type of Bureau (e.g. Credit Bureau, Fraud Bureau, Previous Medical Care Information).
    (c) Competitors' information (from web and other sources), for example prices.
2. Other internal sources.
3. Internal analyses, estimations, predictions.

Sources may vary, depending on the quality and comprehensiveness of data contained in the data warehouse. In some cases, data from legacy systems are required, or data from data warehouses from the other companies in the same business group is very useful, if legal regulations allow it.

1.1.3 Leveraging profit from data

The data warehouse is usually the best source for data analyses. In most companies, large investments into the data warehouse were made. The data warehouse is usually understood as a general data source for analyses. Most of business information needed by business comes from the data warehouse. But there is also the other side of the coin. Large investments were made into the data warehouse and large investments still continues, but the effect of the data warehouse was smaller than the company had expected. There may be many reasons of it, which includes

- Data in the data warehouse are not complete.
- Business users do not know how to find data needed for their work.
- Business users do not understand data available in the data warehouse.
- Some crucial area is missing in the data warehouse.

The data analysis is usually the way how to convert effort of building the data warehouse into the profit. Many data analysis tasks would not be done without the data warehouse (because the data preparation would be very long and expensive). When launching a project of data analysis, we should focus on the ROI (Return On Investment). For many business tasks, we can estimate costs and benefits of the data analysis project, so we can measure the return on investment. Usually, profits are very high and they should be treated also as benefits of investments into the data warehouse. The data analysis is usually very hard without the data warehouse and the data warehouse without data analyses is usually an investment without the leveraged profit.

1.2 Typical Applications of the Data Analysis

Typical tasks solved by data analysis vary from company to company, from industry to industry and from country to country. The level of data analysis in a particular industry in a particular country usually depends on the level of data analysis of company's competitors. But there are some common applications of the data analysis.

The first typical application is customer retention. The *churn* (=customers leaving our company) is usually the big problem. When the market grows, usually no one cares about the churn because the churn rate is low and the customer acquisition is very large. A minor improvement in the customer acquisition is usually much better (=bigger profit) than a churn analysis. But one day we can found that the market became exhausted and divided to companies. In this situation, most of acquisitions means acquiring customers from competitors, and vice versa. So we have to manage the churn, analyse why our customers are leaving our company. Reducing the churn may very affect the number of customers. In some situations there is no increase in the number of customers when applying the churn management but it can be evaluated that without the churn management there would be a significant decrease in the number of customers.

The next applications of the data analysis are *propensity to buy* models (or *affinity models*). These models try to estimate which customers are willing to buy some product. Application of propensity to buy models is in selecting the target group for direct marketing campaigns. The typical situation is that the company has models for all key products. So it can say for each key product to who is good to offer this product. The more advanced application is that the company estimates the best product for the customer (we have moved from the product-oriented view into the customer-oriented view) and this is the one which the company offers him in a direct marketing campaign. The analysis of the best product for the customer is a very advanced analysis. It usually requires all propensity to buy models for individual products, profit analyses for products and the other information which also comes from the strategy of the company.

The *customer segmentation* is another typical application of the data analysis. When we have an individual (for example a self-employed person) which offers services to his customers, he usually knows his customers, he is able to estimate the profit from each individual customer, he knows their preferences and so on. So he can estimate who is likely to churn, to whom offer a new product and other business tasks. To apply the similar task in a company which has several millions of customers is a very hard issue. The customer segmentation is dividing customers into groups, within these groups customers have similar properties like behaviour, the value and preferences and between group customers have different properties (e.g. behaviour and preferences). The strategy can be established for every individual segment. We should better focus on high value segments, ask how to convert customers from low value segments into high value segments, to develop products for segments and so on.

Estimating the customer value is crucial for most of business companies. The company should focus on customers with the high value, build their retention and loyalty. In many cases, customers with a very low value are identified. Depending on the strategy, the company should choose from many alternatives what to do with them, include:

1. Reducing costs for these customers (deactivating some services, decreasing level of a customer care, . . .).
2. Let them leave to competitors.
3. Try to increase their value.

Note that the strategy 'Let them leave to competitors' is very sensitive. In the first phase, usually customers who generate a loss are leaving. When not using supplementary advanced analyses, then in a second phase many customers follow customers which left to competitors. When this strategy is chosen, it should be analysed what effect will be reached on other customers.
Another typical task is a *fraud detection.* Many transactions may be fraudulent. Fraud means the loss and the level of fraud should be very well managed. In insurance, estimations says that 10 - 15 % of claims are fraud. In telecommunications, many calls are realised and interconnect fees paid to partners (including roaming), but not billed, incorrectly billed or never paid on overdrafted prepaids. This behaviour is classified as a fraud only in situation that it was done intentionally, with the knowledge that the customer misuse some bug in a company process. A fraud analysis is crucial, because it can become very large and it can be critical for the company.
The *Credit Risk* is a next application of the data analysis. This problem is solved in banks and other industry where customer pays their bills on a postpaid base. Banks need to know the level of the risk that customer will not repay the loan when deciding whether to accept or reject the loan for a customer. Also, according to the Basel II (a prepared standard for calculating requirements on own capital), they need to estimate the risk of not repaying for loans already provided – banks need to know how much capital they need to cover risks from not repaying loans in total. Another industries, where the credit risk applies, include telecommunication operators, utility providers (gas, electricity), which provides the service and the billing is done after a specified period, and includes regular fees and fees for consumed services. When consumption is high, the risk of not paying the bill is more critical for the company.

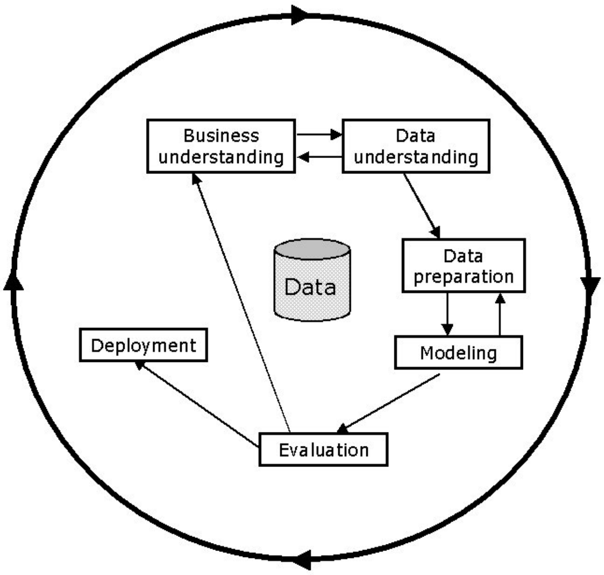


Figure 1.1: Phases of the CRISP-DM process ([4],[12]).

1.3 Methodology CRISP-DM

The abbreviation CRISP-DM stands for *the CRoss-Industry Standard Process for Data Mining.* This methodology describes a complete process of the data mining application, beginning with the business understanding and ending with the evaluation and the deployment. This methodology can be used as a cookbook for the data analysis or the data mining project. The complete CRISP-DM methodology is available in [12]. We show here only a brief introduction. The basic overview is shown in Figure 1.1.

1.3.1 Individual tasks

In the following text, we briefly describe phases of the CRISP-DM methodology.

Business understanding – This phase emphasises the understanding to business and its needs. The definition of the problem and understanding to business processes are crucial things to do. Only a little misunderstanding or a neglecting of issue that we may find useless may lead into delivering a solution which does not solve our problem. In a business understanding, it is good to recognise business processes and estimate a future use of the prepared data mining solution. The good question in this phase is ”If we had this solution, how would we use it?” We should also formulate an exact definition of the problem and the planned solution in a language of business (in language, to which business users understand).

Data understanding – In this phase, we should focus on data sources available. In this phase, we should explore, which data sources are available, to obtain the data, to profile the data, attributes and their values, to identify problems with the data (quality, availability, checking whether the data are up-to-date, periodicity of data refreshment) and to exactly formulate the problem in the language of the data (to extend the problem definition with specific attribute names and values).

Data preparation – Data mining methods require the data in a specified form, called a *flat table*. Some algorithms for the data mining have particular requirements on data. The data preparation includes all tasks which ensures that the data will be available in the flat table. The flat table and its structure will be described in a more detail in the subsection 1.7.

Modeling – This phase contains of the application of the data mining algorithm(s). In this phase, the knowledge is mined from the data. Therefore, in some applications, the term *data mining* is used only for the modeling phase.

Evaluation – Ensuring that results of modeling are usable for the business application is a very crucial thing. The evaluation phase also deals with consequences of data quality problems. This phase should find any mistakes we could have made during all previous phases.

Deployment – In this phase, we should ensure that results of the data mining process will help to solve our original problem. This phase differs from task to task. For several tasks, business processes are adjusted with respect to the results of previous phases, for other group of tasks, deployment phase include an integration of results into current data warehouse tables.

Note that the data mining is an iterative process. Results achieved in one phase may force us to return to the previous phase.

1.4 Basic Data Analysis Tasks

We can identify some basic data analysis tasks (with respect to the data mining). These types include

1. Data Description
2. Data Exploration
3. Accepting or rejecting hypotheses
4. Classification/Prediction
5. Clustering

In the *Data Description* task, the goal is usually to organise and visualise the data, maybe in a form in which the data were not displayed and this can bring a new knowledge to the organisation. The data description task is a typical task of the data analysis. Many of these results become to be regular reports. The data description task has various applications, including a costs analysis and a customer behaviour analysis.

The *Data Exploration* task is a task where the only goal is to find something that will help to solve some (usually given) business problem. The data exploration is a very interesting task, but it is not so typical, because there is no guarantee that investments into this data analysis will bring any results/benefits. It is often used in companies with very large data analysis activities to improve the common set of data analyses.

The *accepting or rejecting hypotheses* task usually starts with some hypothesis (based on manager's intuition) and tries to find a suitable data which can say whether the hypothesis is correct or not. This task has various applications in most of major business activities in a company.

Classification and prediction tasks usually estimate some value for each record (customer, claim). The definition of the record depends on a business task solved. The classification estimates a current value, the prediction estimates a future value. The prediction task needs some time shifts and evaluation may take longer time. Typical applications include the churn prediction, the propensity to buy prediction, the credit scoring and the fraud classification. The *clustering* task aggregates records (customers, . . .) into groups, there are records with similar properties within groups, and the properties of records are different between groups. Typical applications include the customer behaviour segmentation, the customer value segmentation and segmentations affiliated to a prediction model (classifying reasons for the predicted behaviour).

1.5 Basic methods for Data Mining

The data mining is a collection of various methods. For a particular business task, an individual preparation and method selection should be done. There is no strict rule which method should be used in a particular situation. The origin of individual methods is very different. Data mining methods have taken methods from statistics, machine learning and artificial intelligence. In following paragraphs, we will describe some common methods.

The first method we mention here is a *decision tree* induction. It has been chosen as the first method intentionally – decision trees are what is this work about. Here we provide only a brief description. Decision trees will be described in a more detail later. The decision tree induction is a method based on two principles. The first principle is called *divide et impera* (i.e. divide and conquer). This means, that in every step, the dataset is split into two or more parts and the algorithm continues recursively on individual parts. The second principle is a greedy principle. That means that the splitting is based only on little information (the best local decision). Decision trees are used mainly for predictions, classifications and descriptions (the data visualisation). The decision tree is a classifier with a very high capacity. The capacity means how much information from the data (e.g. number of parameters) can this classifier store; the capacity is also called the *VC Dimension*, or the *Vapnik-Chervonenkis Dimension* ([9],[31]). For a given capacity and the test set size, we are able to estimate the error on the test set ([33]). So the decision tree can represent perfectly any distribution. But, in real applications, decision trees may tend to overfitting. There are some techniques in decision trees which prevents overfitting, for example pruning. Decision trees are also often used in pre-processing phase for a logistic regression (see later). Decision trees with a logistic regression are major techniques used for prediction and classification tasks.

The *logistic regression* ([1],[23]) is a method which comes from the statistics, but it is often used in data mining applications. The main purpose of this method is to predict (to classify) a binary (categorical) variable. The logistic regression has smaller capacity than decision trees. For real applications, the risk of overfitting is not so high, but the logistic regression is not able to fit some distributions (the most of them). For example the *xor* distribution can be never fitted. This problem can be avoided using interactions (derived attributes, combination of attributes), a selection of interactions used can be done using a manual application of decision trees. The logistic regression has a very wide range of applications. The most of credit-scoring systems are based on a logistic regression.

The *support vector machine* (SVM; [15]) is a relatively new method for the prediction and the classification. This method is suitable in a situation where one target class has only a very small number of records. The capacity of the SVM is a very small, i.e. the SVM can represent only a very small number of distributions. But it is the only method which is able to handle even a very small data. Currently, the SVM is not implemented in a most of leading statistical software packages.

The *K-nearest neighbours* is an another classification method. This method is based on the principle "When I do not know, I will look to the historical data to k most similar cases and their results and based on these results, I will estimate the result for the new record". This method is very simple, but in general, this method is not used so often.

The *perceptron*, or a three layer predictive neural network, is another method, which can be used for the prediction. Neural networks were popular in 80's. Today's world uses neural networks less than other prediction methods, although they have a similar prediction quality as decision trees or the logistic regression. The reason is that the neural network is a black box.

The most common clustering method is the *K-means clustering.* This is a distance-based clustering method, i.e. the distance measure between records is defined. The K-means clustering is an iterative method, where the number of clusters must be specified. The K-means clustering is a widely used method in a situation, where we have only a few attributes (less than 10) and all attributes are important in all cluster profiles. Applications of the K-means clustering include a demographical segmentation and supplementary segmentations for prediction models.

A special kind of neural networks is the *Kohonen neural network*, also known as the *Kohonen self organising map*. This is a distance-based method, but it can handle a bigger number of attributes than the K-means do. Also, the number of final clusters is estimated from the data and final clusters do not need to be defined by spherical regions only (a cluster is defined as an union of spheres).

The next clustering method is called the *EM clustering*, or the *model based clustering*. This method is based on the probability and it can handle more attributes. The profile of each cluster is determined only by attributes which are different from population means, two clusters may have different attributes in their definitions. This method needs to have the data very well prepared. The EM clustering is not implemented in the most of major data mining packages. Applications of the EM clustering include the corporate customer segmentation and the supplementary segmentation for prediction models.

The *association rule analysis* is another data mining method. For a given data, co-occurring items or items matching predefined pattern (in case of the general rule induction) are found. The data exploration is one of typical applications of association rules. Business applications include the market basket analysis and the web tracking analysis.

The *discriminant analysis* is a statistical method. This method was often used for a classification, but nowadays it becomes popular as a supplementary tool for the clustering (segmentation). The typical application is using it as a tool which helps to visualise the clustering.

1.6 Types of Attributes

In many database engines, attribute types are numeric, character (fixed or variable length), binary, large binary, picture and others. For the data analysis, other classes of attributes are used. We will describe these classes and their properties.

1.6.1 Attribute Types for Data Analysis

The first type of attributes used in data analyses is a *binary attribute*. This attribute can have only two values. It can be coded in a numeric or a character variable. Without loss of generality, we may assume that the attribute is numeric and it has values 0 and 1.

The next type is a *categorical attribute*, also known as a *discrete attribute*. The value of this attribute can be one of the limited number of values. Usually, there is only a small number of possible values. The binary attribute is a special type of a categorical attribute, which can have only one of two values.

The special subtype of categorical attributes is an *ordinal attribute*. Values of this attribute type can be also only from a limited list, but values are ordered. We can compare two values and we can say what is more and what is less.

The last major type of attribute is a *continuous attribute*. This attribute is similar to a real number in a computer. Many attributes with numeric values, but a large possible number of these values, are treated as continuous.

1.6.2 Conversion between attribute types

Discretization

The most common conversion is the conversion from the continuous attribute to a discrete attribute. This conversion is called the *discretization*. Many algorithms are not able to handle with continuous variables, so the discretization is needed. In several cases, the discretization is used instead of a large amount of steps in the data preparation, including the outlier handling, the variable transformation, the variable analysis and adjusting.

Assume that we have the data about accounts in a bank and we are interested in a continuous variable *actual balance*. This variable has a strongly asymmetric distribution of values and many outliers (for many data mining methods, values which are very far from other values have to be handled, otherwise they spoil the entire analysis). The basic idea of the discretization is to handle big differences in values. We usually do not care whether client's balance is $1,234.50 or $1,237.30, but the difference between $1.28 and $10,354,583.94 is significant for us. So we split the range of possible values in smaller intervals (as shown in the example in Table 1.1) and we say that client's balance is for example *low* or *very high*.

The discretization can be set by the expert (or boundaries which respect to some division which is commonly used) or can be set automatically. The most common attitude is to use equifrequent bins. This means that we set the required number of bins (categories) and boundaries are set automatically using the following way: in every bin there is the same (or very similar)

Table 1.1: Example of discretization, account balance, retail

Balance range	Balance level
less than \$0.00	overdrawn
\$0.00 – \$29.99	very low
\$30.00 – \$999.99	low
\$1,000.00 – \$2,999.99	lower medium
\$3,000.00 – \$7,999.99	upper medium
\$8,000.00 – \$19,999.99	high
\$20,000.00 and more	very high

number of records. There are usually 10 bins used, because 10 bins is enough to see large differences in values and 10 bins can be easily handled by the human's brain and visualised.

There are some other attitudes to the discretization. One of them is to set intervals in a way that every interval has the same range (the difference between lower and upper bound). The other attitude is to use percentile-based division, for example in a schema 10-25-50-75-90. This means that the lowest 10 percent of values goes to the lowest bin, the next 15 percent of values goes to the second bin (and cumulatively there is 25 percent of values in first two bins) and so on.

The discretization is commonly used method. It clears small differences, handles outliers and may handle missing values in one step (by introducing special bin ”missing”). The next purpose is for algorithms which require discrete variables only (with the small number of possible values) and we have continuous ones which we want to use. But the discretization has also disadvantages. For example, two almost similar values can be put into different bins if the boundary fits between them. The main disadvantage is that boundaries are *artificial* – we introduce into variable properties our boundaries, after discretization, we treat all records into one bin as a same and records from different bins as different, no matter the original distance (only very different values are expected to be in different bins).

Note that we may construct a discretization with respect to the classification (the prediction accuracy). The example of such discretization is to find a cutpoint c for a continuous variable X in order to find a binary variable (outcome is 0 for $X < c$ and 1 elsewhere). The discretization may be global

(i.e. one discretization is made and then the algorithm is executed) or local (i.e. for "divide and impera" algoritms, in every step, ad hoc discretization is made).
In spite of some minor disadvantages, discretization is widely used technique which usually improves the quality of the data analysis.

Dummy variables

The next transformation, which is often used, is the transformation of categorical variables into a set of binary variables (see Table 1.2). This step is needed when the algorithm is able to work with binary or binary and continuous variables only. Treating a categorical variable as a continuous one is an error. The error become more critical when the categorical variable is not ordinal. For example, assume the coding in Table 1.2, when treating the original variable as continuous, then we should interpret results as "red is average between blue and green". For this reason, we should avoid treating the categorical variable as a continuous one in general.

Table 1.2: Example of dummy variables

Original Variable		New Dummy Variables		
Value	Description	I_{blue}	I_{red}	I_{green}
1	Blue	1	0	0
2	Red	0	1	0
3	Green	0	0	1

1.7 Flat Table

The *Flat Table* is one of many terms used for a table which is an input for data mining algorithms. Algorithms for the data mining are designed to work with a flat table. Usually, the data have to be collected from various sources, and then transformed into this flat table. The flat table is a set of records (or observations respectively). For every record, we have many attributes which belongs to this record. The granularity of this record is determined by the business problem actually solved. When we are interested in the customer

churn, then one customer is one record and one attribute is the churn (this attribute is 1 when the customer churns and 0 otherwise). In insurance fraud, one record is one claim. For claims in the past, we have the variable *fraud* which says whether the fraud was confirmed. Note that the fraud task is biased because not all claims were investigated. In the credit scoring, one record is either one application for loan for the application credit scoring, or one customer for the behavioural credit scoring. Note that the credit scoring is also a biased analysis (many applications were rejected).
In Table 1.3, there is an example of the flat table used for the data analysis in mobile telecommunications. This table is intended for use in many CRM (customer relationship management) applications (for example the churn prediction, the propensity to buy (PTB) model for MMS – multimedia message system). The grain of the table (= one record) is one SIM card. For every SIM card, we collect many of attributes. Some sample attributes are shown in our example.

Table 1.3: The example of the Flat Table in a Mobile Telecommunication Company. The granularity of this table is a subscriber (SIM card). This table (”analytical datamart”) is designed for various CRM-based applications. Typically, this datamart contains hundreds to thousands of attributes.

Subscriber ID	Contract ID	Tariff	Spend last month	Spend Last 6M	Voice Mail	...	Churn	PTB MMS
...	...	...	...	...	...	...	...	...
123 456	155 332	Standard	$84.70	$595.42	Yes	...	0	1
123 457	155 332	Basic	$29.32	$984.16	No	...	0	0
123 458	172 315	Exclusive	$242.94	$3,514.34	Yes	...	0	0
123 459	215 794	Exclusive	$99.99	$599.94	Yes	...	0	0
123 460	156 445	Basic	$29.99	$184.46	No	...	1	0
123 461	176 743	Standard	$39.46	$246.79	Yes	...	0	1
123 462	134 491	Standard	$63.37	$197.16	Yes	...	0	0
123 463	137 673	Basic	$19.99	$119.94	Yes	...	0	0
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Chapter 2

Introduction to Decision Trees

2.1 What is a Decision Tree Technique

The decision tree induction is a widely used technique for data analysis. We can identify several reasons for this. First, decision trees can be easily visualised. Second, it is easy to understand the visualised decision tree (model). Decision trees can be the way how can the data miner communicate with business users (including managers) also with a low level of knowledge in statistics. Due to these reasons, decision trees became popular even in management. With a well-prepared data and a short training, managers can use the interactive learning of decision trees (in some software implementations) as well as browsing OLAP (Online Analytical Processing) cubes. Both techniques (OLAP browsing and decision trees interactive growing) are very easy to use for business users. The result from this managerial browsing is usually the knowledge, which can be used for his/her decisions.

Decision tree induction assumes that one variable is target (for prediction, this variable concerns the future value, which is unknown in the time of the decision) and other variables are input variables, or predictors. For known values of predictors, we are able to estimate the value of the target variable. When using decision trees, we can distinguish between two phases. In the first phase, the decision tree is learned from a historical data. Learning can be interactive or automatic. The result of the first phase is a decision tree. In the second phase, the decision tree is used. It can be used as knowledge (manager makes a decision with a knowledge of a decision tree) or as a predictive or a classification model (we estimate the future value of a target

variable or a class for known values of predictors).
Decision trees can handle both categorical and continuous variables. Availability of use of particular variable types strongly depends on the algorithm which we want to use. Most of common decision tree algorithms can now handle both categorical and continuous variable types. We have to strictly distinguish whether a target variable is a discrete or a continuous one. The type of a target variable usually influences a visualisation and the criterion which is used inside the algorithm. We will restrict a following text to a discrete (or binary respectively) target variable. The type of a source variable is handled in a much easier way. Usually, for a continuous predictor, the *cutoff* value is searched. The cutoff value is the value which is a boundary for two groups. In the first group, observations with the predictor lower than the cutoff value are included and in the second one, observations with the predictor greater than or equal to the cutoff value are included. Search for the cutoff value can be done ad-hoc for every individual group inside the greedy principle (local cutoff), or globally before the growing algorithm starts (the global cutoff value, or it can be also called a discretization).
We may assume the following situation in telecommunications. We are trying to predict who is willing to buy an unlimited GPRS data service. Following variables (attributes) were selected for this example (a real data for model learning should contain much much more variables):

Attribute Name	Description
Sp35	Spend over $35 in last month
Sex	Male/Female
GPRS6M	GPRS service was used in last 6 months
BehSegm	The segment of the customer in the corporate behavioural segmentation
PTBUnlimitedData (target)	The customer is willing to buy GPRS unlimited

In the figure 2.1, we can see a Decision Tree built on a sample of 10,000 records from a customer data file.

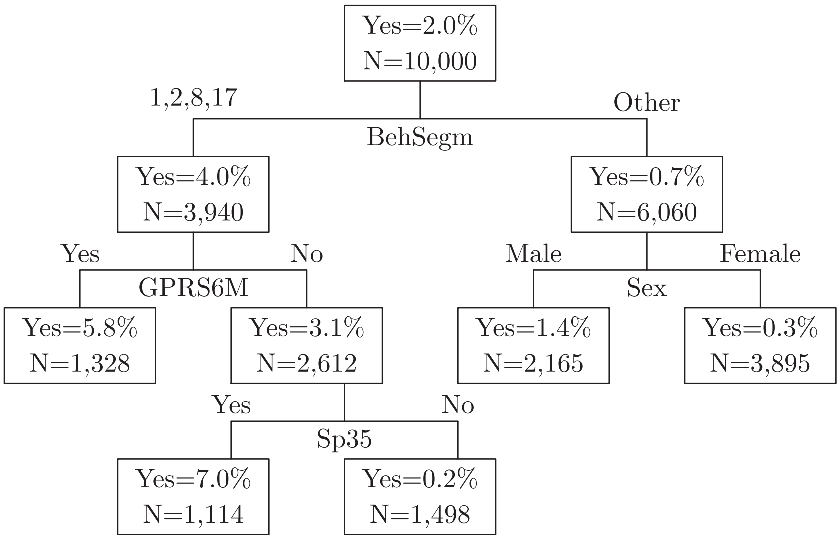


Figure 2.1: The example of a decision tree which may be used in telecommunications. Propensity to buy (PTB) model for the GPRS data unlimited service. A PTBUnlimitedData variable is a target variable.

2.2 Top Down Induction of Decision Trees

The Top Down Induction of Decision Trees (abbreviation TDIDT [26]) is a framework for growing the decision tree. It reflects the idea of learning decision trees based on the principle *Divide et Impera* ("Divide and Conquer"). The basic idea of this principle is the same at any level (at the beginning, for all records).

- For every variable A_i
    - split all given records into two or more groups (it depends on a variable type and the algorithm used), defined by values of variable A_i (e.g. group one is a group of records that satisfies $A_i = 0$ and group two are remaining records)
    - compute the measure of *goodness* for this split (how much it improves the estimation of the target variable)
- choose the variable with the highest measure of goodness and its split and split given records into groups

For every resulting group, repeat the same principle. Further splits of each resulting group do not depend on other groups and their records.
Splits are done until some stopping criterion is reached. When the stopping criterion is reached, this group will not be split anymore. This does not affect the algorithm processing in other groups. The stopping criterion may be one of the following list.

- The group is pure, i.e. all given records have only one value of the target variable.
- The group is too small, i.e. contains only a small number of records.
- Resulting groups would be too small, i.e. every reasonable split leads into groups where one of groups is very small (or one of groups contains almost all given records).
- No variable for the reasonable split is found (no variable is significant for the split).

Items two and three are *stopping rules* which are also called *pre-pruning*. It usually helps to prevent an overfitting (i.e. the classification is good at the data, from which was the decision tree trained, but is a very poor at the other set; the reason is that the method is able to distinguish the behaviour of every single record in the file, including randomness, not focusing on general properties). For some algorithms, the pre-pruning is not used (items 1 and 4 of the previous list are used always – no reasonable split means always stop growing this branch; item 1 is sometimes also treated as 'no variable for reasonable split is found'). Instead of the pre-pruning, the tree is grown to the maximum detail (usually overfitted) and the second phase, *post-pruning* (or also simply *pruning*) is applied. This step tries to combine two neighbouring leaves (groups) and using the statistical criterion which evaluates whether merging will affect a quality of the decision tree or not. This step is repeated until no two leaves can be combined without the loss of the quality.

The TDIDT framework uses a greedy principle (i.e. the splitting variable is chosen by a measure of goodness and it is never changed later). It is searching a local optimum and hoping that we find also the global maximum. In some cases, the optimal tree (the smallest tree which represents the data) can be missed. Also, in some cases, a tree growing is stopped prematurely. A typical example is a xor-based distribution. Assume following records in a flat table:

Target variable	Predictor A	Predictor B	Count of records
0	0	0	5,000
1	0	1	5,000
1	1	0	5,000
0	1	1	5,000

In the entire dataset, 20,000 records is available and a probability of target=1 is 50%. When trying to split by A or B, in both cases we would obtain two groups with 10,000 records and a probability of target=1 is 50% in both groups, so the split (either by A or by B) is evaluated as useless and then the learning is stopped. But the correct optimal tree has a division by one predictor (without loss of generality it is the Predictor A) and both groups have division by a second predictor (without loss of generality it is the Predictor B). This problem was researched in [22]. This work also introduces a new algorithm based on 'look-ahead principle' – the algorithm tries to look not only at potential groups, but also at next levels (how good would be split in next turns when in this turn A will be chosen as a splitting variable). This principle increases a complexity, but the resulting tree may be better. This principle did not become available in today's software packages which provide decision tree technique. The greedy principle is a widely accepted standard for growing decision trees. But to achieve any theoretical results, we have to know that, in some cases, a greedy learning of decision trees can stop prematurely.

2.3 Most Common Algorithms for Decision Trees

There are many algorithms for growing decision trees. Most common algorithms for decision trees will be shown here. These algorithms can be categorised into groups ([5],[8]).

CART algorithm can be categorised in a group called *Statistical Decision Trees*, or CART family

CHAID algorithm is a representative of decision tree techniques which came from *Pattern Recognition*, or the group of algorithms called also the *AID family.* Many other -AID algorithms (THAID, ...) were developed ([8]).

ID3 is the algorithm based on *the information theory, entropy*, this class is also called the *machine learning family.* Most popular software extensions include C4.5 and C5.0.

Brief examples of these algorithms will share the same example data taken from [26]. These examples assume a greedy principle and show how to calculate a measure of goodness, which is different for each algorithm shown. The example data follows.

Play tennis? dataset

Outlook	Temp.	Humidity	Wind	Play tennis?
sunny	hot	high	Weak	no
sunny	hot	high	Strong	no
overcast	hot	high	Weak	yes
rain	mild	high	Weak	yes
rain	cool	normal	Weak	yes
rain	cool	normal	Strong	no
overcast	cool	normal	Strong	yes
sunny	mild	high	Weak	no
sunny	cool	normal	Weak	yes
rain	mild	normal	Weak	yes
sunny	mild	normal	Strong	yes
overcast	mild	high	Strong	yes
overcast	hot	normal	Weak	yes
rain	mild	high	Strong	no

For all these algorithms, the Outlook variable is chosen for the root. In general, for the same data, different variables can be chosen for the split by individual algorithms.

2.3.1 CART

The CART algorithm was developed by Breiman et al [2] in 1984. The CART is an abbreviation which stands for *Classification And Regression Trees.* These two types of trees have one difference – classification trees assume a discrete target variable. Conversely, regression trees assume a continuous target variable. We will focus on classification trees only.

When a split variable is being chosen, all possible variables are tested for a criterion and the best one is selected.

CART trees assume binary splits only. When the predictor is

binary variable, each value defines one group for a split,

discrete variable, all possible combinations of groups are assumed (for n value discrete variable, $(2^n - 2)/2$ subsets are tested),

continuous variable, all possible cut points are tested (for n records, at most $n - 1$ cut points are tested).

CART uses as a split selection criterion 'weighted average impurity', which is based on GINI index. So at first, GINI indices are computed and then combined into the criterion. We show here computation of the GINI index for two groups from a possible binary split only (we assume binary variable, otherwise see predictor types and handling with them which is a few lines above). First, for each group (a possible node), the GINI index defined as $1 - \sum_i p_i^2$, where p_i is a proportion of a target category (e.g. i=$\{$Yes,No$\}$), is computed. Next, the GINI index for a variable splitting is computed as a weighted average of GINI indices for each group. Note that the best value for splitting is the minimal value of GINI index. So we may say that the measure of goodness is a negative value of the GINI index.

The example of the GINI index computation for Play tennis? data is shown in following tables. First, we decide how to split three possible values of *Outlook* into subgroups.

	Grouping 1		Grouping 2		Grouping 3	
Play tennis?	Sunny + Rain	Overcast	Sunny + Overcast	Rain	Sunny	Rain + Overcast
Yes No	5 5	4 0	6 3	3 2	2 3	7 2
GINI for group	0.5	0	0.44	0.48	0.48	0.35
GINI for variable	0.36		0.46		0.39	

The best division into subsets is the Grouping 1. A similar test can be computed for the *Temperature*. The result is that the best division into subsets is *Hot + Cool* in one group and *Mild* in the other group.
Now, we will compute GINI indices for all variables (and a best division into subsets). Results are shown in a following table.

	Outlook		Temperature		Humidity		Wind	
Play tennis?	Sunny + Rain	Overcast	Hot + Cool	Mild	High	Normal	Weak	Strong
Yes	5	4	5	4	3	6	6	3
No	5	0	3	2	4	1	2	3
GINI for group	0.5	0	0.47	0.44	0.49	0.25	0.38	0.5
GINI for variable	0.36		0.46		0.37		0.43	

The best value of the GINI index has the *Outlook* variable, so it will appear in a split on the root level. Next, the group defined as *Outlook=Sunny or Rain* with 10 records will be split using the same algorithm, and the second group defined as *Outlook=Overcast* will be also split using the same algorithm until the stopping criterion 'no variable for reasonable split is found' is reached. The CART algorithm usually does not use a pre-pruning, it uses a post-pruning.

2.3.2 CHAID

The CHAID abbreviation stands for the *Chi-square Automatic Interaction Detection*. This algorithm ([17]) can handle nominal, ordinal and continuous variables. We will restrict here to a binary target and nominal predictors. The CHAID uses the TDIDT schema (framework), the split is chosen by the best value of a measure of goodness for each variable. This measure is based on the statistical test, the p-value (significance level) is computed for the result. This measure is calculated for every variable as follows.

- When a nominal variable has more than two values, try to combine some values.
  - Find the pair of categories, which is least significant (i.e. largest p-value) with respect to the target variable.
  - Repeat the previous step until no pair is statistically insignificant to combine (until $p < \alpha_{combine}$ for all pairs).
- Compute a chi-square statistics and a p-value for this variable vs. target.

- Adjust the resulting p-value with the Bonferroni's adjustment.

The p-value criterion may prefer more categories. So it is used the Bonferroni's adjustment. This is a correction for a p-value. For a binary target and a k-ary predictor, the adjusted p-value (denoted as α in expressions) is computed as $\alpha_{adj} = \alpha \cdot \frac{k(k-1)}{2}$.
The chi-square statistics is computed by a general schema for independence of two categorical variables

$$\chi^2 = \sum \frac{(observed - expected)^2}{expected}.$$

For the fourfold table with variables $A = \{a_0, a_1\}$ and $B = \{b_0, b_1\}$, the expected probability of $A = a_i, B = b_j$ is given by $P(A = a_i) \cdot P(B = b_j)$ (note that $P(A = a_i)$ and $P(B = b_j)$ are computed as marginal relative frequencies from the fourfold table).
Resulting χ^2 is then converted to the p-value and it is adjusted. For this example, we will compute the p-value for the attribute Humidity. First, we will look at the data

	Humidity		
Play tennis?	High	Normal	Any
Yes	3	6	9
No	4	1	5
	7	7	14

Now, we will compute

$$\chi^2 = \frac{(3 - \frac{9 \cdot 7}{14})^2}{\frac{9 \cdot 7}{14}} + \frac{(6 - \frac{9 \cdot 7}{14})^2}{\frac{9 \cdot 7}{14}} + \frac{(4 - \frac{5 \cdot 7}{14})^2}{\frac{5 \cdot 7}{14}} + \frac{(1 - \frac{5 \cdot 7}{14})^2}{\frac{5 \cdot 7}{14}} =$$

$$= \frac{(-1.5)^2}{4.5} + \frac{1.5^2}{4.5} + \frac{1.5^2}{2.5} + \frac{(-1.5)^2}{2.5} = 1 + \frac{4.5}{2.5} = 2.8.$$

From $\chi^2 = 2.8$, we may compute
the p-value $\alpha = 0.0943$. The Bonferroni adjustment does not change the p-value for this case (we have 2 values of a predictor only). We may continue to compute the p-value for other variables. Note that for 3-ary variables, we should apply "merging categories".
CHAID trees usually use pre-pruning, also called as *stopping criteria*. That means, that the growing is stopped when given criteria is met (at least one criterion is met). These criteria usually include

- the node is pure (i.e. only one target category is in a node),
- the minimal number of records in the node to allow next splitting,
- the minimal number of records in newly created leaves (this disqualify some potential splits) and
- the maximum tree depth is reached.

This algorithm has also an alternative approach called the *Exhaustive CHAID*. In some cases, the CHAID may not find the optimal category merging (it may stop merging of categories prematurely). So the Exhaustive CHAID algorithm continues until 2 categories are left. Then, it looks at all steps of merging and chooses the best grouping, based on adjusted p-values.

2.3.3 ID3

The algorithm ID3 ([26]) is an entropy-based algorithm. This algorithm uses also the TDIDT schema (framework), the split criterion is designed to prefer an increase in the purity of leaves. This criterion is based on the information theory and the maximum information gain is preferred. The information gain for a variable A is defined as

$$Gain(A) = I(p) - E(A),$$

where $E(A) = \sum_i \frac{n_i}{n} I(p_i)$ and $I(p) = -p \log p - (1-p)\log(1-p)$. Note that the logarithm is the base 2 logarithm.
Now, we can calculate the information gain for our example "Play tennis". We will try to compute the information gain for a variable *Humidity.* First, we will look at a following table, which will help us.

	Humidity		
Play tennis?	High	Normal	Any
Yes	3	6	9
No	4	1	5

Information Gain for Humidity is

$$Gain(Humidity) = I(\frac{9}{14}) - E(Humidity) = I(\frac{9}{14}) - (\frac{7}{14} \cdot I(\frac{3}{7}) + \frac{7}{14} \cdot I(\frac{6}{7})) =$$

$$= I(\frac{9}{14}) - \frac{1}{2} \cdot I(\frac{3}{7}) - \frac{1}{2} \cdot I(\frac{6}{7}) = 0.940 - \frac{1}{2} \cdot 0.985 - \frac{1}{2} \cdot 0.592 = 0.152.$$

In similar way we may compute the information gain for all attributes. The results are shown in a following table.

Attribute	Information Gain
Humidity	0.152
Wind	0.048
Outlook	0.25
Temperature	0.03

We will choose the attribute with the highest information gain in the split, so Outlook will be selected.

The algorithm continues in a similar way until pure leaves are achieved. To prevent overfitting, post-pruning methods are used.

2.3.4 C4.5

The C4.5 algorithm [25] is a software extension of the ID3 algorithm. This algorithm was developed by Quinlan. The most current version of this software extension is C5.0, which has many advantages in comparison with C4.5, but it is commercial software so the availability and possibility to use is very reduced. Advantages are in details (minor details in criteria, for example an adjustment for continuous variables), but the principle remains the same.

C4.5 can use one of two criteria. The first one is the information gain shown in previous section. The second one is also an entropy-based criterion, which is based on the information gain. It was observed that information gain itself prefers attributes with a very high cardinality (with a very high number of possible values). The best variable for split is a variable, which is unique for each record, so only one record would appear in every category (leaf). So C4.5 calculates also the potential information from every partition (maximum possible information) and compares it with the actual information. The potential split information (SI) is defined as follows

$$SI(A) = -\sum_i \frac{n_i}{n} \log(\frac{n_i}{n}).$$

The gain ratio G is then defined as

$$GR(A) = Gain(A)/SI(A).$$

So the gain ratio is calculated for every variable and the variable with the highest gain ratio is selected fo a split. We may calculate $SI(A)$ for each variable. The only inputs for $SI(A)$ are group sizes in division by variable A. We will show the example of SI for the variable wind. We have two classes of the variable wind – weak and strong. The number of records for weak wind is 8, the number of records for a strong wind is 6. So

$$SI(wind) = -\frac{8}{14}\log(\frac{8}{14}) - \frac{6}{14}\log(\frac{6}{14}) = 0.985$$

We may compute the Split information for other variables in a similar way. The following table shows the Gain, the Split information and the Gain Ratio for every variable. Note that the Gain is taken from the previous section.

Variable	Class sizes	Gain	Split information	Gain ratio
Humidity	7,7	0.152	1	0.152
Wind	8,6	0.048	0.985	0.049
Outlook	5,4,5	0.25	1.577	0.156
Temperature	4,6,4	0.03	1.362	0.021

So the variable Outlook is also selected. In general, the variable for the split may be different for the Gain criterion (for the ID3) and for the Gain ratio criterion (C4.5).

Chapter 3

Optimal Tree – Problem Definition

3.1 Problem Definition

Algorihm based on decision tree induction use in the growing phase a greedy principle. This is a very good method when we know that to find a decision tree which represents the distribution is a NP-complete task ([10],[11]). When learning decision trees, the focus is usually on the prediction accuracy. The pruning phase of decision tree induction algorithms is used to prevent overfitting.
Our approach in this thesis requires the best possible prediction accuracy, but also the smallest possible tree which achieves this best accuracy. We may find groups of decision trees that every tree within every group gives the same information. That means that every decision tree from this group gives the same classification on every input dataset. We will call these trees classificationally equivalent. We will try to improve the post-pruning phase to allow more prunes in order to get the smaller tree. Next, we will examine our improvement to post-pruning whether it is possible to find the optimal tree for a distribution – the smallest tree which is classificationally equivalent to any tree which represents the given distribution.

That means we will not give up the prediction accuracy requirement, we will only add the simplicity of the tree as a secondary requirement. Let us trust that in many cases decision trees resulting from the greedy algorithm (with post-pruning respectively) can be optimised. It will also help to understand the true structure of the data by users (it helps to a business understanding in a language of the CRISP-DM). Moreover, making the tree simpler reduces overfitting.

Assume the following situation. We will try to predict a success at the exam for pupils. For boys, a success depends on knowledge and for girls, a success depends on a diligence. Moreover, the knowledge and the diligence are strongly correlated. The decision tree for this distribution is shown in Figure 3.1 on the left.

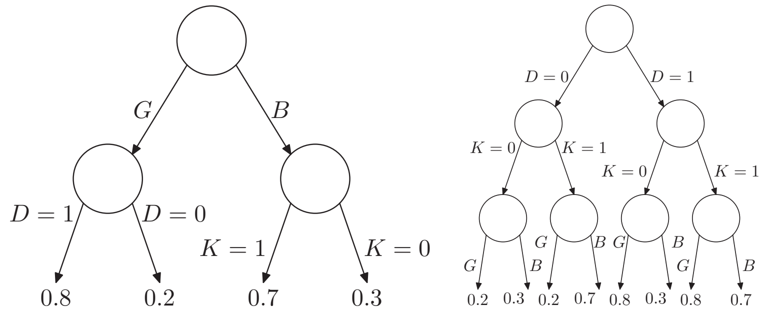


Figure 3.1: Two decision trees with the same classification on every data set and a different number of leaves. The target variable is a pass/fail mark at the exam (pass=1, fail=0). A probability is a proportion of the pass mark. D=Very Diligent, K=High Knowledge, B/G=Boy/Girl

Our original situation can be described by the optimal tree with 4 leaves (in Figure 3.1 on the left), the greedy algorithm chooses the diligence for the root, where the boy/girl split is in a deeper level and pruning is not able to propagate it to the root.

The basic idea of the proposed algorithm is that it will transform the tree grown by some standard greedy algorithm to the smaller one (ideally to the optimal one). The complexity of the proposed algorithm should not be exponential in the number of leaves of the tree resulting from the greedy phase (this occurs when a trivial exhaustive search is used).

3.2 Framework of this thesis

This research was inspired by results achieved in bayesian networks few years ago ([13],[14]). These works have restricted the set of available distributions and proved theorems about finding optimal bayesian networks. Our work should be similar in this way (it will use the same idea for decision trees) – to restrict reasonably the set of distribution and to provide an algorithm which finds the optimal decision tree for any distribution from our restricted set (and finding the optimal tree should be proved theoretically for a distribution).

Our goal was to find a smaller tree using some improvements into post-pruning phase and to examine whether it can find an optimal tree for a distribution under some reasonable assumptions. This means to find the combination of a restriction/set of operations such that applying this restriction (some restriction is needed due to the NP-completeness of the original problem), there exists a sequence of operations which converts a tree from the greedy phase to the optimal one.

Next task was to find an algorithm which finds this sequence for a distribution in the time which is better than exponential (in the number of leaves of the tree which is the result of the growing phase). At the beginning, it was not sure whether original restrictions and the set of operations will be sufficient for the (constructive) algorithm, but it is shown that proposed assumptions are sufficient.

Author's original work (chapters 4 and 5) include

- introducing some improvements into the post-pruning phase to allow more prunes,
- examining whether this algorithm can find the optimal tree for a distribution (and under which assumptions), this include
  - providing theorems on finding decision trees (which can help to develop an algorithm) and proving them,
  - providing an algorithm to find the optimal decision tree and proving its properties for a distribution,
- testing the proposed algorithm on both simulated and real data and comparing the results with the standard decision tree algorithm (the CART algorithm).

To achieve any reasonable theoretical results, due to NP completeness of the problem solved in general, there was defined a restriction for the set of available distributions, which have helped to develop a polynomial algorithm. This work is restricted to binary variables only. Other variables can be easily transformed to binary ones. When we have a categorical predictor with more than two values, (e.g. *Colour*={*Blue, Yellow, Green*}), we may convert it into several binary attributes (i.e. $I_{Blue}, I_{Yellow}, I_{Green})$). This approach was described in the subsection 1.6.2. For a continuous variable X, we may find the cutpoint c and define one binary variable $I_X = 1$ if $X < c$ and $I_X = 0$ otherwise.

Our approach will be a little bit different in comparison with standard decision tree algorithms. We implicitly assume that leaves might be impure. So we expect that for a given set of predictors and their values, both values of a target variable are allowed, with the given proportion (of probability that a target variable has a value "1"). When describing some non-deterministic behaviour, like customer churn, propensity to buy, but also fraud etc., we may have different values of the target variable for the same set of predictors. The reason is that we will never have all attributes which influences the behaviour which we predict. For example, customer feelings, behavioural standards and many attributes that we will never have available may influence churn, fraud etc.

We will say that the decision tree represents a distribution (even for the distribution), we do not require the data to represent strict fixed rules (like "for 100% of cases from this group, the target variable is 1"). It will be very helpful for both theoretical and practical results.

Now, we look at our approach in a more detail. We will do several things concurrently – we will introduce some operations into the post-pruning phase, we will try to examine their properties and "common power" and we will try to prove theorems for a distribution whether the optimal tree can be found using these operations.

There was tested several post-pruning operations and their combinations. As a result, it was shown that (under some restrictions) the only one operation is sufficient. For these purposes, some restrictions of the set of distributions were made. This allowed us to formulate and to prove theorems, to establish the algorithm and to prove its properties. The last phase, testing the algorithm on the simulated data will be proceeded on all distributions (not only distributions from our restricted set).

Our algorithm is only a modification of current algorithm (the CART algorithm is used, but also the other greedy algorithm may be used). The growing phase remains the same, the only difference is in the post-pruning phase, to where we introduce only one more tree operation. This operation locally rearranges the tree. Our approach will be following. We will try to prune leaves with the same probability of a target variable and which can be neighbours after some changes in the inner-structure of the tree. Multiple applications of our new operation could get these two leaves to be neighbours and able to be pruned.
Now we look at the approach from a theoretical point of view, which will be proved for the distribution. The first phase will introduce some terms used in the following text. Then we show a theoretical theorem that the optimal tree can be found by applying two operations – the prune operation and our new operation called the *parent-child exchange* (PCE). This new operation only locally rearranges the inner structure of the tree, remaining leaves' definition unchanged. First theoretical theorems do not show the way to find the optimal tree, only that there exists a sequence of two types of operations that can find it. Nevertheless, this theorem has a very important corollary – after applying the prune operation (which cannot be undone in our approach), we are still able to get to the optimal tree by another sequence of two types of operations. So, we may prune anytime we want. In addition, because we sometimes need to make large rearrangements of the tree before pruning, we will design our algorithm to prune always when it is possible. The next part of this thesis will introduce the algorithm and proves that this algorithm finds the optimal tree. The complexity of this algorithm is polynomial in the number of leaves of the full tree (or, of the tree resulting from the greedy phase respectively). The exhaustive search for the optimal tree is exponential in the number of leaves of the same tree.

Note that theoretical results will be provided for a distribution. The algorithm will be designed for both the distribution and the dataset, but the proof will be for a distribution only. The only difference is that for a distribution, we say that two leaves can be pruned, if the proportions of the target variable are the same in these two leaves. For a dataset, a statistical test is used – the tested hypothesis is: these two proportions are equal.

3.3 Results on the Tree Size and Optimality

There are many works related to decision trees, their growing, a complexity, and a split selection, among others. Most of them were focused to improve current algorithms on any distribution.
Most of decision tree algorithms use top down induction of decision trees and the greedy principle on one attribute (attributes are treated as standalone, one best attribute is chosen for a split). Some works are improving the growing phase of the algorithm to find a tree which represents the data for more distributions. The greedy principle is a very good method, which is commonly accepted. It is true that there exists a class of distributions, for which the greedy principle does not find the tree which represents this distribution. There were proposed many algorithms which use lookaheads to prevent to stop growing the tree prematurely ([22],[28]).

There was also a research which was focused on using the other criteria (not only the prediction accuracy) when growing a tree. The example is the algorithm DMTI ([29]), which may use also the tree size as one of criteria used when growing. This approach focus on both accuracy and tree size in a growing phase and allows a restructuring of a decision tree as well. The algorithm presented in [7] takes in question the estimated subtree size when the splitting variable is chosen. An interesting idea is used in [6]. That work improves a time-consuming algorithm for decision trees in the following way: algorithm is anytime interruptible and it returns a reasonable result at any moment.
Another work which concerns the accuracy and the tree size together is [24]. It proposes a method which uses pre-pruning based on a cross validation, so growing stops when it seems that no further split is needed. This leads into a non-overfitted tree with comparable prediction accuracy.

We know that no algorithm can give the optimal solution without the reduction of the set of distributions (under assumption $P \neq NP$). It can be proven that for a given data set, the question whether there exists a decision tree with at most k nodes is NP-complete ([10],[11]). Therefore, we will stay at the most common greedy algorithm and we will reduce the set of distributions – this can be a way to get out of the trap. We will focus primary on the accuracy (we do not allow to reduce the accuracy) and we will focus on the tree size as a secondary criterion (we are trying to reduce a tree size without reducing the prediction accuracy, or more general, in theoretical case without changing the classification).

In this thesis, we define tree optimality as a number of leaves of the decision tree. There are used also some other criteria for the optimality, for example the number of nodes. For binary attributes, to that is this work restricted to, the criterion the number of leaves and the criterion the number of nodes are the same. The next commonly used criterion is the average tree depth. This criterion is used when we have to deal with the CPU time. In the most of data mining applications, the CPU time is not a problem. For several critical applications, as online credit card misuse detection, cluster of computers may be used. Therefore, we will focus on the tree size only in our thesis.
The last note is about decision tables and decision rules. There are many algorithms to find decision tables. Note that decision tables are *not* the same as decision trees. It can be proved that the conversion of the smallest decision tree from a decision table is NP-complete ([11]). Therefore, algorithms for decision tables will not help us.

Chapter 4

Finding Optimal Trees

This chapter is the core chapter of this thesis. This chapter is divided into 3 sections.

In Section 4.1, a specific terminology will be defined. We will provide here intuitive definitions, strict formal definitions and examples to illustrate the meaning of these definitions. We will show also some supplementary theorems, which may help to explain terms and their properties.

Section 4.2 provides some theorems on finding optimal decision trees. The main theorems were the essentials for creating the framework of the algorithm. Some of these theorems will be used in the proof that the algorithm finds the optimal decision tree for particular distributions (called strong faithful distributions – see later). The core of this section is to prove the *existence* of a sequence of steps to get the optimal decision tree from a given tree which represents a given distribution (it assumes that we have the optimal tree, the given tree which represents the given distribution and we will show that there exists a sequence of operations which converts a given tree to the optimal one). Construction of this sequence is based on knowledge both given and optimal tree. There is one more result in this section – it will be proven that for strong faithful distributions that we can design our algorithm to prune anytime – this will not mislead us so we will be able to get to the optimal tree from a tree with prune operation applied.

The algorithm which finds the optimal decision tree from the distribution is provided in Section 4.3. This algorithm is designed to work on any data and allows more prunes, but only for strong faithful distributions is guaranteed that the optimal tree will be found. This section also provides a proof that the provided algorithm really finds the optimal tree under given assumptions.

4.1 Definitions

This section includes notation, formal definitions and theorems. We will use following notation for variables:

- A_0 for a target variable and
- $A_1, A_2, \ldots, A_n$ for input variables, also called predictors.

We will assume that these (binary) variables have values 0 and 1.

4.1.1 Probability Distribution

The first notion we will define is the *(joint) probability distribution.* The distribution assigns a probability for each possible combination of values of attributes (variables).

Definition 1 (Joint Probability Distribution) *Let $A_0, A_1, A_2, \ldots, A_n$ be a set of binary variables. Then the* distribution *(or the* joint probability distribution*), denoted by $P(A_0, A_1, A_2, \ldots, A_n)$, is a probability function assigning to each combination of values $(a_0, a_1, \ldots, a_n) \in \{0,1\}^{n+1}$ a non-negative number, i.e. $P(A_0 = a_0, A_1 = a_1, \ldots, A_n = a_n) \geq 0$ in such way that the sum of these assigned values is 1, i.e. $\sum\limits_{a_0=0}^{1} \sum\limits_{a_1=0}^{1} \ldots \sum\limits_{a_n=0}^{1} P(A_0 = a_0, A_1 = a_1, \ldots, A_n = a_n) = 1$.*

Remark 1 *We will use some more notation. The marginal distribution $P(A_0 = a_0, A_1 = a_1, \ldots, A_{i-1} = a_{i-1}, A_{i+1} = a_{i+1}, \ldots, A_n = a_n)$ will be abbreviated expression for $\sum\limits_{a_i=0}^{1} P(A_0 = a_0, A_1 = a_1, \ldots, A_{i-1} = a_{i-1}, A_i = a_i, A_{i+1} = a_{i+1}, \ldots, A_n = a_n)$. We may apply this sum several times to get the marginal distribution of any dimension (lower than n).*

In this text, we will assume only distributions, which holds

$$P(A_1 = a_1, A_2 = a_2, \ldots, A_n = a_n) > 0$$

$\forall a_1, a_2, \ldots, a_n$, i.e. every possible combination of input variables has nonzero probability (note that A_0 is a target variable). These distributions will be called *full distributions.*

Considering full distribution allows us to easily define some other notions. Note that results of this work can be easily applied to all distributions, it is only a large technical work (to handle special situations) in definitions and theorems. This extension is not presented here.
The example of the probability distribution is shown in Table 4.1.

Table 4.1: Probability distribution P_1. This distribution will be used in many examples later.

A_0	A_1	A_2	P_1
0	0	0	0.08
0	0	1	0.16
0	1	0	0.20
0	1	1	0.03
1	0	0	0.09
1	0	1	0.18
1	1	0	0.20
1	1	1	0.06

Let us define one more notion – the probability measure, or simply the probability. This notion is a generalization of the distribution to the all subsets of the $\{0,1\}^{n+1}$. Therefore, we will denote this generalization by the same symbol P.

Definition 2 (Probability Measure) *Let $P(A_0, A_1, \ldots, A_n)$ be a probability distribution. Then* a probability measure P *is defined for every set $S \subseteq \{0,1\}^{n+1}$ by the expression*

$$P(S) = \sum_{(a_0,a_1,\ldots,a_n)\in S} P(A_0 = a_0, A_1 = a_1, \ldots, A_n = a_n).$$

Next, we will define the conditional probability distribution. In this text, we will pay special interest to conditional probability distributions. The conditional probability distribution is the probability of the target variable when we know values of all predictors (for every combination of values of predictors, probability of the target is assigned).

Definition 3 (Conditional Probability Distribution)
Let $A_0, A_1, A_2, \dots, A_n$ be a set of binary variables. Let A_0 be a target variable. Then, the conditional probability is the function

$$P(A_0|A_1, A_2, \dots, A_n) = \frac{P(A_0, A_1, \dots, A_n)}{P(A_1, A_2, \dots, A_n)}.$$

Notice that conditional probability distribution is also a function of $n + 1$ variables assigning to each combination of values $(a_0, a_1, \dots, a_n)$ a non-negative value from $[0, 1]$. In addition, $P(A_0 = 1|A_1 = a_1, A_2 = a_2, \dots, A_n = a_n) + P(A_0 = 0|A_1 = a_1, A_2 = a_2, \dots, A_n = a_n) = 1$ and therefore it is sufficient to know the (conditional) probability only for one outcome (e.g. $P(A_0 = 1|A_1 = a_1, A_2 = a_2, \dots, A_n = a_n)$).

In a similar way we define a conditional probability distribution for given variables $A_{i_1}, A_{i_2}, \dots, A_{i_k}$, where $\{i_1, i_2, \dots, i_k\} \subseteq \{1, 2, \dots, n\}$, i.e.

$$P(A_0|A_{i_1}, A_{i_2}, \dots, A_{i_k}) = \frac{P(A_0, A_{i_1}, A_{i_2}, \dots, A_{i_k})}{P(A_{i_1}, A_{i_2}, \dots, A_{i_k})}.$$

Given a probability distribution $P(A_0, A_1, \dots, A_n)$ and an arbitrary set $L \subset \{1, \dots, n\}$ one can always calculate the conditional probability distribution $P(A_0|A_i, i \in L)$ and this conditional probability distribution is unique because we consider only full distributions.

The example of the conditional probability distribution is in Table 4.2. This conditional probability distribution is derived (calculated) from the joint probability distribution P_1 shown in Table 4.1.

For a conditional probability distribution $P(A_0|A_i, i \in L')$ there exists an infinite number of distributions $P(A_0, A_1, \dots, A_n)$, from which we can derive $P(A_0|A_i, i \in L')$. The only difference in these joint probability distributions is in $P(A_i, i \in L')$. We may see such an example in Table 4.3.

In this example, there are two differences in $P(A_1, A_2)$ and $P'(A_1, A_2)$. The first difference is that $P(A_1 = 1, A_2 = 0)$ was decreased form original $0.20 + 0.20 = 0.40$ to new $0.11 + 0.11 = 0.22$. The second difference is an increase in $P(A_1 = 1, A_2 = 1)$, which was changed from the original size $0.03 + 0.06 = 0.09$ to the new size $0.09 + 0.18 = 0.27$. Note that *all* conditional probabilities $P(A_0|A_1, A_2)$ remain the same.

Table 4.2: The conditional probability distribution $P(A_0 = 1|A_1, A_2)$ for the distribution P_1. There are three more tables here – tables with $P(A_0 = 1|A_2)$, $P(A_0 = 1|A_1)$ and $P(A_0 = 1)$. These tables are pre-calculated for the easy reference in following examples.

A_1	A_2	$P(A_0 = 1\|A_1, A_2)$
0	0	9/17
0	1	9/17
1	0	1/2
1	1	2/3

A_1		$P(A_0 = 1\|A_1)$
0		9/17
1		26/49

	A_2	$P(A_0 = 1\|A_2)$
	0	29/57
	1	24/43

		$P(A_0 = 1)$
		53/100

Table 4.3: Probability distributions P_1 and P'_1, which both define the same conditional probability distribution.

A_0	A_1	A_2	P_1
0	0	0	0.08
0	0	1	0.16
0	1	0	0.20
0	1	1	0.03
1	0	0	0.09
1	0	1	0.18
1	1	0	0.20
1	1	1	0.06

A_0	A_1	A_2	P'_1
0	0	0	0.08
0	0	1	0.16
0	1	0	0.11
0	1	1	0.09
1	0	0	0.09
1	0	1	0.18
1	1	0	0.11
1	1	1	0.18

4.1.2 Decision Tree

Next, we will define the binary tree using notions of the graph theory (see for example [20]).

Definition 4 (Binary Tree) *Let T=(V,E) be a graph with the following properties:*

1. $|E| = |V| - 1$,
2. *T is connected, i.e. for every pair of vertices $v_1, v_2 \in V$, there exists a path from v_1 to v_2,*
3. *one vertex $v_r \in V$ is called a root; length of a path from the root v_r to v is called a depth of v. Children of $v \in V$ are those $u \in V$, which are adjacent to v and their depth is by 1 larger than depth of v,*
4. *every $v \in V$ has zero or two children.*

Then, T is called a Binary Tree*. Vertices with no children are called leaves. For easy reference, we will split the set of vertices V into two disjoint subsets $V = W \cup U$, where W is the set of all leaves and U is the set of internal nodes, or a set of non-leaf vertices.*

The example of a binary tree is shown in Figure 4.1.
In the following text, the root vertex will be drawn as the top vertex of the tree.
Now, we introduce two functions – first for vertices (it assigns a split variable to every internal node) and second for edges (it assigns a value of the respective split variable to every edge).
In this definition, the following notation will be used:

- Function m assigns a split variable A_i to every internal node v.
- Function s is defined for every edge. Having the vertex v with the split variable A_i and children v_1 and v_2, function s defines which child is defined by the condition $A_i = 0$ and which one by the condition $A_i = 1$ (i.e. formally $s(v, v_1) = 0$ and $s(v, v_2) = 1$).

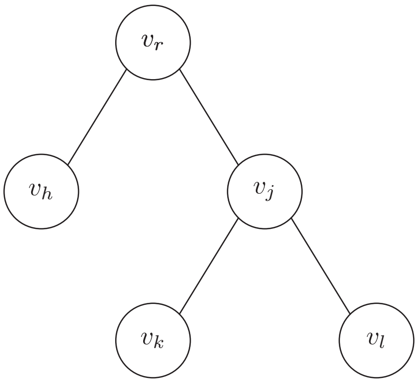


Figure 4.1: Example of a Binary Tree

Definition 5 (Binary Decision Tree) *Let $T = (V, E)$ be a binary tree. Let $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ be a set of binary variables. Let $m : U \rightarrow \mathcal{A}$ be a mapping such that on every path P from the root v_r to the leaf w, every variable $A \in \mathcal{A}$ can be assigned to none or one vertex (node). Let $s : E \rightarrow \{0, 1\}$ be a mapping such that $\forall v \in U$ with children v_1 and v_2 there is $s(v, v_1) + s(v, v_2) = 1$.*

Then, $\mathbf{T} = (T, \mathcal{A}, m, s)$ *is a* binary decision tree*.*

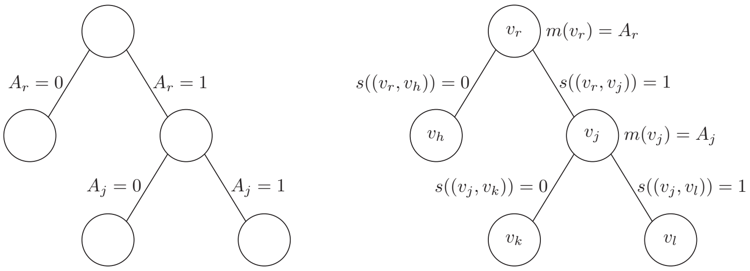


Figure 4.2: The example of the Binary Decision Tree

The example of a decision tree is shown in Figure 4.2. In this Figure, there are two visualizations of the same tree shown. The right tree corresponds to a definition of a binary decision tree, the left one is a simplified visualisation. In our example, the root node is split into two children by the condition $A_r = 0$ (the left child), or $A_r = 1$ respectively (the right child). The right child of the root is the node, which is split again, now by the variable A_j.

Remark 2 *In the following trees, for an internal vertex v with split variable A_i and children v_1 and v_2, the child with the value of the split variable $A_i = 0$ will be situated in the left, if not explicitly expressed otherwise.*

There is one more point to emphasize. The definition of a decision tree also requires that we will split by any variable at most one time on the any path from the root to a leaf. This is without loss of generality, because the second split by the same (binary) variable on the same path would lead into the situation, that all observations would be in one child and second one would be empty.

Now, we will tie the definition of the decision tree with the (conditional) probability distribution. For a given distribution $P(A_0, A_1, \ldots, A_n)$ and a given decision tree $\mathbf{T}$, we can derive the *tree CPD P_T* as follows: for each combination of values $(a_1, a_2, \ldots, a_n)$, we find a leaf to which "this combination belongs to" and we denote the condition in this leaf as C_v. Then, we define $P_T(A_0 = 1|A_1 = a_1, A_2 = a_2, \ldots, A_n = a_n)$ as $P(A_0 = 1|C_v)$.

For a tuple of a given probability distribution P and a given tree T, we have defined a tree CPD P_T. Note that for a given probability distribution P and different trees $\mathbf{T}_1$ and $\mathbf{T}_2$, we may get different tree CPDs P_{T_1} and P_{T_2}.

We will define one more notion here. For a given probability distribution $P(A_0, A_1, \ldots, A_n)$ and a tree T we say that *the tree T represents the distribution $P(A_0, A_1, \ldots, A_n)$*, or *the tree T represents the CPD $P(A_0|A_1, A_2, \ldots, A_n)$* if

$$P_T(A_0|A_1, A_2, \ldots, A_n) = P(A_0|A_1, A_2, \ldots, A_n).$$

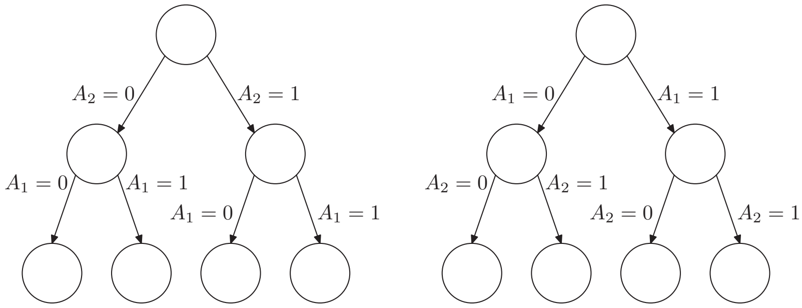
Next, we introduce a *full tree* as the tree with all leaves in depth n. The full tree is a tree with a maximum size. Note that the full tree represents any distribution.

4.1.3 Equivalence of Decision Trees

In this section, we will introduce two equivalences on decision trees. These equivalences will be used later.

Definition 6 (Equivalent Decision Trees) *Let $\mathbf{T}_1, \mathbf{T}_2$ be decision trees with the same number of leaves. Let $w_1, w_2, \ldots, w_k$ be all leaves of $\mathbf{T}_1$ and $x_1, x_2, \ldots, x_k$ be all leaves of $\mathbf{T}_2$. Let $C_1, C_2, \ldots, C_k$ be conditions assigned to leaves $w_1, w_2, \ldots, w_k$ and let $D_1, D_2, \ldots, D_k$ be conditions assigned to leaves $x_1, x_2, \ldots, x_k$. Let $M_1 = \{C_1, C_2, \ldots, C_k\}$ and let $M_2 = \{D_1, D_2, \ldots, D_k\}$. Then $\mathbf{T}_1$ and $\mathbf{T}_2$ are equivalent binary trees if $M_1 = M_2$.*

We have defined equivalence on decision trees. Two different trees may belong to one equivalence class, if their set of conditions assigned to leaves are identical. The example of two such trees is shown in Figure 4.3.



Set of conditions (four items, shown one per row)
$A_1 = 0,\ A_2 = 0$
$A_1 = 0,\ A_2 = 1$
$A_1 = 1,\ A_2 = 0$
$A_1 = 1,\ A_2 = 1$

Figure 4.3: Two decision trees which belong to one equivalence class and their set of conditions assigned to leaves (identical for both trees).

Remark 3 *Note that*

1. *all trees from one equivalence class define the identical tree CPD P_T for every probability distribution, but*

2. *two trees from different equivalence classes may also define the identical tree CPD for a given probability distribution P.*

Remark 4 *All full trees are equivalent, i.e. set of all full trees corresponds to one equivalence class.*

We have defined equivalence on decision trees. Now, we are going to introduce one more equivalence, which considers also the probability distribution.

Definition 7 (CPD Equivalence) *Let P be a probability distribution. Then decision trees T_1 and T_2 are* CPD equivalent with respect to P, *if their respective tree CPDs are identical, i.e. $P_{T_1} = P_{T_2}$.*

Note that for a given distribution P, when we use the abbreviated notion *CPD equivalent trees*, we mean *CPD equivalent trees with respect to P*.
In Figure 4.4 we may see two trees, which are not equivalent, but which are CPD equivalent. Due to the previous remark, every class of CPD equivalence consists of one or several classes of decision tree equivalence.

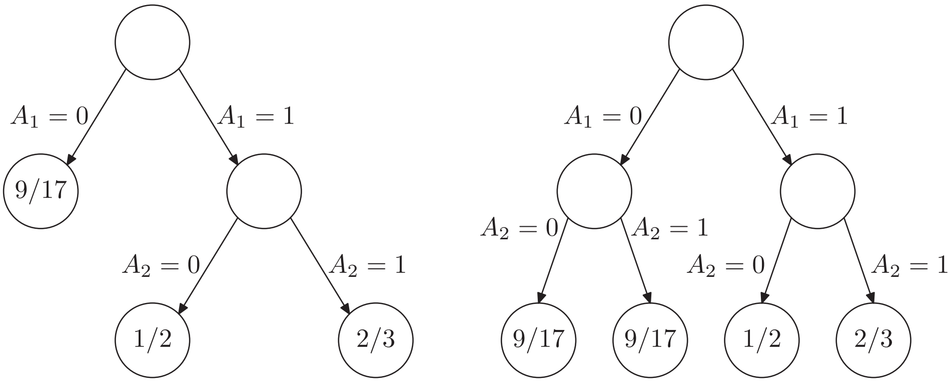


Figure 4.4: Two trees, which are not equivalent (definitions of leaves are different), but which are CPD equivalent (tree CPDs are identical) with respect to distribution P_1 (see table 4.2 on page 50).

Let us emphasise that CPD equivalence is defined for a given distribution. When we use another distribution, we may obtain completely different CPD equivalence classes.

Remark 5 *Let T and T' be two equivalent trees. Then they are CPD equivalent with respect to any distribution P.*

4.1.4 Optimality of Decision Trees

In some situations, two or more decision trees may represent the same distribution. Which one will be the best for us? We will require the simplest one. The simplicity will be expressed as the number of leaves (we will call it *the size of decision tree*). Note that this definition of a size leads to the same simplest tree as when we would count the number of nodes for binary trees. This is not necessarily valid for non-binary trees – assume that the variable B_1 is binary and B_2 has 60 distinct values, then the tree T_1 (the root splits by B_1 and then all nodes split by B_2) would have 123 nodes and the tree T_2 (the root splits by B_2 and then all nodes split by B_1) would have 181 nodes.

Definition 8 (Size of the Decision Tree) *Let* $\mathbf{T}$ *be a decision tree. Then, the size of the decision tree* $\mathbf{T}$ *is the number of its leaves (i.e.* $|W|$*). The size of the decision tree will be denoted as* $|\mathbf{T}|$.

The tree with a maximum size is the full tree and it has 2^n leaves. Let us remind that the full tree always represents all distributions. Now, we will define the *optimal decision* tree as the tree which is the smallest one from all trees representing a given probability distribution.

Definition 9 (Optimal decision tree) *Let* P *be a probability distribution. Let* $\mathcal{T} = \{\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_t\}$ *be a set of all decision trees which represent the probability distribution* P*. Then* $\mathbf{T}^* \in \mathcal{T}$ *is the* optimal decision tree with respect to P*, if* $|\mathbf{T}^*| \leq |\mathbf{T}|\ \forall \mathbf{T} \in \mathcal{T}$.

Note that for a given distribution P, we will use the abbreviated notion *the optimal decision tree* instead of the notion *the optimal decision tree with respect to* P with the same meaning.

In Figure 4.5, we have four decision trees. Their tree CPDs are derived from P_1 (the distribution defined in Table 4.1 on page 48), but only (b), (c), (d) represent the probability distribution P_1 and (b) is the optimal one.

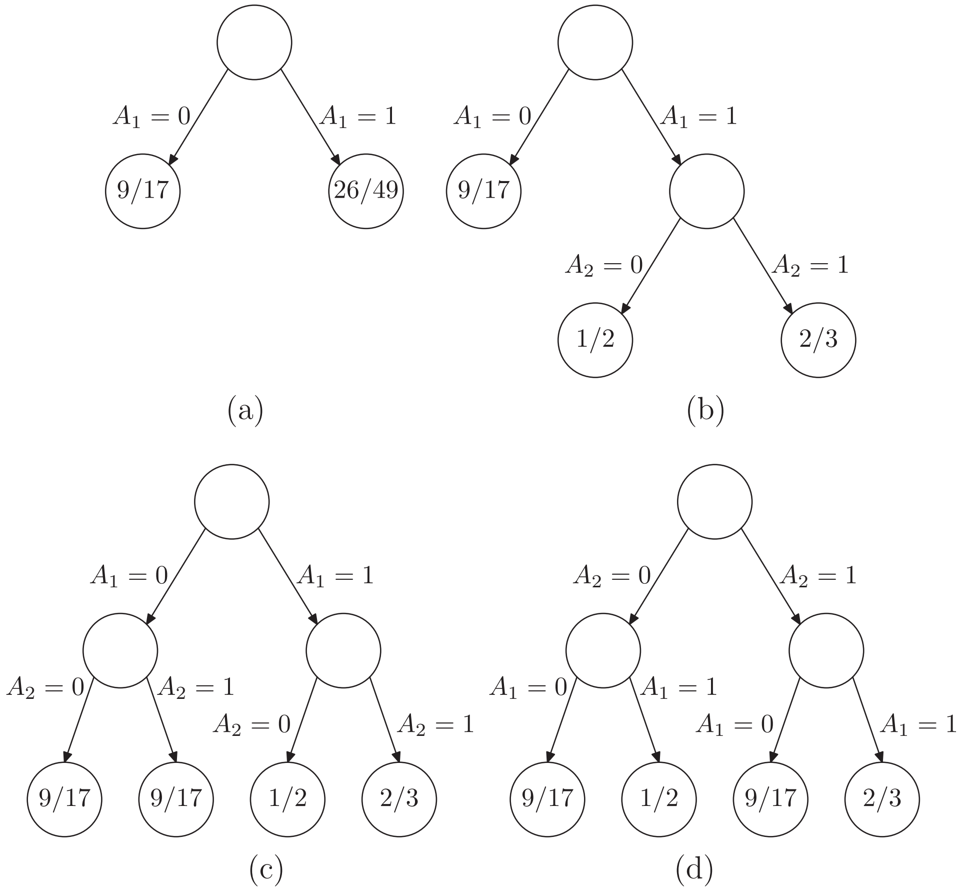


Figure 4.5: Four decision trees and distribution P_1

In general, more than one tree can be optimal. Consider the distribution P_2 from Table 4.4. Let a, b, c, d, e be distinct numbers from interval $[0, 1]$.

Table 4.4: Conditional probability distribution P_2. For this distribution, there exist more than one optimal decision tree (in this case there exist three optimal decision trees, which are not equivalent). Note that $a, \ldots, e$ stands for distinct numbers between 0 and 1.

A_1	A_2	A_3	$P(A_0 \vert A_1, A_2, A_3)$
0	0	0	a
0	0	1	d
0	1	0	c
0	1	1	c
1	0	0	a
1	0	1	b
1	1	0	e
1	1	1	b

We can find three decision trees representing P_2, which are not equivalent to each other. Two of them are shown in Figure 4.6.

It is easy to show that there are exactly three equivalence classes of optimal trees representing the distribution P_2. Every tree in every equivalence class has six leaves. There does not exist a tree representing P_2 with a lower number of leaves.

Note that for non-binary trees, we can find an example of the distribution on two variables only, where there exist at least two non-equivalent optimal decision trees.

We will define later a particular class of distributions (we will call them faithful, or weak faithful distributions), for which the only tree is optimal (the only one equivalence class is optimal respectively).

4.1.5 Operations with Decision Trees

We will define some operations on decision trees. These operations transform one decision tree representing a given distribution to another one representing the same distribution. These operations will be used to reach our goal: to find the optimal decision tree. The first operation we will define is a

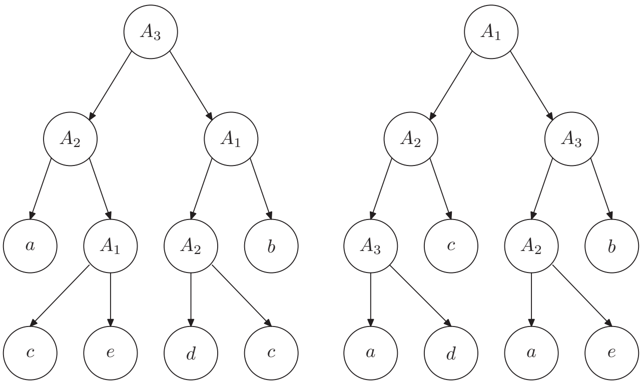


Figure 4.6: Two non-equivalent optimal decision trees for P_2.

prune operation. We will use only harmless prunes (with respect to a given probability distribution), it means that the prune operation combines two leaves with identical conditional probabilities $P(A_0|C_1) = P(A_0|C_2)$ into one leaf (to be more precise, it combines two leaves with their parent, which becomes a leaf). The prune operation is shown in Figure 4.7 (note that in this Figure, there is only a subtree which is a subject of change – one node and two leaves are shown, other parts of the tree remain unchanged).

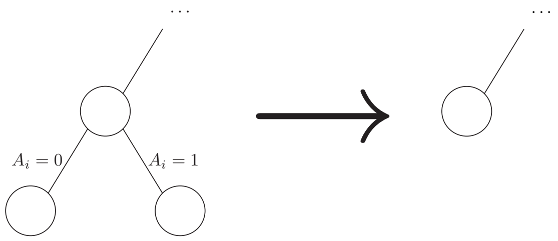


Figure 4.7: The prune operation

The following definition introduces this operation formally.

Definition 10 (Prune operation) *Let $\mathbf{T} = (T, \mathcal{A}, m, s)$ be a decision tree. Let v, v' be two leaves in T with a common parent v_x. Let $\mathbf{T'} = (T', \mathcal{A}, m', s')$ be a decision tree constructed as follows*

1. $V(T') = V(T) \backslash \{v, v'\}$,

2. $E(T') = E(T) \backslash \{(v_x, v), (v_x, v')\}$*,*

3. $m' = m \upharpoonright U(T')$*, i.e.* m' *is restricted to a new set of internal nodes,*

4. $s' = s \upharpoonright E(T')$*,*

Then, we say that $\mathbf{T}'$ is constructed from $\mathbf{T}$ by one prune operation.

Remark 6 *We will use only harmless prunes. We define that the prune operation is* harmless with respect to a given distribution P*, if* $P(A_0|C_v) = P(A_0|C_{v'})$*, where* $C_v, C_{v'}$ *are conditions assigned to leaves* v, v'*. When we will use* prune operation *in this thesis, we mean* harmless prune operation with respect to a given distribution P.

Remark 7 *The prune operation transforms* T *to* T'*,* T *and* T' *are not equivalent (they have a different number of leaves), but they are CPD equivalent (i.e. they represent the same CPD) for all distributions, which are represented by* T'*.*

The next operation on decision trees is a *parent-child exchange*. This operation locally rearranges the inner-structure of the tree without affecting remaining parts of the tree. Assume that node v_x defines a splits by $m(v_x)$ and the split variables assigned to both children v, v' of v_x are the same (i.e. $m(v) = m(v')$). In any path from the root to a leaf which includes v_x, we will split by $m(v_x)$ and $m(v)$. The local change means the split first by $m(v)$ and then (in both nodes) by $m(v_x)$. This situation does not modify leaves and their conditions, when we rearrange subtrees of v and v' correctly. The detail of this operation is shown in Figure 4.8.

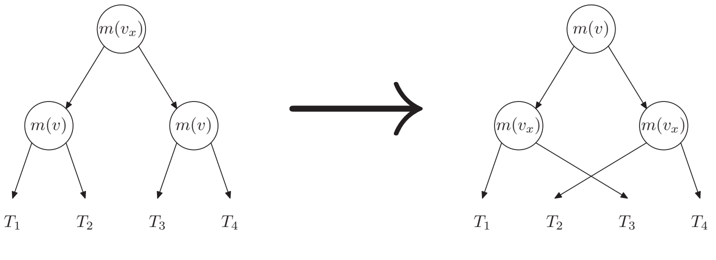


Figure 4.8: The parent-child exchange operation. Symbols T_1 to T_4 denote subtrees.

In a formal definition, we have to distinguish which edge goes to the left (which is assigned the value 0 of the split variable) and which one goes to the right. Two edges (edges connecting roots of T_2 and T_3 with their parents) will be replaced by other two edges as shown in Figure 4.8.
Note that subtrees T_1 to T_4 may also be one-node subtrees (leaves).

Definition 11 (Parent-child Exchange Operation)
Let $\mathbf{T} = (T, \mathcal{A}, m, s)$ be a decision tree. Let $v_x \in U$ and let $v, v' \in U$ be its children. Let $m(v) = m(v')$.
Assume without loss of generality that $s((v_x, v)) = 0$ and $s((v_x, v')) = 1$.
We denote

- *v_1 a child of v such that $s((v, v_1)) = 0$,*
- *v_2 is a child of v such that $s((v, v_2)) = 1$,*
- *v_3 is a child of v' such that $s((v', v_3)) = 0$ and*
- *v_4 is a child of v' such that $s((v', v_4)) = 1$.*

Let $\mathbf{T}' = (T', \mathcal{A}, m', s')$ be a decision tree constructed as follows

1. $V(T') = V(T)$
2. $m'(w) = m(w)$ *for all* $w \neq v, v', v_x$
3. $m'(v_x) = m(v)$
4. $m'(v) = m(v_x), m'(v') = m(v_x)$
5. $E(T') = E(T) \setminus \{(v, v_2), (v', v_3)\} \cup \{(v, v_3), (v', v_2)\}$
6. $s'(f) = s(f) \restriction (E(T) \cap E(T'))$
7. $s'((v, v_3)) = 1, s'((v', v_2)) = 0$

Then, we say that $\mathbf{T}'$ is constructed from $\mathbf{T}$ by one parent-child exchange operation.

Remark 8 *Let $\mathbf{T}$ be a decision tree. Let $\mathbf{T}'$ be a decision tree constructed from $\mathbf{T}$ by one parent-child exchange operation. Then, $\mathbf{T}$ and $\mathbf{T}'$ are equivalent.*

4.1.6 Faithfulness and its properties

Most common algorithms for learning decision trees use the greedy principle. This principle can fail on some CPDs (conditional probability distributions) – it does not find a tree which represents the CPD. Consider a distribution where dependent variable A_0 is calculated as a function of independent variables A_1 XOR A_2 and $P(A_1, A_2)$ is an uniform distribution (this distribution is called *xor distribution*). For this distribution, the greedy algorithm will stop with a one node tree, but this tree does not represent a given distribution. Note that XOR distribution is not a full distribution (as defined on page 48), but we may construct an example of full distribution, for which the greedy principle fails.

Moreover, we will look for an *optimal* decision tree. We can divide the solution of this problem in two phases.

In the first phase, we will find the tree which represents a given distribution. In the second phase, we will start with the tree from the first phase (i.e. with the tree which represents a given distribution) and will find the optimal decision tree using a sequence of operations defined in previous subsection. We will show the algorithm for these two phases. We will also check the complexity of this algorithm.

In this subsection, we introduce assumptions for the distribution (we will call them weak and strong faithfulness), under which our algorithm will work correctly and with the acceptable complexity.

For the first phase of our algorithm, we will need the strong faithfulness, for the second phase, the weak faithfulness will be sufficient. We will show that, under assumption of the weak faithfulness, it is sufficient to use only two operations (prune and parent-child exchange) in the second phase to find the optimal decision tree. Moreover, every sequence of prune and parent-child exchange operations leads into the tree, from which we can get the optimal decision tree by another sequence of prune and parent-child exchange operations (so we will not get into a "dead end" by applying prune and parent-child exchange operations).

Applying our algorithm to a distribution, which is not faithful, may lead into a "dead end" situation (e.g. in situation, where we are not able to reduce the number of leaves by any combination of prune and parent-child exchange operations, but the current tree is not optimal). Next, we will show that the random distribution is strong faithful with probability one.

In this thesis, we do not want to introduce operations which add extra leaves

Table 4.5: Conditional probability distribution P_3, which is not faithful, and for which algorithm may fail (* means any value, $a \neq b$.)

A_1	A_2	A_3	$P(A_0 \mid A_1, A_2, A_3)$
$*$	0	0	a
$*$	1	0	b
0	$*$	1	a
1	$*$	1	b

and nodes. It could lead to uncontrolled growth of the tree and there is no guarantee that the tree will be reduced afterwards. It could also affect the complexity (every newly created leaf should be removed later, or, to be more precise, for every newly created leaf there have to exist one or more other leaves which will be removed later).

Now, we show an example of non-faithful distribution, for which the algorithm may fail. The conditional probability distribution P_3 is shown in Table 4.5.

In Figure 4.9 on the left, there is the decision tree representing P_3 shown, which cannot be transformed to the optimal tree by application of a sequence of prune and parent-child exchange operations only. Note that we are not able to neither apply any parent-child exchange nor prune operation.

The optimal tree which represents the conditional probability distribution P_3 is shown in Figure 4.9 on the right.

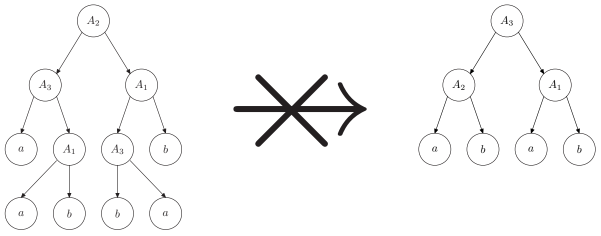


Figure 4.9: Non weak faithful distribution. Both trees represent P_3, tree on the right is optimal. It is impossible to transform the tree on the left to the optimal tree by a sequence of prune and parent-child exchange operations.

Definition 12 (DT defining weak faithfulness)
Let P be a distribution. Let $\mathbf{T} = (T, \mathcal{A}, m, s)$ be a decision tree which represents P. Let $v_1, v_2, \ldots, v_k$ be all leaves of $\mathbf{T}$ and $C_1, C_2, \ldots, C_k$ be their corresponding conditions. Let $P_T(A_0|C_i) \neq P_T(A_0|C_j)$ for $i \neq j$. Then, we say that the decision tree $\mathbf{T}$ is a tree defining a weak faithfulness*, or $\mathbf{T}$ is a* wFF tree*.*

Definition 13 (Weak Faithfulness for Distributions) *Let P be a distribution. We say, that the distribution P is* weak faithful *(wFF DT), if there exists a wFF tree $\mathbf{T}$ which represents P.*

We will look at some properties of weak faithful distributions.

Theorem 9 *Let P be a weak faithful distribution. Then, all optimal decision trees are equivalent.*

Proof. Obvious. It is sufficient to show two claims – first, the wFF tree $\mathbf{T}$ is optimal and second, any optimal tree is equivalent to $\mathbf{T}$.

Now, we show that $\mathbf{T}$ is optimal. All leaves of $\mathbf{T}$ denoted by w_j have different probabilities $P(A_0|C_j)$, where C_j is a condition assigned to a leaf w_j. (it follows from the definition of wFF tree). Trees which have a lower size than $\mathbf{T}$ does not represent P – every tree, which represents P, have exactly $|\mathbf{T}|$ distinct values of $P(A_0|A_1, A_2, \ldots, A_n)$. Therefore, the tree which represents P must have at least $|\mathbf{T}|$ leaves.

The remaining claim is that any optimal tree is equivalent to $\mathbf{T}$. It is clear from the following.

1. Trees with a bigger size than $|\mathbf{T}|$ are not optimal ($\mathbf{T}$ has the lower number of leaves).
2. Tree $\mathbf{T}'$ for which holds $|\mathbf{T}'| = |\mathbf{T}|$ represents P iff its set of conditions assigned to leaves $C_1, C_2, \ldots, C_{|\mathbf{T}|}$ is the same as for the tree $\mathbf{T}$ (we have $|\mathbf{T}|$ distinct values of $P(A_0|A_1, A_2, \ldots, A_n)$ and there is only one possibility how $C_1, C_2, \ldots, C_{|\mathbf{T}|}$ can be defined, so the optimal tree must be equivalent to T).

$\square$

Corollary 10 *Every wFF tree is optimal.*

Remark 11 *We will use the notion* classical decision tree growing algorithm. *This is any algorithm for decision tree growing which satisfies the following assumption. When the decision tree algorithm is deciding whether it is possible to make a split of the vertex v and if there exists a variable A_i such that assigning the variable A_i to the node v we would obtain children v', v'' of v with corresponding conditions $C_{v'}$ and $C_{v''}$ such that $P(A_0|C_{v'}) \neq P(A_0|C_{v''})$, then algorithm does not stop with v as a leaf. Note that we do not have any requirements which variable is chosen if more variables satisfy this condition.*

Definition 14 (Strong Faithfulness for Distributions)
Let P be a probability distribution. We say that P satisfies strong faithfulness for decision trees *(or sFF DT) if P satisfies wFF DT and for every decision tree T which represents P the following holds : Let $v \in U$ be a node in T and let $v', v'' \in V$ be its children. We denote $C_v, C_{v'}$ and $C_{v''}$ conditions assigned to the corresponding leaves. Let $P(A_0|C_{v'}) = P(A_0|C_{v''})$ $(= P(A_0|C_v))$. Then for every node v''' in the subtree defined by the node v (in any depth, including leaf nodes) there is $P(A_0|C_v) = P(A_0|C_{v'''})$.*

This definition guarantees that the greedy tree growing will not stop prematurely. The rationale follows. Let P be a probability distribution and let $\mathbf{T}_F$ be any full tree (we know that $\mathbf{T}_F$ represents the probability distribution P). We assume that growing was stopped with the tree $\mathbf{T}_G$ because for every node v which can be split into v' and v'' by installing the variable A_i, there is $P(A_0|C_{v'}) = P(A_0|C_{v''})$. Due to the definition of sFF DT, every node v''' in the subtree defined by a vertex v has $P(A_0|C_{v'''}) = P(A_0|C_v)$. This holds for every such vertex v and therefore $\mathbf{T}_F$ represents the same distribution as $\mathbf{T}_G$. And because the full tree $\mathbf{T}_F$ represents P, the tree $\mathbf{T}_G$ also represents P.

4.2 Theorems about Finding The Optimal Decision Tree

In the previous section, we have defined strong faithful distributions. We will present and prove our main theorem for strong faithful distributions – we can find the optimal decision tree for a given distribution using the classical decision tree growing algorithm. The proof of the theorem is based on two key lemmas – the first one shows that we can find a tree that represents data (which is not necessarily optimal), and the second one shows the way how to find the optimal tree from the tree from the first lemma. Finding a tree which represents the data requires the assumption of sFF DT, ”restructuring” the tree to find the optimal tree requires only wFF DT. We know that the strong faithfulness implies the weak faithfulness, so the following theorems will be proven under a ”common” assumption of the strong faithfulness.

Theorem 12 (Finding the optimal tree from data) *Let P be a distribution $P(A_0, A_1, A_2, \ldots, A_n)$ which is strong faithful (sFF DT). Let $\mathbf{T}^*$ be an optimal tree which represents the distribution P. Then, there exists a sequence $\mathcal{K}$ of trees $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_{RD}, \mathbf{T}_{RD+1}, \ldots, \mathbf{T}_k$ such that*

- $\mathbf{T}_1$ *is a tree with a single node (=root node),*
- $\mathbf{T}_{RD}$ *is a tree that represents P (not necessarily optimal),*
- *for all $i = 1, 2, \ldots, RD-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one step of the classical algorithm for decision tree growing,*
- *for all $i = RD, RD+1, \ldots, k-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one prune operation or one parent-child exchange operation,*
- $\mathbf{T}_k = \mathbf{T}^*$.

Proof. To prove this theorem, we will introduce two lemmas – Lemma 13 and Lemma 14. The proof of this theorem immediately follows from the these two lemmas.

□

Lemma 13 (Algorithm finds a tree that represents a distribution) *Let P be a distribution, which satisfies the strong faithfulness for decision trees. Let $\mathbf{T}$ be a tree which is the output from the classical algorithm for decision tree growing. Then, $\mathbf{T}$ represents the distribution P.*

Proof. The proof immediately follows from the definition of the strong faithfulness. Let **T** be a tree which is a result of a classical algorithm for decision tree growing. Our assumption on this algorithm says that every node v, which would be split into children v_1 and v_2, holds that $P(A_0 = 1|C_{v_1}) = P(A_0|C_{v_2})$ and therefore also $P(A_0 = 1|C_{v_1}) = P(A_0|C_{v_2}) = P(A_0|C_v)$. Let $\mathbf{T}'$ be a tree, which is based on **T**, but all branches are "grown" to the full tree (by the same greedy algorithm, neglecting a stopping criteria for decision tree growing, even when we would obtain identical conditional probabilities in both children). The situation is shown in Figure 4.10.

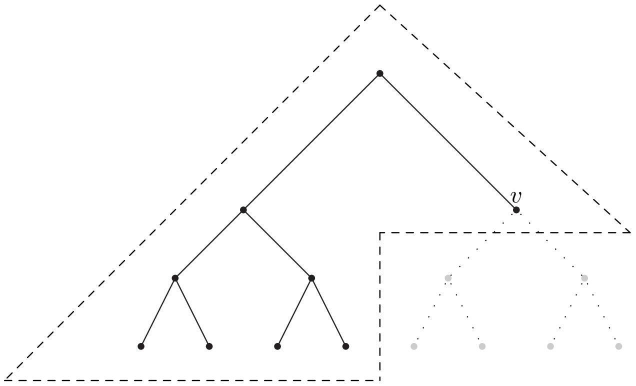


Figure 4.10: Two decision trees. The decision tree **T** is a tree drawn in full lines, bounded by dashed lines. The tree $\mathbf{T}'$ is a tree with all drawn nodes and branches (including dotted branches and grey nodes). Note that v is a leaf in the tree **T**, but an inner-node in the tree $\mathbf{T}'$.

It is obvious that $\mathbf{T}'$ represents the distribution P ($\mathbf{T}'$ is the full tree). Due to the strong faithfulness (we will use the definition of the strong faithfulness applied to a specific case T'), all descendants of v have assigned conditional probability $P(A_0|C_v)$, so both **T** and $\mathbf{T}'$ represent the same CPD and therefore **T** represents P.

□

In the next lemma, we will show that for a given wFF DT distribution P and for any tree **T** which represents the distribution P, there exists a sequence of prune and parent-child exchange operations which transforms **T** to the optimal tree $\mathbf{T}^*$. Note that this lemma is not a "cookbook" how to construct this sequence, we will prove only that this sequence exists.

Lemma 14 (Transformation to the optimal tree) *Let P be a distribution which satisfies the weak faithfulness for decision trees (wFF DT). Let $\mathbf{T}$ be a tree which represents the distribution P. Let $\mathbf{T}^*$ be the optimal tree which represents the distribution P. Then, there exists a sequence $\mathcal{S}$ of trees $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_k$ such that*

1. *all trees in $\mathcal{S}$ represent the distribution P,*
2. *$\mathbf{T}_1 = \mathbf{T}$ and $\mathbf{T}_k = \mathbf{T}^*$,*
3. *for all $i = 1, 2, \ldots, k-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one prune operation or one parent-child exchange operation.*

The proof of this lemma will be provided after introducing several supplementary lemmas.

The first lemma required for the proof is Lemma 15. It says that when we have any tree which represents the distribution P which is weak faithful, then from the definition there exists some tree $\mathbf{T}^*$ which is the optimal decision tree (defines wFF DT), in which all leaves w_i has different probabilities $P(A_0|C_{w_i})$. We will denote the root of this optimal tree as v_r^* and the variable assigned to v_r^* as A_R. Then, the condition in all leaves of any tree which represents P must include the literal $A_R = 0$ or the literal $A_R = 1$. It is clear because in the optimal tree, in both root's branches we have different conditional probabilities $P(A_0|C_{w_i})$ in leaves w_i. So every condition assigned to leaf in every tree which represents P has to have the literal $A_R = 0$ or $A_R = 1$ in its definition. That means that A_R has to be on every path from the root to the leaf in every optimal tree which represents P.

Lemma 15 *Let P be a distribution $P(A_0, A_1, A_2, \ldots, A_n)$ which satisfies the weak faithfulness for decision trees (wFF DT). Let $\mathbf{T} = (T, \mathcal{A}, m, s)$ be a tree which represents the distribution P. Let $\mathbf{T}^* = (T^*, \mathcal{A}, m^*, s^*)$ be the optimal tree which represents the distribution P. Let v_r^* be the root of the tree T^*. Let $m^*(v_r^*) = A_R$. Then, for every $v \in W(\mathbf{T})$, A_R is on the path to v.*

Proof. By a contradiction. Assume that there exists a branch which do *not* contain A_R. So there exists a leaf v, which does not include A_R in a condition assigned to v. But $\mathbf{T}^* = (T^*, \mathcal{A}, m^*, s^*)$ is the optimal tree which is equivalent to the decision tree defining wFF DT. Now, we will look at the condition assigned to the leaf v in $\mathbf{T} = (T, \mathcal{A}, m, s)$ and we will look which value should be assigned to $P(A_0|C_v)$. We know that both trees represent P, so this conditional probability must be equal in both trees. For the decision tree defining wFF DT, the probability $P(A_0|C)$ *must* be different for leaves where is $A_R = 0$ and leaves where is $A_R = 1$. Due to the full distribution assumption, the tree $\mathbf{T} = (T, \mathcal{A}, m, s)$ and the tree $\mathbf{T}^* = (T^*, \mathcal{A}, m^*, s^*)$ cannot represent the same distribution – this is the contradiction. So all branches contain A_R.

□

In the next lemma, we will keep a notation that A_R is the variable assigned to the root of the optimal tree $\mathbf{T}^*$ and we will prove that in the tree $\mathbf{T}$ (not necessarily optimal; in any tree which represents P) there exists a vertex v which has assigned the variable A_R (i.e. $m(v) = A_R$) and its neighbour v' has also assigned the variable A_R. The set of vertices which have assigned the variable A_R will be denoted by V_R and the parent of v (and v') will be denoted as v_x. As a result, the variable A_R assigned to these two vertices (v, v') can be "moved up" by a parent-child exchange operation (with v_x).

Lemma 16 *Let P be a distribution $P(A_0, A_1, A_2, \ldots, A_n)$ which satisfies the weak faithfulness for decision trees (wFF DT). Let $\mathbf{T} = (T, \mathcal{A}, m, s)$ be a tree which represents the distribution P. Let $\mathbf{T}^* = (T^*, \mathcal{A}, m^*, s^*)$ be the optimal tree which represents the distribution P. Let v_r^* be the root of the tree T^*. We will denote $m^*(v_r) = A_R$. Let V_R be a set of nodes $v'' \in U(\mathbf{T})$ for which $m(v'') = A_R$. Let $v \in V_R$ be a node in the highest depth from all nodes in V_R. (If there are more nodes into the highest depth, we can choose any of them. For all of them the lemma holds). Assume that v is not a root of $\mathbf{T} = (T, \mathcal{A}, m, s)$. Let v' be a neighbour of v and v_x be their parent. Then, $m(v') = A_R$.*

Proof. We assume (for a contradiction) that a node in the highest depth $v \in V_R$ (note that at least one node has assigned the variable A_R due to Lemma 15) has a neighbour v' which has assigned the other variable. Note that A_R can be on every path from the root to the leaf maximally one time (the second occurrence is useless for binary variables; this fact is guaranteed by our definition of the binary decision tree), so neither v_x nor any parent of any level up to the root can have assigned the variable A_R. From Lemma 15, A_R is on every path to a leaf. So splitting by A_R must be done by any vertex below the v' (in some descendant of v') and that is the contradiction with the assumption that v is a node with the highest depth which has assigned the variable A_R.

□

Lemma 17 *Let P be a distribution $P(A_0, A_1, A_2, \ldots, A_n)$ which satisfies the weak faithfulness for decision trees (wFF DT). Let $\mathbf{T}$ be a tree which represents the distribution P. Let $\mathbf{T}^*$ be an optimal tree which represents the distribution P. Let v_r^* be the root of T^*. Let $m^*(v_r^*) = A_R$. Then, there exists a sequence $\mathcal{S}$ of trees $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_k$ such that*

1. *all trees in $\mathcal{S}$ represent the distribution P,*
2. *$\mathbf{T}_1 = \mathbf{T}$ and $\mathbf{T}_k$ has the variable A_R assigned in its root node,*
3. *for all $i = 1, 2, \ldots, k-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one parent-child exchange operation.*

Proof. We will use Lemma 16 several times. To ensure that the entire process is finite, we may define h as a sum of depths of nodes, which have assigned the variable A_R. So when node v has assigned the variable A_R and its distance from the root is b, it contributes to h with $b+1$. We will use the specified Lemma repeatedly and after each step, h will be decreased. We will stop when $h = 1$, i.e. A_R is in the root. The step is: from Lemma 16, we have vertices v, v' and their parent v_x. Nodes v and v' have assigned the variable A_R. So we may apply a parent-child exchange of v, v' with v_x. This step will decrease h, because A_R was moved up by 1 in one branch and removed (due to the parent-child exchange) from the other branch. The parent-child exchange operation does not modify any condition assigned to decision tree leaf, so Lemma 16 can be applied repeatedly until $h = 1$.

□

We are going to prove the main point of Lemma 14 by the following Lemma. We will show that there exists a sequence such that in the first step we will use only parent-child exchange operations and in the second step we will use prune operations only to get $\mathbf{T}^*$ from our $\mathbf{T}$. The first step is almost proved (it will be proven by repeated use of Lemma 17). Lemma 18 will finish the first step – that there exists a tree $\mathbf{T}'$ which represents P (the result from the first step), which can be easily pruned into $\mathbf{T}^*$ (so the second step will be easy to show). In Lemma 18, we will use the notions *restriction* and *mapping*. Mapping r used in the following lemma only 'renames' vertices, allowing $\mathbf{T}'$ to have more vertices than $\mathbf{T}^*$. Also, assigned variables must correspond. In fact, $\mathbf{T}'$ is based on $\mathbf{T}^*$, only some leaves of $\mathbf{T}^*$ are grown to next levels.

Lemma 18 *Let P be a weak faithful distribution $P(A_0, A_1, A_2, \dots, A_n)$. Let $\mathbf{T}$ be a tree which represents the distribution P. Let $\mathbf{T}^*$ be the optimal tree which represents the distribution P. Then, there exists a tree $\mathbf{T}'$ and a sequence $\mathcal{S}$ of trees $\mathbf{T}_1, \mathbf{T}_2, \dots, \mathbf{T}_k$ such that*

1. *the tree $\mathbf{T}^*$ is a restriction of the tree $\mathbf{T}'$, i.e. that there exists a mapping $r : V(\mathbf{T}^*) \to V(\mathbf{T}')$*

   (a) *for the root v_0 of the tree T, $r(v_0)$ is the root of T',*

   (b) *for $v, v' \in V(T)$, if $(v, v') \in E(T)$, then $(r(v), r(v')) \in E(T)$,*

   (c) *for all $v \in U(\mathbf{T})$, there is $m(v) = m'(r(v))$,*

   (d) *for all $(v_1, v_2) \in E(\mathbf{T})$, if $v_1' = r(v1), v_2' = r(v2)$ then $s((v_1, v_2)) = s((v_1', v_2'))$*

2. *all trees in $\mathcal{S}$ represent the distribution P,*

3. *$\mathbf{T}_1 = \mathbf{T}$ and $\mathbf{T}_k = \mathbf{T}'$,*

4. *for all $i = 1, 2, \dots, k-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one parent-child exchange operation.*

Proof. We will use Lemma 17 several times. From the previous lemma, we can transform out tree by a sequence of parent-child exchange operations to the tree T_{i_1} which has assigned the variable A_R in its root. This lemma will be used repeatedly for all non-leaf nodes of the optimal tree from top to down. We can now simply apply this lemma (or a simple generalization of this lemma) to both children of the root (assuming a marginal distribution with respect to the condition corresponding to this node, which is also wFF DT) and so on until we reach leaves in all branches.

□

Proof of Lemma 14. We assume that our distribution is wFF DT, so there exists the wFF tree $\mathbf{T}'$ (from the definition of wFF DT). This tree is optimal due to the corollary 10. Note that all optimal trees are equivalent due to theorem 9 so we may assume without loss of generality that $\mathbf{T}' = \mathbf{T}^*$. We will use the notation $\mathbf{T}^*$ for both optimal tree and wFF tree (tree defining weak faithfulness).
Our goal is to show that we can transform the tree $\mathbf{T}$ to $\mathbf{T}^*$ in two steps. In the first step, we will use only parent-child exchange operations and we will transform the tree $\mathbf{T}$ to the tree $\mathbf{T}''$ such that $\mathbf{T}''$ can be transformed to $\mathbf{T}^*$ by a sequence of prune operations. To be more specific, $\mathbf{T}''$ and $\mathbf{T}^*$ will have assigned identical variables in the root and in all non-leaf nodes in identical positions (i.e. function m restricted to the optimal tree will be identical for $\mathbf{T}''$ and $\mathbf{T}^*$). In the second step, we will prune the tree $\mathbf{T}''$ to get the tree $\mathbf{T}^*$. Lemma 18 will prove the first step, the second step will be easily shown:

We have the tree $\mathbf{T}'$ which is the result of Lemma 18 and we only need to prune it. We will use the notation from the mentioned lemma.

Let $v^* \in W(\mathbf{T}^*)$ be any leaf of the optimal tree. Denote $v' = r(v^*)$. We know that both $\mathbf{T}'$ and $\mathbf{T}^*$ represent the same distribution P and therefore for any descendant v'' of v', there is $P(A_0|C_{v''}) = P(A_0|C_{v'})$. So the entire subtree of v' can be transformed into a single node by a sequence of prunes.

This step will be repeated for every leaf of the optimal tree $\mathbf{T}^*$.

□

Note that we have shown that for any tree which represents a weak faithful distribution P (not only for the tree which is the result of the classical algorithm for decision tree growing), there exists a sequence of parent-child exchange and prune operations that leads to the optimal tree. This is very crucial for an algorithmic approach. Moreover, when we apply any parent-child or prune operation, it is not possible to get in the "situation with no exit" (i.e. to the situation that no sequence of prune and parent-child exchange operations can transform the current tree to the optimal one). After applying the parent-child exchange operation (PCE) , we can undo it by the second parent-child exchange operation. After applying the prune operation, we also have the tree which represents the distribution P, but we have proved that from this tree, we can get the optimal one by another set of prune and parent-child exchange operations.

Corollary 19 (We cannot "lose track" by prune or PCE) *Let P be a weak faithful distribution $P(A_0, A_1, A_2, \ldots, A_n)$. Let $\mathbf{T}'$ be a tree which represents the distribution P. Let $\mathbf{T}$ be a tree which is constructed from $\mathbf{T}'$ by one prune operation or one parent-child exchange operation. Let $\mathbf{T}^*$ be the optimal tree which represents the distribution P. Then, there exists a sequence $\mathcal{S}$ of trees $\mathbf{T}_1, \mathbf{T}_2, \ldots, \mathbf{T}_k$ such that*

1. *all trees in $\mathcal{S}$ represent the distribution P,*
2. *$\mathbf{T}_1 = \mathbf{T}$ and $\mathbf{T}_k = \mathbf{T}^*$,*
3. *for all $i = 1, 2, \ldots, k-1$, the tree $\mathbf{T}_{i+1}$ is constructed from $\mathbf{T}_i$ by one prune operation or one parent-child exchange operation.*

Proof. Immediately follows from Lemma 14.

□

So we can design the algorithm to *apply* the prune operation whenever it is possible. For weak faithful distributions, it have been proved that the optimal tree can be reached.

Note that this corollary is not valid for distributions that are not weak faithful – we *can* get in a situation that there is no way how to get the optimal tree when we will use only parent-child exchange and prune operations.

There is one more thing we can easily prove and which can help us to understand what is the tree defining wFF DT and its properties.

Remark 20 *Let P be a weak faithful distribution.Let $\mathbf{T}^*$ be the optimal decision tree which represents the distribution P.Then, $\mathbf{T}^*$ is a wFF tree.*

Proof. Immediately follows from Theorem 9 and definitions.

□

4.3 Algorithm

In this section, we describe the algorithm which finds the optimal decision tree from the distribution which satisfies strong faithfulness. The subsection 4.3.1 shows a high-level overview of the algorithm, following subsections show the algorithm in detail – a pseudocode for the entire algorithm and individual procedures/functions will be provided.

4.3.1 Algorithm overview

The algorithm, which we propose here, is based on any classical algorithm for decision tree growing (for example, CART). The only change is in the post-pruning phase, which is totally changed. The basic idea is shown in the following schema.

The classical decision tree algorithm with the post-pruning

The growing phase

- Split nodes until no leaf can be split in two children with different conditional probabilities assigned (or some stopping rule is met, see section 2.2)

The pruning phase

- *Repeat*
  - found:=Find two leaves v_1, v_2
    - with the same conditional probability assigned to the leaf
    - which **are** neighbours
  - *If* found, *Then* prune v_1, v_2
- *Until* not found

The FindOptimalTree algorithm

The growing phase

- Split nodes until no leaf can be split in two children with different conditional probabilities assigned (or some stopping rule is met, see section 2.2)

The pruning phase

- *Repeat*
  - found:=Find two leaves v_1, v_2
    - with the same conditional probability assigned to the leaf
    - which **can be** neighbours
  - *If* found, *Then* prune v_1, v_2
- *Until* not found

The only difference in algorithms is which pair of leaves can be pruned. The only change is in the "... which can be neighbours" statement. Note that this operation is not so easy as it may appear. So we will focus in the following text on testing whether two leaves can be neighbours.

The algorithm for testing whether two leaves can be neighbours after some sequence of parent-child exchange operations (overview)

The condition whether leaves v_1 and v_2 can be neighbours after some sequence of PCE operations may be tested in the following way.

1. **test "continue/fail condition"** – leaves, which can be neighbours, have to satisfy this condition: when we look at condition assigned to both leaves v_1, v_2, then variables in C_{v_1} and C_{v_2} are the same and values are the same except one (denote this variable A_{CR}). Consequences of this include that both leaves have to be in the same depth in the tree.

2. **find the Common Root (CR) of the v_1 and v_2** – common root is a node which is a parent (not necessarily direct parent, maybe grandparent, maybe great-grandparent and so on) in the highest depth in the tree. Note that the Common Root can be also found by its another property – it's a parent of v_1 and v_2 which defines a split by the variable A_{CR}.

3. **define variables to be shifted down** – this set of variables L is defined as *variables which define splits from the child of CR into the leaf* v_1 (or, into the leaf v_2, which is equivalent). In the example in Figure 4.11, there is $L = \{A_2, A_3\}$.

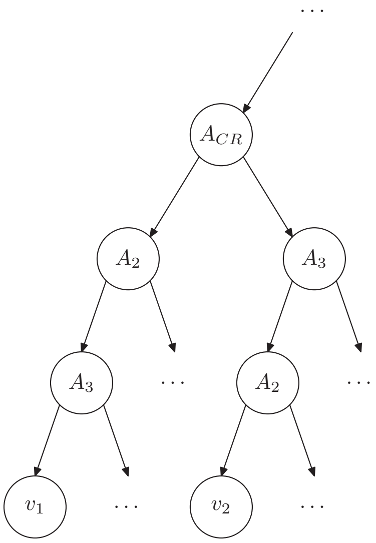


Figure 4.11: Checking whether two vertices v_1 and v_2 can be neighbours.

4. **parent-child exchange with CR** – try if any of elements of L can be moved to both children of CR. In situation that we find such element,

we will move it to both children of CR , next we will apply a parent-child exchange operation for CR and its children and we will repeat this step until L is empty or no variable of L can be moved to both children of CR. Note that Figure 4.11 does not provide enough information whether any variable of L can be moved to both children of CR – we need to know more information. The example will be provided later, when we will describe the algorithm in a more detail.

5. **explore the result** – nodes v_1 and v_2 are neighbouring leaves iff L is empty. In this situation, we will prune them.

The most important step from the practical point of view and the most interesting step from the theoretical point of view is the step 4. This item locally does an operation which constructs a sequence that makes nodes to be neighbours and able to prune. From the theoretical point of view, it was proved that for a weak faithful distribution (see Lemma 17 on page 70, which can be easily extended to a subtree) and leaves v_1 and v_2 for which $P(A_0|C_{v_1}) = P(A_0|C_{v_2})$ there exists at least one element of L that can be moved using a sequence of parent-child exchange operations to both children of CR (this also means that this item is included in all branches under the CR). Our algorithm tries attributes of L one by one, until L is empty or none of elements can be moved to both children of CR. The Corollary of this claim is that there exists a sequence of parent-child exchange operations, which transforms our original tree to the tree where v_1 and v_2 are neighbouring leaves and can be combined. It can be also proven that when more than one attribute can be moved to both children of CR, then we can choose any of these attributes as the first one and this leads to moving v_1 and v_2 to neighbouring nodes (note that for fixed v_1, v_2 and CR, every attribute which was moved to both children of CR, after parent-child exchange disappears from L and there is no way how to introduce new elements into L).

It is easy to show that this algorithm is polynomial in the number of leaves of the tree from the greedy phase of the CART algorithm (by estimating the complexity of every individual procedure; detailed study of the true complexity is out of the scope of this thesis and it is a subject of future work).

On the other hand, an exhaustive searching of all possible candidates for the optimal tree (which are smaller than the result from CART) is exponential in the number of leaves of the tree from CART.

In following sections, we will describe this algorithm in a more detail.

4.3.2 Algorithm FindOptimalTree summary

We describe this algorithm from scratch. Our algorithm (FindOptimalTree) provides an extension to any decision tree algorithm based on TDIDT and the greedy principle with post-pruning (for example, CART, ID3). The only change is in a post-pruning phase, which is totally changed (as it is shown in the beginning of Section 4.3.1). The algorithm uses several pocedures, which are called. The main structure of procedure inclusions is shown in the following scheme.

FindOptimalTree
↓
OptimiseTree
↓
JoinSucceeded
↓
CanBubbleToCRPlusOne
↓
CanBubbleToNode

The more detailed description of each procedure follows.

FindOptimalTree – procedure, which calls the growing procedure and then the optimising procedure *OptimiseTree*.

OptimiseTree – this procedure realises the extended post-pruning phase shown in the overview. It tries to find a pair of leaves with both identical probabilities of target (or with statistically insignificant difference for the finite version) with ability to be moved to the neighbour vertices by a sequence of parent-child exchange operations. The second property is tested by the function *JoinSucceeded*, only the continue/fail condition (see condition 1 in algorithm overview on page 76) is applied in the *OptimiseTree* procedure.

JoinSucceeded – function, which finds a sequence of PCE operations (if exists) for two leaves to be neighbours. It assumes that leaves are in the same depth in the tree (which is a consequence of the continue/fail condition). It arranges a list of variables which should be moved to the root of the subtree defined by CR node and uses multiple calls of the *CanBubbleToCRPlusOne* function.

CanBubbleToCRPlusOne – checks whether a given variable can be moved to both children of CR node; uses two calls of *CanBubbleToNode* function.

CanBubbleToNode – checks whether a given variable can be moved to a specified node.

Now, we will show the *FindOptimalTree* algorithm in detailed steps (in a pseudocode, with comments in the text). The main procedure is in Figure 4.12.

```
Function FindOptimalTree(Data)
    1. Tree:=GrowTree(Data)
    2. Tree:=OptimiseTree(Tree)
End Function
```

Figure 4.12: Steps of FindOptimalTree algorithm

This main procedure only says that finding the optimal tree consists of two phases – growing a tree (function GrowTree) and optimising a tree (function OptimiseTree). The function *GrowTree* will not be shown here, we will use a growing phase of a particular TDIDT based algorithm (examples are shown in Section 2.3). The only change is in the post-pruning phase. The function that realises it is *OptimiseTree* which is described in Section 4.3.3.

4.3.3 Function OptimiseTree

This function realises the changed post-pruning phase. First, we show it in a pseudocode (see Figure 4.13).

```
Function OptimiseTree(Tree)
  1. Create a list containing every pair of leaves. For every pair $(v_1, v_2)$,
     check if
     (a) The conditional probability is the same in both leaves, i.e.
         $P(A_0|C_{v_1}) = P(A_0|C_{v_2})$ (or the difference is statistically insignificant for the dataset).
     (b) (the continue/fail condition for "can be neighbours") Both leaves
         in a pair are defined by the same set of attributes, except one,
         where the attribute name is the same, only the value is different.
  2. $L$:=list of $(v_1, v_2)$ which satisfies 1(a) and 1(b). Sort $L$ in such way
     that all pairs of neighbours in the current tree precede pairs of non-neighbouring
     leaves (this only ensures not to be worse than the original
     post-pruning with prune only) and then by p-values as a secondary sort
     criterion. Then, go through this list (until the first success or the end
     of the list if there is no success).
     (a) If JoinSucceeded(Tree,$v_1, v_2$) Then Goto 4 End If
  3. [no pair can be combined] Exit, Return Tree
  4. Goto 1 (with a new tree, where $v_1$ and $v_2$ are pruned into one leaf now)
End Function
```

Figure 4.13: *Function* OptimiseTree

Let examine the function *OptimiseTree* in a more detailed view. The condition 1(a) says that two leaves in a pair can be neighbours in some tree. They do not need to be neighbours (this is the main improvement in a comparison with a standard post-pruning), but we cannot guarantee that we can reorganise the tree (without adding new nodes) to make them neighbours. Let's have leaves

$$L_1 := (A_1 = 0, A_2 = 1, A_{17} = 0),$$

$$L_2 := (A_1 = 1, A_2 = 1, A_{17} = 0),$$
$$L_3 := (A_1 = 0, A_2 = 1),$$
$$L_4 := (A_4 = 0, A_7 = 1, A_{14} = 0).$$

Then the pair (L_1, L_2) satisfies the condition 1(a), (L_2, L_3) does not and (L_3, L_4) also does not satisfy the condition 1(a).

We will show this procedure on the example. In Figure 4.14, we are given a tree from greedy phase. We will denote $p_i = P(A_0|C_{v_i})$ and assume that $p_1 = p_2$ and $p_i \neq p_j$ for $i \neq j$ otherwise.

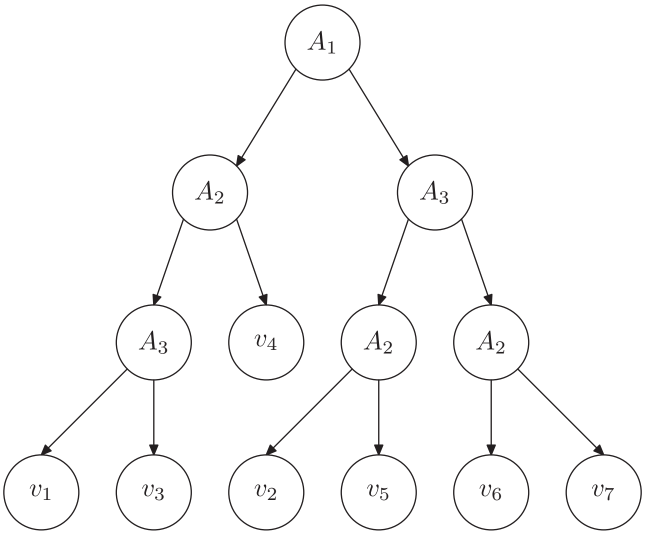


Figure 4.14: The tree given by the greedy phase.

We will show how the algorithm works in detail. First, we will present a supplementary table with a description of leaves v_i and probabilities p_i.

node	A_1	A_2	A_3	p_i
v_1	0	0	0	$\mathbf{p}_1$
v_2	1	0	0	$\mathbf{p}_1$
v_3	0	0	1	p_3
v_4	0	1	any	p_4
v_5	1	1	0	p_5
v_6	1	0	1	p_6
v_7	1	1	1	p_7

In the following table, there is a list of all pairs of nodes (v_i, v_j), where $i < j$.

Node	Node	Node 1			Node 2			continue/	equal target
1	2	A_1	A_2	A_3	A_1	A_2	A_3	fail cond.	probability
v_1	v_2	0	0	0	1	0	0	1	1
v_1	v_3	0	0	0	0	0	1	1	0
v_1	v_4	0	0	0	0	1	any	0	0
v_1	v_5	0	0	0	1	1	0	0	0
v_1	v_6	0	0	0	1	0	1	0	0
v_1	v_7	0	0	0	1	1	1	0	0
v_2	v_3	1	0	0	0	0	1	0	0
v_2	v_4	1	0	0	0	1	any	0	0
v_2	v_5	1	0	0	1	1	0	1	0
v_2	v_6	1	0	0	1	0	1	1	0
v_2	v_7	1	0	0	1	1	1	0	0
v_3	v_4	0	0	1	0	1	any	0	0
v_3	v_5	0	0	1	1	1	0	0	0
v_3	v_6	0	0	1	1	0	1	1	0
v_3	v_7	0	0	1	1	1	1	0	0
v_4	v_5	0	1	any	1	1	0	0	0
v_4	v_6	0	1	any	1	0	1	0	0
v_4	v_7	0	1	any	1	1	1	0	0
v_5	v_6	1	1	0	1	0	1	0	0
v_5	v_7	1	1	0	1	1	1	1	0
v_6	v_7	1	0	1	1	1	1	1	0

In this example, there is only one pair with equal probabilities p_1. Moreover, this pair passed the continue/fail condition (condition 1b in Figure 4.13). So we will try to use a sequence of parent-child exchange operations to move these two nodes to be neighbours. This sequence is found (if exists) by the *JoinSucceeded* function, where the example continues.

4.3.4 Function JoinSucceeded

Now, let examine the Function *JoinSucceeded* (see Figure 4.15) in a more detail. First, what is the Common Root in our example (see Figure 4.14)? When v_1 and v_2 would be neighbours, CR would be their (direct) parent. When v_1 and v_2 differs in a value of the attribute A_1 and a (great-grand-)

```
Function JoinSucceeded(Tree,$v_1, v_2$)
    1. Find the Common Root of $v_1$ and $v_2$ and put it into $CR$. The Common Root $CR$ is such vertex in the tree that both $v_1$ and $v_2$ are descendants of $CR$ (not necessarily children, they are only in the subtree with root $CR$) and $CR$ is the node in the maximal depth which satisfies the previous condition.
    2. $S$:=list of variables which defines splits on the path from $CR$ to $v_1$ (or $v_2$, lists are the same), except a variable which defines the split in $CR$.
    3. For all $s \in S$
        (a) If CanBubbleToCRPlusOne($s$,$CR$,$v_1$,$v_2$) Then
        (b) S:=S\{$s$}, PCE(CR,CR+1)
        (c) Goto 5
        (d) End If
    4. [no variable from $S$ can bubble to both children of $CR$ ] Exit, Return FALSE
    5. If $S = \emptyset$ Then Prune($CR$,$v_1$,$v_2$), Return TRUE, Exit Else Goto 3 End If
End Function
```

Figure 4.15: *Function* JoinSucceeded

parent of both named v_q has a split variable A_1, then v_q is the common root of v_1 and v_2.
Next, we want to use a set of parent-child exchanges (PCEs) (to "bubble") to move the attribute s into children of CR in order to make PCE (a parent-child exchange operation) for CR and his children. Note that we do not require all variables from S to be able to "bubble" to both children of CR, we only require one variable is able to "bubble" to both children of CR. After this and PCE, CR's depth is increased by 1 and the length of S decreased by 1. This process ends when $S = \emptyset$, or, that is, CR has two children v_1 and v_2. In this situation, we can use the standard Prune operation.

Note that we require that there exists an ordering of items in S such that the first item (attribute) can bubble into both children of CR. Then, after applying PCE, the second can bubble to both children of CR (now in a higher depth, so it is easier) and so on. It is easy to observe that when there exists such ordering and when the attribute s_j can be bubbled into both children of CR, then there exists such ordering with s_j at the first place, so we cannot lose track by this greedy algorithm. We will find this ordering in a following way: we try all attributes of S to be at the first place in this ordering. And if at least one succeeded, then we will continue with the same approach again until S is empty.

Now, we will continue with our example from the previous section. The initial situation is shown in Figure 4.14. This tree is given as the argument $Tree$ of the *JoinSucceeded* function. Remaining two arguments are nodes v_1 and v_2. In this situation, the common root is the root of the $Tree$. Definitions of nodes v_1 and v_2 are different only in the value of the variable A_1 (= the variable corresponding to the common root). The list S has two elements – A_2 and A_3. The variable A_3 is not able to be moved to both children of the common root (the function *CanBubbleToCRPlusOne* fails for A_3), but A_2 can be moved to both children of the common root and then we may use the parent-child exchange operation with the common root. The rationale will be shown in the subsection 4.3.5.
In Figure 4.16, there is one inner iteration of the *JoinSucceeded* function shown (one pass through steps 3 to 5 of the *JoinSucceeded* function). After this iteration, A_2 is removed from the list S. Now, the list S consists of the only one item A_3 and the common root of nodes v_1 and v_2 is one level deeper. The second inner-iteration (step 3) of this procedure makes the

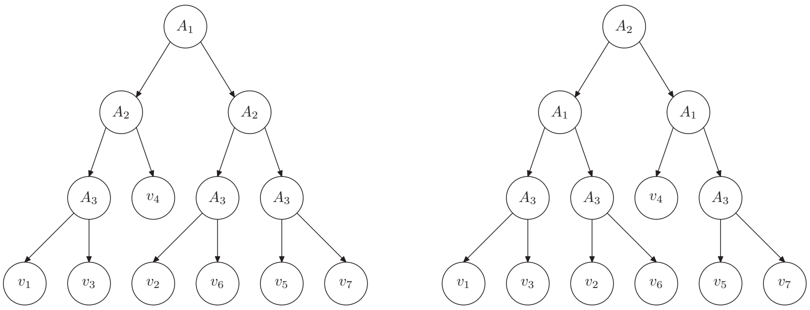


Figure 4.16: One inner-iteration of the *JoinSucceeded* function. The function *CanBubbleToRoot* converts the tree in Figure 4.14 to the tree in this figure on the left and the PCE operation finishes moving the common root to deeper position – the result is on the right.

nodes v_1 and v_2 to be neighbours by one parent-child exchange (the function CanBubbleToCRPlusOne returns immediately true without rearranging the tree). This situation is shown in Figure 4.17 on the left.

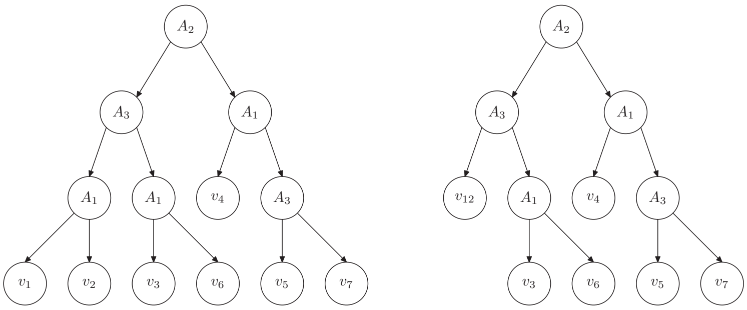


Figure 4.17: The second inner-iteration of the *JoinSucceeded* function. After step 3, nodes v_1 and v_2 are neighbours (on the left). After the step 5 of the *JoinSucceeded* function, two vertices are pruned; moreover, we have obtained the optimal tree (on the right).

The step 5 of this function prunes leaves v_1 and v_2. Because of all conditional probabilities assigned to leaves p_i are now different, there is no pair of leaves to combine in the *OptimiseTree* procedure and we have obtained the optimal tree (see figure 4.17 on the right).

4.3.5 Functions CanBubbleToCRPlusOne, CanBubbleToNode

Functions *CanBubbleToCRPlusOne* and *CanBubbleToNode* are shown in Figure 4.18.

In order to make a parent-child exchange operation with CR, we need to bubble some variable s into both children of CR. We can bubble the attribute s to the node *node* (the child of CR) only if s is in all branches from *node* to leaves – then we can find the node v where s is in the maximal depth. Note that the neighbour of v must also define a split by s (it cannot be in a lower depth - s would repeat for v, and it cannot be in a higher depth – v is in the maximal depth – and we assume that s is in all branches). Therefore, we can apply PCE operation to vertices v, the neighbour of v and their parent.

In our example, we have used this function(s) several times. In all cases except one, the variable which was required to bubble to a particular node was already in that node, so M (the set of all vertices which define a split by *varname* is a subtree defined by node *node*) had only 1 item, no "while" was executed and the function returned True. The only situation in our example where M had more than 1 item was when we need to move A_2 to both children of the root (the situation in Figure 4.14). There was the only one parent-child exchange which was done in the *CanBubbleToNode* function. Note that in a described situation, the variable A_3 was unable to be moved to the left child of the root, because it was not included in every branch from the left child to leaves. Formally, M had contained the only item with the variable A_3 (so "while" did not applied), but it was not in the root so the step 4 returned False.

4.3.6 Correctness of the Algorithm

In this subsection, we show the theoretical correctness of the algorithm *FindOptimalTree*. To be more specific, we will assume the strong faithful distribution and we will show that the optimal decision tree will be found. To do so, we will use lemmas and theorems that we have proven in previous sections.

```
Function CanBubbleToCrPlusOne($s$,$CR$,$v_1$,$v_2$)

  1. Return CanBubbleToNode($s$,Left Child Of $CR$) And
     CanBubbleToNode($s$,Right Child Of $CR$)

End Function
```

```
Function CanBubbleToNode($varname$,$node$)

  1. $M$:=$\emptyset$, For Every branch from $node$ to the leaf, find a node $m$, which
     defines a split by $varname$ and its depth (distance) from $node$, M:=M
     $\cup\{m\}$ (If there is for any branch no such node, Then Return FALSE,
     Exit)

  2. While $M$ contains more than 1 item

       (a) $M'$= items of $M$ with maximal depth in $M$

       (b) find $m_1, m_2 \in M'$, which are neighbours (have the same parent).
           If such nodes do not exist Then Return FALSE, Exit

       (c) PCE($m_1$,$m_2$,Parent $m_1$ and $m_2$)

  3. Wend

  4. If $varname$ in $node$ then Return True else Return False

End Function
```

Figure 4.18: *Function* CanBubbleToCRPlusOne, *Function* CanBubbleToNode

Theorem 21 *Let P be a strong faithful distribution. We will apply the* FindOptimalTree *to this distribution. Let T_1 be a tree which is the result of the* GrowTree *procedure and T be the tree which is a result of the* FindOptimalTree *procedure. Then,*

- *the tree T_1 represents P,*
- *the tree T is the optimal tree (which represents P).*

Proof. We know that T_1 represents P – it is Lemma 13. So we need to prove that the procedure *OptimiseTree* finds the optimal tree T from the tree T_1 which represents the distribution P. We know that P satisfies the weak faithfulness for decision trees (it follows from the fact that P satisfies the strong faithfulness), so there exists the only one optimal decision tree equivalence class.
Moreover, Corollary 10 suggests that it is sufficient to find a tree with leaves w_i with assigned different conditional probabilities $P(A_0|C_{w_i})$.

Now, we will show that for every tree T_i which represents the distribution P and is not optimal, we are able to get the tree T_{i+1} which also represents P and which has the lower number of leaves. After a finite number of steps, we will get the optimal tree.

So, let us assume that T_i is not optimal (otherwise we are done). We claim that the algorithm is able to choose the appropriate pair of nodes v_1, v_2, to find a sequence of parent child exchanges (PCEs) and to prune these nodes. Assume for a contradiction that the algorithm is not able to do so for any pair of leaves (denote this assumption as B). We will show for a contradiction that there exists at least one pair of leaves with this property.

Let $w_i, i = 1, 2, \ldots, m'$ be all leaves of T_i and C_i conditions assigned to corresponding leaves. Let $G_1, G_2, \ldots, G_m$ be groups of leaves such that for every pair of leaves (v_3, v_4), there is $P(A_0|C_3) = P(A_0|C_4)$ if and only if v_3 and v_4 belong to the same group G_j. Let G_j be a group with at least two leaves (if all groups have one leaf, all probabilities $P(A_0|C_i)$ are different and we have the optimal tree). Observe that due to the weak faithfulness (and the existence of the decision tree defining the weak faithfulness T_w), the group G_j is represented by the only one leaf in the decision tree defining weak faithfulness.

First we will show that there exist two leaves v_5, v_6 of T_i which can be neighbours in some tree (so they satisfy the "continue/fail condition"). As it follows from Lemma 14, there exists a sequence of PCEs and prunes which results in the optimal tree, so all leaves in G_j are pruned into one leaf. The first pair of the original G_j which would be pruned in this theoretical sequence is the pair v_5, v_6 of leaves which can be neighbours in some decision tree (moreover, there exists a sequence which (applied to T_i) makes these nodes to be neighbours).

Now, we have to be more specific. From the proof of Lemma 14, we know that there exists a sequence of parent-child exchange and prune operations, where in the first phase we use only parent child exchanges (PCEs) and in the second phase, we use only prunes. Let $\mathbf{T}_P$ be a tree which is a result of the first phase (we have used only PCEs, so conditions assigned to leaves remain unchanged). In this tree, there exist two neighbouring leaves (denote them v_7 and v_8) with the same conditional probability $P(A_0|C_{v_7}) = P(A_0|C_{v_8})$. So there exists a sequence of PCE operations, which makes v_7 and v_8 leaves. This property (we will call it Property Q) will be used later in this proof.
Let CR be the common root of v_7 and v_8 in the T_i. Let S be a list from the procedure *JoinSucceeded*. Assume without loss of generality that no variable from S can be moved to both children of the CR (without loss of generality – if any variable can be moved, so move it there, do PCE with the CR and continue with this until no variable from S can be moved to both children of the CR). If S is empty, then v_7 and v_8 are neighbours and can be pruned – the contradiction with the assumption B. If S is not empty, assume the subtree of T_i defined by CR and all its children of any level (denote it T_{CR}).

Now, we will examine two alternatives which can occur (in order to "move variables down") and then we mention one more possibility – whether it would help to introduce some variables above the CR into the subtree T_{CR}. So look at the alternatives. The first alternative is that at least one variable of S is in all branches of the subtree (denote it A_{all}). In this situation, we can move A_{all} to the root (we find nodes with a split by A_{all} in a maximal depth and which are neighbours and we apply PCE with their parent recursively until A_{all} is in the root of T_{CR} – repeatedly used an easy idea which was a core of the proof of Lemma 17). That is the contradiction with the assumption that no variable of S can be moved into the root.

The second alternative is that no variable of S is in all branches of T_{CR}. So look at the tree $\mathbf{T}_P$, which can be obtained from T_i by a sequence of PCE operations (Property Q). Now, we will show for the contradiction that such sequence does not exist. We look what this sequence does with CR. To make v_7 and v_8 neighbours, we need to have at least one variable from S to be in all branches (it would be moved above the CR in our theoretical sequence, so it must be in all branches). So this is a contradiction with assumption B.

Now we examine one more chance how could we make v_7 and v_8 neighbours (and we will show that it does not work) – we will try to introduce some new variables to the subtree T_{CR} (to the list S) from the remaining parts of the tree (the set of variables to be shifted up would be S', where $S \subset S'$). But, in a new subtree of CR, we have introduced only variables which comes originally from upper levels that CR, so no new splits by any element of S is introduced under CR (due to non-repeating variables on every path to the leaf). The only possibility for this solution to work is that we can move some element of the original S above the CR But we can move just newly introduced variables, i.e. variables from $S' \backslash S$ (after applying discussion in alternative 1), because there still exists for every element of $s \in S$ at least one leaf which does not have s in its condition (PCE does not change condition assigned to leaves). So such sequence cannot exist and this is a desired contradiction with assumption B.

□

Chapter 5

Tests of Proposed Algorithm on Data

5.1 Reconstructing Known Tree From Data

This section describes experiments with the proposed algorithm. We have tried to generate data records randomly from several trees (all possible tree structures from the smallest ones – these ones are included as subtrees in larger trees) and we have compared results – which trees were obtained by tested algorithms from such generated data. There were compared two algorithms – both started with a decision tree learned by CART (possibly overfitted). The first one uses only the post pruning (denoted by "CART-PO" – prune only) and the second one uses prune and parent child exchange operations as proposed in previous sections (denoted by "CART-PP").

Note that properties of the algorithm are proven for a distribution only. For a distribution, two neighbouring leaves can be pruned if their proportions of the target variable are the same. For a dataset, there is a statistical test used. This test is based on hypothesis that both proportions are the same. Due to proven properties for a distribution, we have tested the data which are big enough, or properties of the algorithm where number of observations tends to infinity respectively. The main question (to which should these tests reply) is whether we are able to achieve a good reconstruction at least in the situation where data are large enough.

We have tested several tree structures and categorised them into "patterns". The pattern means here a structure of a probability distribution (what will equal to what – we will generate a fictive full tree where some leaf probabilities p_i equal to some other ones). The original tree structure, the probability distribution and results of the comparison of these two algorithms (the first – the original post-pruning algorithm and the second – advanced by parent-child exchanges) will be shown for each pattern.

There were six tree structures (in ten patterns) tested. Patterns (tree structures) was selected in a following way: construct six smallest decision trees (logically same trees are treated as one type – for example, when the tree has only two splits – one split in the root and the second split in the left leaf, we treat it as the same as it would have the second split in the right leaf; changing variable names (ordering) is also ignored).
There are several reasons for choosing the data with few attributes only. First, large datasets include this data as subtrees. Second, having smaller number of attributes with fixed dataset size, there is a bigger chance on pruning two (originally not neighbouring) leaves. Note that this may look as finding trees where our algorithm works better, but honestly we want to know whether our algorithm can achieve significantly better results at least for data which are very large. This allows us to test our algorithm on relatively small datasets.

For every pattern, we introduce how was this pattern generated (a source distribution description will be provided) and brief results which shows numbers of correctly reconstructed trees.
The first subsection shows trees where results of both algorithms should be similar (Pattern A, Pattern B) – because the full tree is the optimal one for them (so they cannot be improved). In the second subsection (for patterns 1 to 4), results should be different.
Tree structures and patterns are shown in Figure 5.1. For patterns 1 to 4, we have tried two alternatives (for example, pattern 1 and 1b), which differs only in probability tables – for leaves, where should be the same proportion of the target variable in a full tree (i.e. leaves of the full tree, which are coalesced in the optimal tree), "non-b" have also the same expected size of this leaf. Note that for every combination of the pattern, the algorithm and the dataset size, there were generated 8 samples of data.

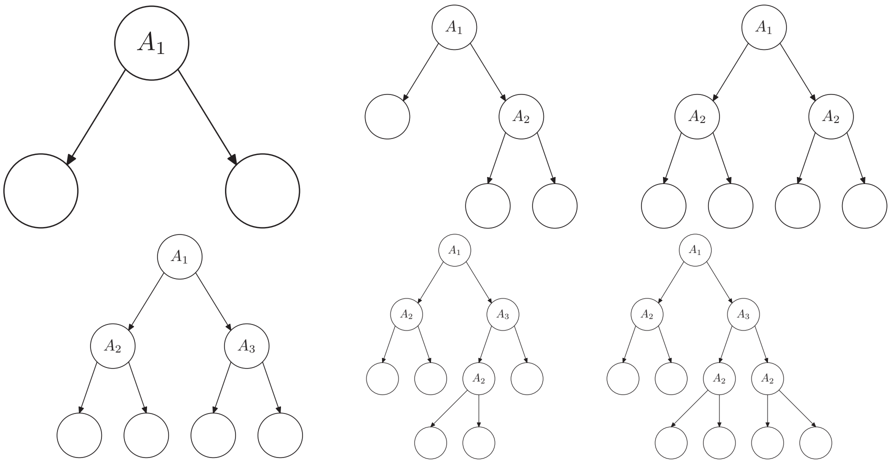


Figure 5.1: Six tree structures, from which were tested ten patterns: Pattern A, Pattern 1+1b, Pattern B, Pattern 2+2b, Pattern 3+3b and Pattern 4+4b

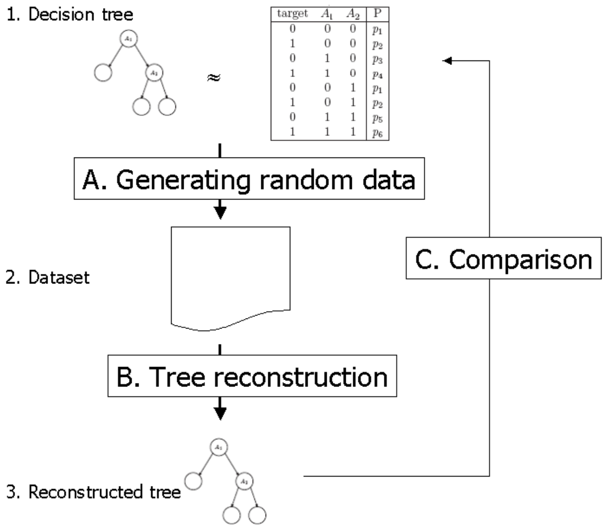
In tests, we have tried more dataset sizes, from 300 to 30,000 observations in a dataset. We have also used test sets (with size about 1/3 of the original dataset and we have evaluated the prediction accuracy. This accuracy was different only exceptionally (and not biased to prefer either algorithm), therefore the comparison of prediction accuracies is not shown here. Due to practical application purposes, we will closely focus on bigger dataset sizes.

5.1.1 Test design

We will look at the test design in a more detail. We have tried to test, what is the quality of the reconstruction of the given tree by the tested algorithm. First, we have tried to generate the data from the known tree, then we have tried to execute the algorithm and then compare the result of the algorithm with the original tree. This test was executed several times for both tested algorithms.

To be more specific, for every individual pattern and for a given algorithm, the test schema in Figure 5.2 was realised.

We will describe individual parts and steps from this figure.



target	A_1	A_2	P
0	0	0	p_1
1	0	0	p_2
0	1	0	p_3
1	1	0	p_4
0	0	1	p_1
1	0	1	p_2
0	1	1	p_5
1	1	1	p_6

Figure 5.2: Individual test for a given algorithm and for a given pattern.

1. **Decision Tree** – a given decision tree. The tree may be derived from a given pattern. The way to obtain a decision tree (or a probability table) is the following. The pattern is a template of a probability table. Probabilities are given by virtual numbers p_i. We will generate p_is randomly. The sum of p_i is to be 1. At first, we will generate B_i as a random number between 1 and 100. Next, p_i are B_i adjusted in a way that sum of p_is is 1, so we define $p_i = \frac{B_i}{\sum B_i}$.

A. **Generating random data** – we have generated records with respect to the probability table. So cumulative probabilities were counted and random number r between 0 and 1 (uniformly distributed) was computed. The first record in the probability table which has the cumulative probability greater than or equal to r was selected into the dataset. This gives us one record into the generated dataset. So we can repeat it how many times we need. Note that generating one record is nothing else than generating records with a given probability of occurrence.

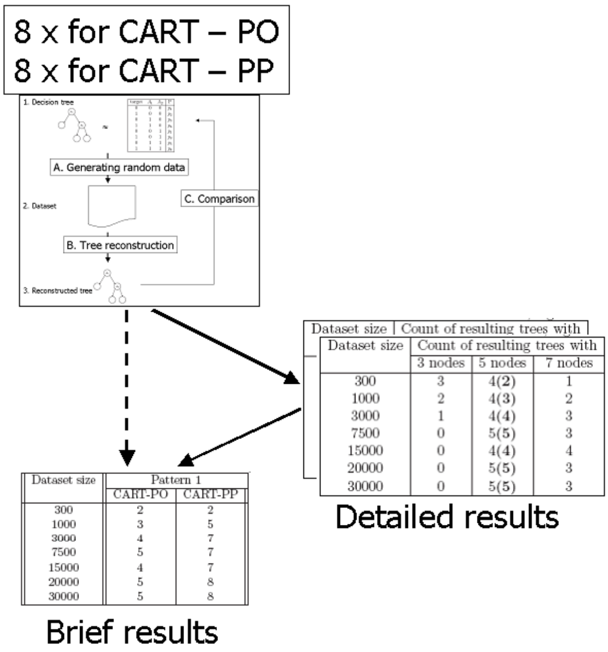
2. **Dataset** – set of records from Step A.

B. **Tree reconstruction** – the tested algorithm was executed. The result of this algorithm will be called the reconstructed tree.

3. Reconstructed tree – the result of the tested algorithm

C. Comparison – the reconstructed algorithm was compared with the original tree. We have checked the number of leaves and variables in inner nodes. We define that the reconstruction of the tree is good, if both trees are one node trees or roots of both trees have the same variable A_R and their left subtrees (i.e. defined by $A_R = 0$) are the same and right subtrees are the same. Note that we have checked also numbers of nodes of reconstructed trees. The good reconstruction requires that both trees have the same number of nodes. Conversely, the same number of nodes does not necessarily ensure that reconstruction is good.

The entire test was executed eight times for both tested algorithms (see Figure 5.3).

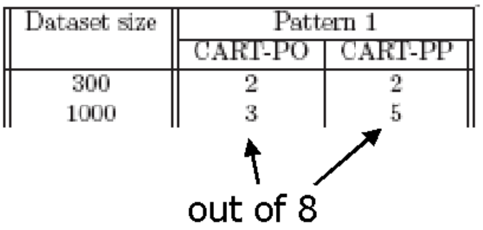


Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	4(2)	1
1000	2	4(3)	2
3000	1	4(4)	3
7500	0	5(5)	3
15000	0	4(4)	4
20000	0	5(5)	3
30000	0	5(5)	3

Dataset size	Pattern 1	
	CART-PO	CART-PP
300	2	2
1000	3	5
3000	4	7
7500	5	7
15000	4	7
20000	5	8
30000	5	8

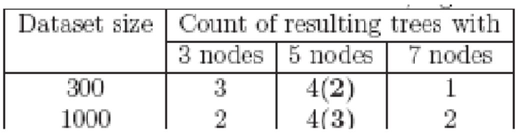
Figure 5.3: The complete test for a given pattern.

The number of good reconstructions for each algorithm is the main result, therefore this number is included in both brief and detailed results.



Dataset size	Pattern 1	
	CART-PO	CART-PP
300	2	2
1000	3	5

Figure 5.4: Brief results. We may see the number of good reconstruction for a given pattern and every tested dataset size. The maximal number of good reconstructions is 8.



Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	4(2)	1
1000	2	4(3)	2

Figure 5.5: Detailed results. We may see sizes of reconstructed trees. In this example, the original tree has 5 nodes, so the column ”5 nodes” has one extra number in parentheses – this is the number of good reconstructions.

Brief results are shown in Figure 5.4. Detailed results include sizes of reconstructed trees. Detailed results are explained in Figure 5.5. These results are not shown in this chapter, they are in Appendix C (they are referenced from this chapter). Detailed results are shown for a given pattern and a given algorithm. Two tables of detailed results correspond to one table of brief results.

For patterns 1 to 4, we have two alternatives. We call them ”b” and ”non-b” (e.g. we have patterns 1 and 1b, the pattern 1 is called ”non-b” and the pattern 1 is called ”b”). The difference is following: we assume the full tree T_F corresponding to a given probability table (some leaves may have same probabilities of target). In the ”non-b” alternative, we assume that leaves in T_F which are assigned the same probability have also the same proportion (the expected number of records). The ”b” alternative does not have this requirement. We may study the behaviour of the algorithm under (”non-b” alternative) and without (”b” alternative) the assumption that the pair of leaves which is to be pruned has the similar number of records in both leaves.

We should mention also some details about implementation of the growing a tree and the post-pruning. We have created and used an implementation which was made only for purposes of evaluation of this algorithm’s properties. To grow a tree, external software for decision trees was used. We have tried to improve post-pruning so we have set up parameters for growing in order to obtain (very) overfitted tree (for example, the minimal size for a decision tree node was set up to 5 records). Then, the resulting tree was transferred into our implementation and two post-pruning alternatives were tested (every alternative was executed on the grown tree obtained from the external software, not on the result of the second algorithm). The first alternative was pruning only and the second one was pruning and parent-child exchanges (our algorithm).

We have tried to reconstruct full trees as well as trees which are not full. For full trees, we expected to achieve a 100% of good reconstructions. These results can also show the effect of the situation when we have only a very small number of records in dataset (errors caused by growing – the tree growing was stopped prematurely and error caused by pruning – two leaves were pruned due to small number of records and small difference in the target variable distribution).

5.1.2 Reconstruction of Full Trees

Pattern A

This pattern is the easiest design of the decision tree. The tree structure is shown in Figure 5.6 and the distribution is shown in Table 5.1.

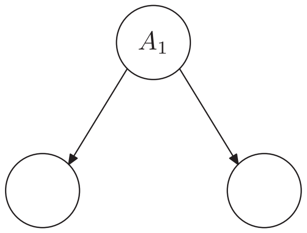


Figure 5.6: Pattern A

Table 5.1: Pattern A – probabilities, from which the data was generated

target	A_1	P
0	0	p_1
1	0	p_2
0	1	p_3
1	1	p_4

We have generated the data 8 times for each Dataset size and we have evaluated the number of nodes in target trees (using CART-PO and CART-PP algorithm), the classification accuracy and the equivalence to original tree. Note that the classification accuracy was different only exceptionally (in most cases it was identical, when the difference appeared, sometimes it was a little bit better and sometimes a little bit worse for the CART-PP algorithm), so the classification accuracy is not shown in these results. The number of equivalent trees (the number of correct reconstructions) is shown in individual tables. The comparison the CART-PO algorithm with the CART-PP algorithm for the data generated from "Pattern A" is shown in Tables C.1 and C.2 in the Appendix C. The brief version of results is shown in Table 5.2.

Table 5.2: Comparison of CART-PP and CART-PO algorithm, pattern A

Dataset size	Pattern A	
	CART-PO	CART-PP
300	7	7
1000	7	7
3000	8	8
7500	8	8
15000	8	8
20000	8	8
30000	8	8

The reconstructed tree should have 3 nodes, in most cases it has 3 nodes for both algorithms.

Pattern B

The pattern B is the full tree with 2 variables, so the reconstructed tree should have 7 nodes (4 leaves plus 3 non-leaf nodes) – see Figure 5.7.

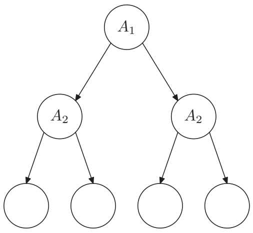


Figure 5.7: Pattern B

The probability distribution, from which the data was generated, is shown in Table 5.3.
Results for "Pattern B" are shown in Tables C.3 and C.4 in the Appendix C. Brief results are shown in Table 5.4.
We may see that both algorithms give similar results.

Table 5.3: Pattern B – probabilities, from which the data was generated

target	A_1	A_2	P
0	0	0	p_1
1	0	0	p_2
0	1	0	p_3
1	1	0	p_4
0	0	1	p_5
1	0	1	p_6
0	1	1	p_7
1	1	1	p_8

Table 5.4: Comparison of CART-PP and CART-PO algorithm, pattern B

Dataset size	Pattern B	
	CART-PO	CART-PP
300	5	3
1000	6	5
3000	8	8
7500	8	8
15000	8	8
20000	8	8
30000	8	8

5.1.3 Reconstruction of Trees which are not full

In this subsection, we will test three patterns (patterns 1 to 3) in two alternatives (”non-b” and ”b”).

Pattern 1

Pattern 1 has a very easy structure, which is shown in Figure 5.8.

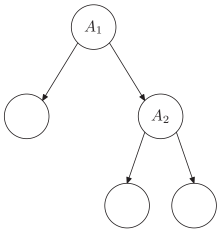


Figure 5.8: Pattern 1

The probability distribution tested is shown in Table 5.5.

Table 5.5: Pattern 1 – probabilities, from which the data was generated

target	A_1	A_2	P
0	0	0	p_1
1	0	0	p_2
0	1	0	p_3
1	1	0	p_4
0	0	1	p_1
1	0	1	p_2
0	1	1	p_5
1	1	1	p_6

Now we look at results of both algorithms. Brief results are shown in Table 5.6. Full results are shown in Tables C.5 and C.6 in Appendix C.
From these results we may see that the CART-PP algorithm finds in most cases (for datasets which are big enough) the optimal tree, but the CART-PO algorithm (the classical CART algorithm) finds a tree with a bad structure,

Table 5.6: Comparison of CART-PP and CART-PO algorithm, pattern 1

Dataset size	Pattern 1	
	CART-PO	CART-PP
300	2	2
1000	3	5
3000	4	7
7500	5	7
15000	4	7
20000	5	8
30000	5	8

which classifies the same, but in a more complicated way. Moreover, the real structure of data is in many cases hidden for the CART-PO algorithm.

For the CART-PO algorithm, finding a correct/incorrect (more complicated) tree depends on "guessing" the correct variable for the root in the first step of tree learning. If the CART-PO chooses a wrong variable (the other than in the optimal tree) for the root in a learning phase and a variable on the next level is significant, the optimal tree can be never found. The CART-PP is able to fix this problem in the advanced post-pruning phase by parent-child exchange operations. So correct variable may become the new root and an additional prune may be available and finally the optimal tree is found.

Pattern 1b

The pattern 1b is only an adjustment of the pattern 1. The only difference is a way the probabilities are generated. The probability table is shown in Table 5.7.
The results for the CART-PO algorithm and the CART-PP algorithm are shown in Tables C.7 and C.8 in the Appendix C. Brief results are shown in Table 5.8.

We may see again that the CART-PP algorithm finds a real structure of the tree in 100% of tested cases (for the datasets which are big enough), but the CART (CART-PO) algorithm does not in approx. 38%.

Table 5.7: Pattern 1b – probabilities, from which the data was generated

target	A_1	A_2	P
0	0	0	p_1
1	0	0	p_2
0	1	0	p_3
1	1	0	p_4
0	0	1	p_5
1	0	1	$\frac{p_2}{p_1}p_5$
0	1	1	p_6
1	1	1	p_7

Table 5.8: Comparison of CART-PP and CART-PO algorithm, pattern 1b

Dataset size	Pattern 1b	
	CART-PO	CART-PP
300	3	3
1000	5	5
3000	5	8
7500	5	8
15000	5	8
20000	5	8
30000	5	8

Pattern 2

This pattern is typical for a real world business data. Two variables are significant, but for one group of clients is significant the first variable and for the second group the second variable is significant. This difference is usually caused by business rules (for example, price lists for two groups) and it appears in many forms in a real world data. The structure of the tree for this pattern is shown in Figure 5.9 and the probability table, from which data were generated, is shown in Table 5.9.

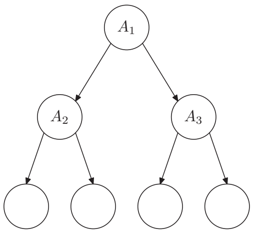


Figure 5.9: Pattern 2

The results are shown in Tables C.9 and C.10 in the Appendix C. Brief results are shown in Table 5.10.
We can see that the algorithm CART-PP finds the optimal tree in more than 90% cases, but the algorithm CART-PO finds the optimal tree in only approximately 50% cases.

Pattern 2b

The pattern 2b is similar to the pattern 2, the only difference is the probability table from which the data are generated. The probability table for the pattern 2b is shown in Tables C.11 and C.12 in the Appendix C. Brief results are shown in Table 5.11.
Results for the pattern 2b are shown in Table 5.12.
From results, we can see that the algorithm CART-PO generates more inoptimal trees (with more than 7 nodes) than the algorithm CART-PP do (for large datasets, the CART-PP algorithm generates inoptimal tree in approximately 25% of cases, but the CART-PO algorithm in almost 50% of cases).

Table 5.9: Pattern 2 – probabilities, from which the data was generated

target	A_1	A_2	A_3	P
0	0	0	0	p_1
1	0	0	0	p_2
0	1	0	0	p_3
1	1	0	0	p_4
0	0	1	0	p_5
1	0	1	0	p_6
0	1	1	0	p_3
1	1	1	0	p_4
0	0	0	1	p_1
1	0	0	1	p_2
0	1	0	1	p_7
1	1	0	1	p_8
0	0	1	1	p_5
1	0	1	1	p_6
0	1	1	1	p_7
1	1	1	1	p_8

Table 5.10: Comparison of CART-PP and CART-PO algorithm, pattern 2

Dataset size	Pattern 2	
	CART-PO	CART-PP
300	3	4
1000	4	4
3000	4	7
7500	4	7
15000	4	8
20000	4	8
30000	4	7

Table 5.11: Pattern 2b – probabilities, from which the data was generated

target	A_1	A_2	A_3	P
0	0	0	0	p_1
1	0	0	0	p_2
0	1	0	0	p_3
1	1	0	0	p_4
0	0	1	0	p_5
1	0	1	0	p_6
0	1	1	0	p_7
1	1	1	0	$\frac{p_4}{p_3}p_7$
0	0	0	1	p_8
1	0	0	1	$\frac{p_2}{p_1}p_8$
0	1	0	1	p_9
1	1	0	1	p_{10}
0	0	1	1	p_{11}
1	0	1	1	$\frac{p_6}{p_5}p_{11}$
0	1	1	1	p_{12}
1	1	1	1	$\frac{p_{10}}{p_9}p_{12}$

Table 5.12: Comparison of CART-PP and CART-PO algorithm, pattern 2b

Dataset size	Pattern 2b	
	CART-PO	CART-PP
300	3	2
1000	4	2
3000	2	5
7500	4	5
15000	4	5
20000	5	7
30000	5	7

Pattern 3

The pattern 3 is a next tree structure which will be tested. Again, we will use Pattern3 and Pattern3b alternatives which differs only in probability tables. The tree structure for both patterns is shown in Figure 5.10.

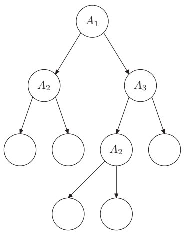


Figure 5.10: Pattern 3

Probability tables will be no longer shown, their structure can be easily derived from tables for patterns 1 to 2b. Results for the pattern 3 are shown in Tables C.13 and C.14 in the Appendix C. Brief results are shown in Table 5.13.

Table 5.13: Comparison of CART-PP and CART-PO algorithm, pattern 3

Dataset size	Pattern 3	
	CART-PO	CART-PP
300	0	3
1000	1	4
3000	2	6
7500	1	8
15000	1	8
20000	1	7
30000	1	6

In this more complicated pattern, we can clearly see that the algorithm CART-PP finds in most cases the optimal tree (for tested data, which are big enoughm in approximately 90% of cases), but the algorithm CART-PO finds in many cases an inoptimal tree (for a big data, the optimal tree was found in only approx. 15% of cases).

Pattern 3b

Results for the pattern 3b are shown in Tables C.15 and C.16 in the Appendix C. Brief results are shown in Table 5.14.

Table 5.14: Comparison of CART-PP and CART-PO algorithm, pattern 3b

Dataset size	Pattern 3b	
	CART-PO	CART-PP
300	1	1
1000	2	5
3000	1	3
7500	2	7
15000	2	7
20000	1	7
30000	2	7

We can see again, that the algorithm CART-PP finds in many cases the optimal tree (almost 90% for a big data), but the algorithm CART-PO tends to find an inoptimal tree (the optimal tree is found in less than 25% cases for a big data).

Pattern 3 and 3b show a great improvement in finding the optimal tree when we use the CART-PP algorithm (due to our extension to post-pruning phase).

Pattern 4

The pattern 4 is the last tree structure, which we will examinated here. The tree structure, which is common for patterns 4 and 4b, is shown in Figure 5.11.

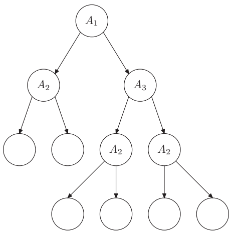


Figure 5.11: Pattern 4

Result of the reconstruction for the pattern 4 is shown in Tables C.17 and C.18 in the Appendix C. Brief results are shown in Table 5.15.

Table 5.15: Comparison of CART-PP and CART-PO algorithm, pattern 4

Dataset size	Pattern 4	
	CART-PO	CART-PP
300	0	0
1000	2	1
3000	2	2
7500	3	5
15000	3	5
20000	3	5
30000	3	6

We can see again that the algorithm CART-PP finds the optimal tree for much more samples than the algorithm CART-PO do. For CART-PO, in less than 40% cases, the optimal tree is found (usually the reason is that the number of nodes is bigger than the number of nodes in the original tree). The algorithm CART-PP has more than 50% of optimal trees for a big data.

Pattern 4b

Again, the only difference of the pattern 4 and the pattern 4b is the probability table from which data are generated. Results for the pattern 4b are shown in Tables C.19 and C.20 in the Appendix C. Brief results are shown in Table 5.16.

Table 5.16: Comparison of CART-PP and CART-PO algorithm, patterns 4 and 4b

Dataset size	Pattern 4b	
	CART-PO	CART-PP
300	0	0
1000	3	3
3000	2	3
7500	4	5
15000	4	4
20000	4	6
30000	4	6

Once again, the algorithm CART-PP shows better reconstruction of the optimal tree and the true structure of the data. The algorithm CART-PO finds the optimal tree in approx. 50% of cases, the algorithm CART-PP in approximately 65% of cases (both for a big data).

5.1.4 Results summary

We have tried to reconstruct 6 structures of trees, which should cover all combinations of small trees with a small number of variables (and which are not only the other arrangement of previous trees, for example, the situation, where we have only two splits, one in the root and the second one in it child, we assume that design where the split is in the left child and the design where the split is in the right child to be the same).

In this experiment, we estimated that two designs (patterns) would have same results (the success of finding optimal tree) – in situations where the optimal tree was a full tree, and for four patterns the results could be improved. Results show that when using the CART-PP algorithm instead of the CART-PO algorithm, we can find the optimal tree (and the true structure of the

data) in significantly more cases. For the CART-PO algorithm, in some cases the randomness influences the chance of finding the optimal decision tree.
The main problem of the original CART algorithm (CART-PO) is when it chooses a wrong attribute into the node (wrong=in the optimal tree, the other attribute is used in this context). Then it cannot be repaired except the situation that the entire subtree of this node is pruned. Otherwise, post-pruning never fix this problem. The CART-PP algorithm is able to rearrange attributes in nodes, so we can find the optimal tree in much more cases. The summary of results is provided in Figures 5.12 and 5.13.
Note that we can estimate how can the data size influence the number of correctly reconstructed trees (due to the insignificance in split/prune tests for small data) from full tree patterns (pattern A and pattern B). Results show that data are big enough for sample size 3,000 and more.

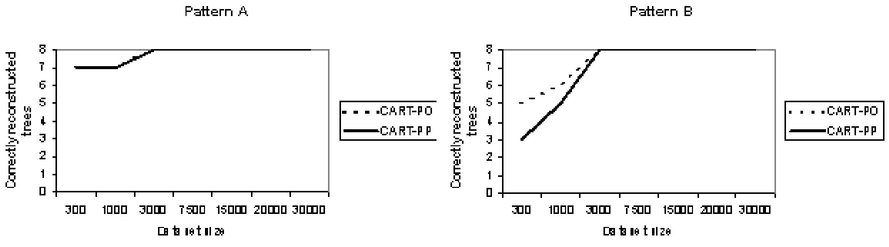


Figure 5.12: The comparison of the CART-PP against the CART-PO algorithm (full trees – patterns A and B).

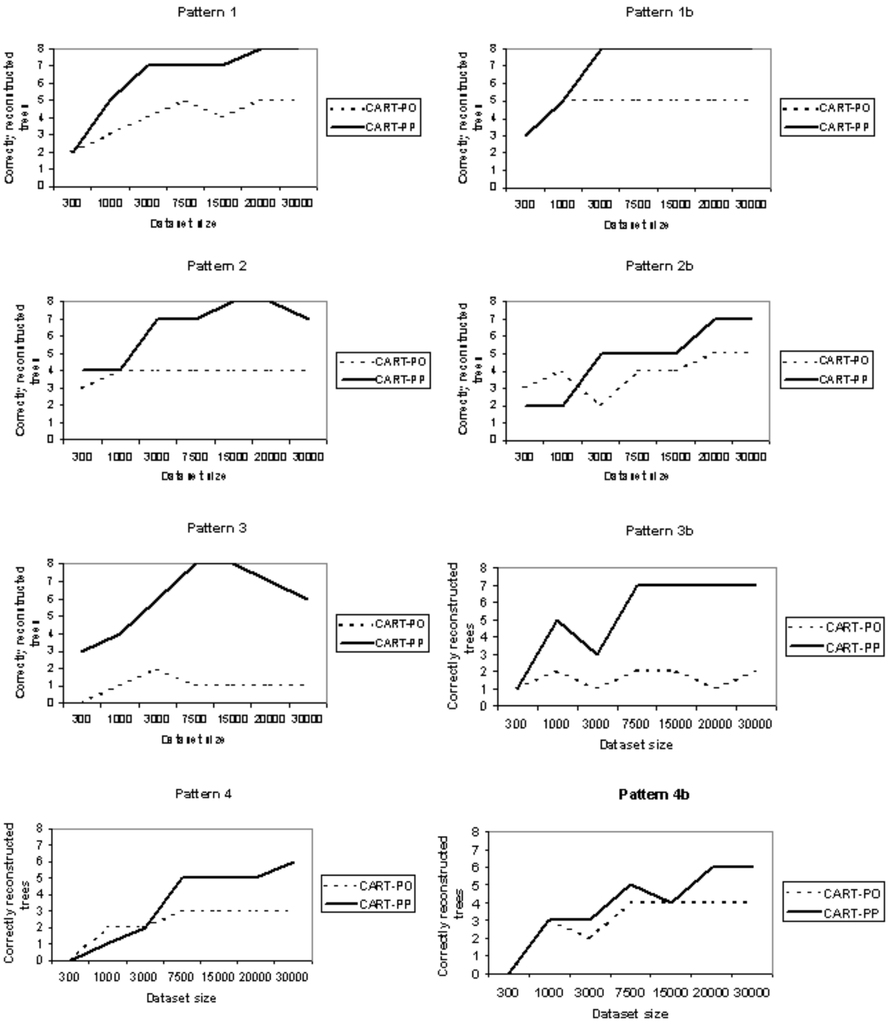


Figure 5.13: The comparison of the CART-PP against the CART-PO algorithm (trees which are not full – patterns 1 to 4b).

5.2 Comparison On Real Data

The proposed algorithm was tested on a real business data – a MTPL claim data (the simplified version with 3 binary predictors and one binary target only). Attributes was

- claim occurred (target)
- personal car/truck
- measure attribute 1 (based on vehicle weight)
- measure attribute 2 (based on engine size)

Analysed data were created by a particular sampling. The data in an aggregated form are shown in Table 5.17. More details about this data and the preparation method cannot be provided.

Table 5.17: Insurance data in the aggregated form

Is truck?	High weight	High engine size	Is claim?	Count(*)
0	0	0	0	977
0	0	0	1	786
0	0	1	0	17
0	0	1	1	25
0	1	0	0	256
0	1	0	1	284
0	1	1	0	301
0	1	1	1	456
1	0	0	0	11
1	0	0	1	28
1	0	1	1	1
1	1	0	0	1
1	1	0	1	5
1	1	1	0	47
1	1	1	1	137

The algorithm CART-PO have found the tree with 5 leaves, the algorithm CART-PP have found the tree with 4 leaves with identical prediction accuracy on the test set. Originally, the attribute measure 2 (based on engine size) was in the root. Algorithm CART-PP proposed to the root the attribute, which means personal car/truck, which we had expected and what is generally expected and used by business. Results for this real data experiment are summarized in Table 5.18 and resulting trees are shown in the Figure 5.14.

Table 5.18: Results on real data. Both trees have the same prediction accuracy on the test set.

Algorithm	Number of leaves	Variable in the root
CART (CART-PO)	5	High_engine_size
CART-PP	4	Is_truck

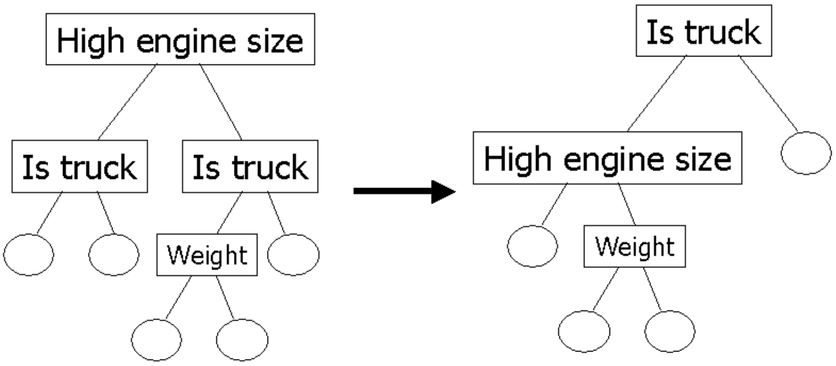


Figure 5.14: Resulting trees for the insurance data, the result for the CART-PO algorithm is on the left, the result for the CART-PP algorithm is on the right.

Note that we have tried this algorithm on several parts of UCI ML Datasets. Due to binary data requirement, analysed data (predictors) was prepared into binary form. For many predictors, rarely any non-neighbouring pair could be joined. And when so, the resulting tree was the same as when we have used pruning only. When we have tried to use only subset of predictors, it was observed that some more post-pruning operations could improve results –

sometimes the ”expansion” (the opposite operation for the prune operation – it increases the number of leaves) could be helpful. But using the expansion, we need to be careful – we can get in the loop (expansion – prune cycle or some more complicated cycle) and it is not guaranteed that we will get the smaller tree than the original tree before post-pruning. So this issue is a part of the future work.

Chapter 6

Conclusions and Future Work

6.1 Results achieved

We have introduced the extension into the post-pruning phase of the decision tree algorithm. We have studied this extension for a distributions and for real data. It was shown that the algorithm finds the optimal decision tree for faithful distributions. This algorithm is based on a theoretical basis, we have proved that it works correctly for the distribution under the assumption of faithfulness, in contrast with traditional methods, where the optimal result is not guaranteed even for the distribution. The proposed algorithm gives magnificently better results when reconstructing the known decision tree from data (where can be easily measured whether the result is correct or not). The algorithm has also succeeded on a real business data, where the better tree with the identical prediction accuracy was found. Moreover, the tree found by the proposed algorithm better reflects the business meaning of results. The algorithm provided is polynomial in a number of leaves of the tree which is the result of the growing phase and it is the improvement in comparison with the exponential complexity of the trivial exhaustive search algorithm. Note that the algorithm is not necessarily polynomial in the number of variables. In contrast to surprising results on a real business data, the algorithm in proposed form was also tested on some datasets from the UCI-ML Data Repository (selected attributes were prepared to their binary form) and both CART-PO and CART-PP result in the same tree. Nevertheless, results on the simulated data and the real business data have shown the improvement.

6.2 Future work

Having completed the core skeleton of the algorithm and the implementation to be able test this algorithm, the following items may extend this work

- Fine-tune the algorithm to achieve even better results, concentrating on a real data.
- Detailed study of the algorithm complexity and its possible improvements.
- Extend the algorithm to non-binary discrete variables and continuous variables in a more sophisticated way (not only recoding them into binary variables).
- Develop a routine applicable implementation of the proposed algorithm.

Bibliography

[1] Berry, M.J., Linoff G.: Data Mining Techniques, Wiley, 1997.

[2] Breiman et al.: Classification and Regression Trees, Woodsworth International Group, 1984.

[3] Carbonell, J., Michalski, R.S., Mitchell, T.: An Overview of Machine Learning, in Machine Learning: An Artificial Intelligence Approach, TIOGA Publishing, 1983.

[4] CRISP-DM: Process Model. Retrieved September 20, 2005 from `http://www.crisp-dm.org/Process/index.htm`.

[5] Dunham, M.H.: Data Mining – Introductory and Advanced Topics, Prentice Hall, 2003.

[6] Esmeir, S., Markovitch, S.: Interruptible Anytime Algorithms for Iterative Improvement of Decison Trees, Proceedings of the 1st international workshop on Utility-based data mining, UBDM, 2005, pp. 78–85.

[7] Esmeir, S., Markovitch, S.: Lookahead-based algorithms for anytime induction of decision trees, in Proceedings of the 25th international conference on machine learning, 2004.

[8] Gentle, J.E., Härdle, W., Mori, Y: Handbook of Computational Statistics, Springer, 2004.

[9] Goldberg, P., Jerrum, M.: Bounding the Vapnik-Chervonenkis Dimension of Concept Classes Parametrised by Real Numbers, Proceedings of the sixth annual conference on Computational learning theory, 1993, pp. 361–369.

[10] Goodrich, M.T., Mirelli, V., Orletsky, M., Salowe, J.: Decision Tree Construction in Fixed Dimensions: Being Global is Hard but Local Greed is Good, Technical Report, Johns Hopkins University, Computer Science Dept., 1995.

[11] Hyafil, Rivest: Constructing Optimal Binary Decision Trees is NP-Complete, Proc. Letters, 1976, pp. 15–17.

[12] Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., Wirth, R: CRISP-DM 1.0, 2000, `www.crisp-dm.org`.

[13] Chickering, M.: Learning Equivalence Classes of Bayesian-Network Structures, Journal of Machine Learning Research, 2002, pp. 445–498.

[14] Chickering, M., Meek, Ch.: Finding Optimal Bayesian Networks, In Proceedings of Eighteenth Conference on Uncertainty in Artificial Intelligence, 2002, pp. 94–102.

[15] Cristianini, N.,Shawe-Taylor, J.: An Introduction to Support Vector Machines (and other kernel-based learning methods), Cambridge University Press 2000.

[16] Imhoff, C. et al.: Mastering Data Warehouse Design, John Wiley & Sons, 2003.

[17] Kass, G.V.: An exploratory technique for investigating large quantities of categorical data, Applied Statistics, 1980, pp. 119-127.

[18] Kimbal, R. et al.: The Data Warehouse Lifecycle Toolkit, John Wiley & Sons, 1998.

[19] Masa, P., Kocka, T.: Finding Optimal Decision Trees, accepted for IIPWM 2006.

[20] Matousek, J., Nesetril, J.: Invitation to Discrete Mathematics, Oxford University Press, 1998.

[21] Mitchell, T.: Machine Learning, McGraw Hill, 1997.

[22] Murthy, K.: On Growing Better Decision Trees from Data, PhD Thesis, 1997.

[23] Parr-Rud, O.: Data Mining Cookbook, Wiley, 2000.

[24] Pfahringer, B.: Inducing Small and Accurate Decision Trees, Technical Report, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, 1998.

[25] Quinlan, J.R. : C4.5: Programs For Machine Learning, Morgan Kaufmann Publishers, 1993.

[26] Quinlan, J.R.: Induction of Decision Trees, Machine Learning, 1986, pp. 81–106.

[27] Quinlan, J. R.: Simplifying decision trees. International Journal of Man-Machine Studies, 1987, pp. 221–234.

[28] Ragavan, H., Rendell, L.A.: Lookahead Feature Construction for Learning Hard Concepts, ICML, 1993, pp. 252–259.

[29] Utgoff, P., Berkman, N.C. , and Clouse J.A.: Decision tree induction based on efficient tree restructuring, Machine Learning, 1997, pp. 5–44.

[30] Utgoff, P.E.: Decision Tree Induction Based on Efficient Tree Restructuring, Technical Report 95-18, University of Massachusetts, Department of Computer Science, Amherst, 1996.

[31] Vapnik, V. N., Chervonenkis, A. Y.: On the uniform convergence of relative frequencies of events to their probabilities, Theory Probability and its applications, 1971, pp. 264–280.

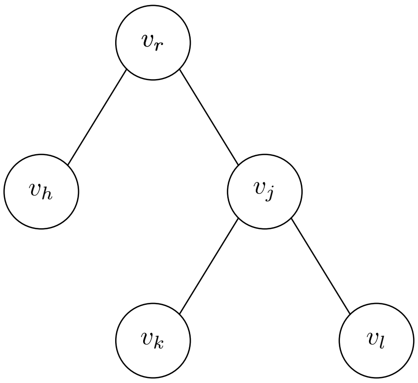
[32] Vapnik, V.N.: Statistical Learning Theory, John Wiley & Sons, 1998.

[33] VC dimension, Wikipedia, The Free Encyclopedia. Retrieved November 19, 2005 from `http://en.wikipedia.org/wiki/VC_dimension`.

Appendix A

Notation Quick Reference

A.1 Tree and Graph Theory



- v_r is a *root node*

- the two vertices v_h and v_j are *neighbour nodes*; there is an another pair of neighbour nodes in our tree

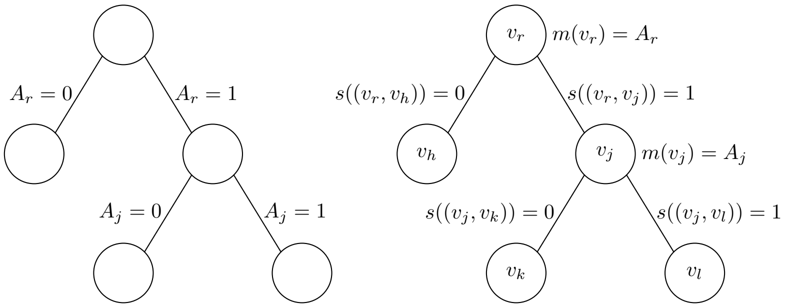
- v_h, v_k, v_l are *leaf nodes*

- v_r is node in depth 0, v_h, v_j are nodes in depth 1, v_k, v_l are nodes in depth 2; *depth* is a distance of vertex from root node

- v_h and v_j are *children* of v_r; v_k, v_l are *children in depth 2* of v_r; v_r is a *parent* of v_h or v_j

A.2 Decision Tree Formal Description

$\mathbf{T} = (T, \mathcal{A}, m, s)$



- $T = (V, E)$ is a *tree*, V is a vertex set, E is a node set
  - $v_r, v_j, \ldots$ are *nodes*, or *vertices*
  - $V(T)$ is an abbreviation for node set of a tree T
  - $E(T)$ is an abbreviation for edge set of a tree T
- $\mathcal{A} = (A_0, A_1, \ldots, A_n)$ is a variable set, $A_1, \ldots, A_n$ are input variables
- A_0 is target variable
- function m assigns variable to each internal (i.e. non-leaf) node, this variable defines a split
- function s assigns 0 or 1 to each edge $e = (v_1, v_2)$, this value defines value of the split variable in node v_1

Appendix B

Terminology

Binary Decision Tree – see *Decision Tree*, all splits are binary.

Binary Tree – a structure of a decision tree (as a graph in the graph theory), i.e. the decision tree without variables and their values, only nodes and edges are present. The exact definition can be found on page 51 and overview of the Binary Tree can be found in Section A.1 on page 121.

Capacity of the Classifier – see *VC dimension.*

C4.5 – the most popular free implementation of the ID3 algorithm (see *ID3*).

C5.0 – the most recent implementation of the ID3 algorithm (see *ID3*).

CART – the algorithm for decision trees (decision tree induction) which is based on statistical methods and which usually uses post-pruning.

CPD Equivalence of Decision Trees – two decision trees are *CPD equivalent* for a given CPD if their tree CPDs are identical. The exact definition can be found on page 55.

Condition Assigned To Decision Tree Node – a combination of attributes and their values which defines the set of records belonging to a particular node.

CPD – *Conditional Probability Distribution.* See page 49.

CRISP-DM – CRoss-Industry Standard Process for Data Mining. A methodology used in data mining projects.

Decision Tree – a structure used in the *Decision Tree Induction* technique. This structure represents dependencies in order to better estimate the value of the target variable. The exact definition can be found on page 52 and overview of the Decision Tree can be found in Section A.2 on page 122.

Decision Tree Defining Weak Faithfulness – the decision tree, for which all leaves have assigned different values of the CPD. The exact definition can be found on page 64.

Decision Trees – a classification technique, used for data descriptions, classifications and predictions.

Equivalence of Decision Trees – two decision trees are *equivalent* if they have same leaves (defined by identical variable combinations), the only difference may be in the inner structure.

Faithfulness – see *Strong Faithfulness.*

Flat Table – a table which is the input for the data mining procedure. Details can be found in subsection 1.7.

Full Tree – decision tree, where every leaf has all variables in the condition assigned to this leaf.

CHAID – the algorithm for decision tree induction which was originally used for the pattern recognition. It usually uses pre-pruning.

ID3 – the algorithm for decision tree induction which is based on the information theory. It usually uses post-pruning.

Optimal Decision Tree – a decision tree which represents a given distribution and it has the smallest number of leaves from all trees which represent the same distribution.

Parent-Child Exchange – an operation on a decision tree which locally changes the inner structure of the decision tree, leaving the leaves' definition untouched. The exact definition can be found on page 61.

Pre-pruning – a method for reducing the decision tree size in order to prevent overfitting. Also known as *Stopping Rules*. It may be understood as the set of rules which decides whether growing of the tree may continue or not.

Probability measure – the generalization of the (joint probability) distribution to all subsets of $\{0, 1\}^{n+1}$.

Post-pruning – a method for reducing the decision tree size in order to prevent overfitting. First, the tree is grown. Then, the prune operation (see *Prune*) is applied repeatedly (until no pair of neighbouring leaves can be pruned).

Prune – an operation on a decision tree which combines two neighbouring leaves with same conditional probabilities assigned to these leaves. The exact definition can be found on page 59.

Representing distribution – the decision tree represents the distribution if the tree CPD is the same as a given CPD.

Strong Faithfulness – the property of the distribution. It has to satisfy the Weak faithfulness (see *Weak Faithfulness*) and one more condition. The exact definition can be found on page 65.

Tree CPD – a conditional probability distribution, where CPD for a combination of values is defined by CPD assigned to corresponding leaf of a given decision tree.

VC dimension – abbreviation of the Vapnik-Chervonenkis Dimension. It shows how much information can the given classifier (for example, decision trees) store.

Weak Faithfulness – the property of the distribution. For the weak faithful distribution, there exists a tree defining weak faithfulness (see *Tree Defining Weak Faithfulness*) which represents our distribution. The exact definition can be found on page 64.

Appendix C

Detailed results of algorithm testing

This appendix shows sizes of trees, which were reconstructed from data, which were generated from known structures. Details about source data structures and the way of generating data can be found in Section 5.1.
For patterns A and B, only the size of reconstructed tree is shown. There is a correct size shown in bold. For these two patterns, the full tree is the optimal one, so every tree with the correct number of leaves is the tree, from which data were generated.
For remaining 8 patterns, there are shown sizes of the tree for both algorithms, the correct size does not ensure that the correct tree was reconstructed. In the column with the number of trees with the correct size, there is additional information, which is in parentheses **in bold** and which means how many trees have the correct structure (the number of correctly reconstructed trees).

The detailed description of these tables can be found in Subsection 5.1.1.

Table C.1: Pattern A – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with	
	1 node	3 nodes
300	1	**7**
1000	1	**7**
3000	0	**8**
7500	0	**8**
15000	0	**8**
20000	0	**8**
30000	0	**8**

Table C.2: Pattern A – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with	
	with 1 node	3 nodes
300	1	**7**
1000	1	**7**
3000	0	**8**
7500	0	**8**
15000	0	**8**
20000	0	**8**
30000	0	**8**

Table C.3: Pattern B – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with			
	with 1 nodes	3 nodes	5 nodes	7 nodes
300	0	0	3	**5**
1000	0	0	2	**6**
3000	0	0	0	**8**
7500	0	0	0	**8**
15000	0	0	0	**8**
20000	0	0	0	**8**
30000	0	0	0	**8**

Table C.4: Pattern B – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with			
	with 1 nodes	3 nodes	5 nodes	7 nodes
300	0	0	5	**3**
1000	0	0	3	**5**
3000	0	0	0	**8**
7500	0	0	0	**8**
15000	0	0	0	**8**
20000	0	0	0	**8**
30000	0	0	0	**8**

Table C.5: Pattern 1 – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	4(**2**)	1
1000	2	4(**3**)	2
3000	1	4(**4**)	3
7500	0	5(**5**)	3
15000	0	4(**4**)	4
20000	0	5(**5**)	3
30000	0	5(**5**)	3

Table C.6: Pattern 1 – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	5(**2**)	0
1000	2	6(**5**)	0
3000	1	7(**7**)	0
7500	0	7(**7**)	1
15000	0	7(**7**)	1
20000	0	8(**8**)	0
30000	0	8(**8**)	0

Table C.7: Pattern 1b – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	5(**3**)	0
1000	0	8(**5**)	0
3000	0	7(**5**)	1
7500	0	6(**5**)	2
15000	0	5(**5**)	3
20000	0	5(**5**)	3
30000	0	5(**5**)	3

Table C.8: Pattern 1b – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with		
	3 nodes	5 nodes	7 nodes
300	3	5(**3**)	0
1000	0	8(**5**)	0
3000	0	8(**8**)	0
7500	0	8(**8**)	0
15000	0	8(**8**)	0
20000	0	8(**8**)	0
30000	0	8(**8**)	0

Table C.9: Pattern 2 – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	3	4**(3)**	1	0	0	0
1000	0	0	5**(4)**	1	1	0	1
3000	0	0	5**(4)**	0	3	0	0
7500	0	0	4**(4)**	0	3	1	0
15000	0	0	4**(4)**	0	4	0	0
20000	0	0	4**(4)**	0	4	0	0
30000	0	0	4**(4)**	0	3	1	0

Table C.10: Pattern 2 – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	2	5(**4**)	1	0	0	0
1000	0	0	4(**4**)	1	2	1	0
3000	0	0	7(**7**)	0	1	0	0
7500	0	0	7(**7**)	0	1	0	0
15000	0	0	8(**8**)	0	0	0	0
20000	0	0	8(**8**)	0	0	0	0
30000	0	0	7(**7**)	0	1	0	0

Table C.11: Pattern 2b – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	4	4**(3)**	0	0	0	0
1000	0	2	5**(4)**	1	0	0	0
3000	0	2	4**(2)**	0	1	1	0
7500	0	1	6**(4)**	0	1	0	0
15000	0	1	5**(4)**	1	1	0	0
20000	0	0	6**(5)**	1	1	0	0
30000	0	0	5**(5)**	1	2	0	0

Table C.12: Pattern 2b – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	4	2**(2)**	0	2	0	0
1000	0	2	3**(2)**	1	2	0	0
3000	0	3	5**(5)**	0	0	0	0
7500	0	3	5**(5)**	0	0	0	0
15000	0	2	5**5)**	0	1	0	0
20000	0	1	7**(7)**	0	0	0	0
30000	0	0	7**(7)**	1	0	0	0

Table C.13: Pattern 3 – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	2	0	4(**0**)	2	0	0
1000	0	0	1	3(**1**)	3	1	0
3000	0	0	0	4(**2**)	1	3	0
7500	0	0	0	3(**1**)	2	2	1
15000	0	0	0	2(**1**)	2	3	1
20000	0	0	0	2(**1**)	2	3	1
30000	0	0	0	2(**1**)	2	3	1

Table C.14: Pattern 3 – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	2	0	5(**3**)	1	0	0
1000	0	0	0	5(**4**)	2	1	0
3000	0	0	1	**7(6)**	0	0	0
7500	0	0	0	8(**8**)	0	0	0
15000	0	0	0	8(**8**)	0	0	0
20000	0	0	0	8(**7**)	0	0	0
30000	0	0	0	**7(6)**	0	1	0

Table C.15: Pattern 3b – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	1	1	0	1(**1**)	4	0	1
1000	0	0	0	3(**2**)	0	3	2
3000	0	0	1	1(**1**)	2	2	2
7500	0	0	0	2(**2**)	2	2	2
15000	0	0	0	2(**2**)	2	1	3
20000	0	0	0	1(**1**)	3	1	3
30000	0	0	0	2(**2**)	2	0	4

Table C.16: Pattern 3b – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	1	1	2	2(**1**)	2	0	0
1000	0	0	0	5(**5**)	3	0	0
3000	0	0	1	3(**3**)	3	1	0
7500	0	0	0	7(**7**)	0	1	0
15000	0	0	0	7(**7**)	1	0	0
20000	0	0	0	7(**7**)	0	1	0
30000	0	0	0	7(**7**)	0	1	0

Table C.17: Pattern 4 – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	2	1	3	2(**0**)	0	0
1000	0	0	1	3	3(**2**)	1	0
3000	0	0	1	2	2(**2**)	1	2
7500	0	0	0	2	3(**3**)	0	3
15000	0	0	0	1	3(**3**)	1	3
20000	0	0	0	1	4(**3**)	0	3
30000	0	0	0	1	3(**3**)	1	3

Table C.18: Pattern 4 – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	2	4	2	0(**0**)	0	0
1000	0	0	1	4	3(**1**)	0	0
3000	0	0	1	4	2(**2**)	1	0
7500	0	0	0	2	5(**5**)	0	1
15000	0	0	0	2	5(**5**)	0	1
20000	0	0	0	2	5(**5**)	0	1
30000	0	0	0	1	6(**6**)	0	1

Table C.19: Pattern 4b – Results of reconstruction, algorithm CART-PO

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	1	2	5	0(**0**)	0	0
1000	0	0	1	1	4(**3**)	2	0
3000	0	0	0	3	4(**2**)	1	0
7500	0	0	0	1	5(**4**)	1	1
15000	0	0	0	0	5(**4**)	2	1
20000	0	0	0	0	5(**4**)	2	1
30000	0	0	0	0	5(**4**)	2	1

Table C.20: Pattern 4b – Results of reconstruction, algorithm CART-PP

Dataset size	Count of resulting trees with						
	3 nodes	5 nodes	7 nodes	9 nodes	11 nodes	13 nodes	15 nodes
300	0	1	4	2	0(**0**)	1	0
1000	0	0	1	2	4(**3**)	1	0
3000	0	0	0	3	5(**3**)	0	0
7500	0	0	0	2	5(**5**)	1	0
15000	0	0	0	1	6(**4**)	1	0
20000	0	0	0	0	7(**6**)	1	0
30000	0	0	0	0	7(**6**)	1	0