FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Merging of Biomedical Decision Diagrams

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medizinische Informatik

eingereicht von

Dipl.-Ing. Christoph Redl, BSc.
Matrikelnummer 0525250

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: O.Univ.Prof. Dipl.-Ing. Dr. techn. Thomas Eiter
Mitwirkung: Dipl.-Ing. Thomas Krennwallner

Wien, 13. Oktober 2010

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Christoph Redl
Kieslingstraße 9
3500 Krems

Hiermit erkläre ich, dass ich diese Arbeit selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 13. Oktober 2010

(Unterschrift Verfasser)

Abstract

Decision diagrams are an important decision aid in medical applications. One of their main advantages compared with other formalisms like production rules is that they are intuitively understandable by clinicians, health care and administration personal. It is not necessary to be an expert in information systems to act according to a diagram.

Possible application scenarios are medical screening tests, classification of DNA or multidimensional data structures. In screening tests, we usually collect certain chunks of information about the patient. This includes current disorders (if any), blood values, the medical history and data about the personal life style. In certain cases, additional results from image processing techniques like computer tomography can be added. Then we need to decide if this patient shows evidence for the disease in question or not. If this is the case, a medical expert will take a closer look at the patient and request further tests. This procedure can elegantly be represented as decision tree or diagram. The inner nodes refer to some data about the patient and the leaf nodes deliver the answer "yes" or "no".

Another scenario is the classification of DNA in molecular biology. Today, the procedures for automatically sequencing the DNA of organisms have reached remarkable performance. Therefore the next task is to divide the useful subsequences, which encode for proteins from the rest, called *junk DNA*. This can be done by statistical features that are computed for a given sequence, and which vary significantly between coding and non-coding DNA.

These two examples show the relevance of decision diagrams in biomedicine. In this thesis it will be argued, that not only standalone diagrams are of importance, but that we can find scenarios where several similar but not equivalent diagrams have to be merged into a single one. It is highly desirable to have an automatic or semi-automatic procedure that supports this process in order to take the burden of routine tasks from the user.

Therefore it will be shown how such a tool can be implemented. Technically we will use dlvhex, an open-source reasoner for HEX programs. A plugin for dlvhex will be developed, that adds support for decision diagram processing in general and merging in particular. The actual merging step is strongly application dependent, i.e., there does not exist *one* correct result. This problem will be regarded by providing different merging algorithms, where the user can select an appropriate one.

The strength of the plugin and the actual benefit for the user is the possibility to try out different merging algorithms quickly, which makes it possible to focus on the most interesting

tasks like algorithm optimization and result evaluation, whereas routine tasks are performed by the plugin. Finally this tool will be demonstrated when we take a closer look at the DNA classification example.

Zusammenfassung

Entscheidungsdiagramme sind ein beliebtes und wichtiges Hilfsmittel in medizinischen Anwendungen. Einer ihrer Hauptvorteile im Vergleich zu anderen Formalismen, wie etwa Regelsystemen, ist ihre intuitive Verständlichkeit. Dies gilt nicht nur für technische Fachkräfte sondern auch für Mediziner und administratives Personal.

Anwendungsszenarien finden sich unter anderem in der Durchführung von Screening-Tests, bei der Klassifizierung von DNA oder in mehrdimensionalen Datenstrukturen. Bei Screening-Tests werden üblicherweise zuerst verschiedene Informationen über den Patienten erhoben. Dazu zählen beispielsweise Blutwerte, Daten aus der Krankengeschichte und Informationen über die Lebensführung und gegebenenfalls Daten aus bildgebenden Verfahren. Anschließend gilt es zu entscheiden, ob der Patient Symptome für das Vorhandensein einer bestimmten Krankheit zeigt. Ist das der Fall, so ist eine genauere Untersuchung durch einen Arzt angebracht. Die Entscheidungsprozedur bis zu diesem Ergebnis lässt sich elegant als Entscheidungsbaum oder -diagramm darstellen, wie das auch in vielen medizinischen Leitlinien gemacht wird. Die inneren Knoten fragen Patientendaten ab, an den Blattknoten ist schließlich die ja/nein-Entscheidung abzulesen.

Ein weiteres Anwendungsbeispiel ist die Klassifizierung von DNA-Sequenzen in der Molekularbiologie. Heutzutage werden riesige Mengen an DNA automatisch sequenziert und die Performance der eingesetzten Verfahren hat beeindruckende Ausmaße angenommen. Der nächste Schritt ist es, aus diesen Sequenzen nützliche und nutzlose Teilsequenzen, sogenannte *junk-DNA*, herauszufiltern und eine Einteilung vorzunehmen. Dazu können Sets von statistischen Features angewendet werden, die sich für jede beliebige Basenfolge berechnen lassen. Von einigen Features weiß man, dass sie sich signifikant zwischen codierender und nicht-codierender DNA unterscheiden und daher für die Klassifikation geeignet sind.

Diese beiden Beispiele demonstrieren die Relevanz von Entscheidungsdiagrammen in der Biomedizin. In dieser Arbeit wird weiters argumentiert, dass manchmal nicht nur einzelne Diagramme von Bedeutung sind, sondern mehrere, die einander zwar ähnlich aber nicht völlig äquivalent sind. Dies kann beispielsweise daher kommen, dass sie von unterschiedlichen Autoren stammen, die in ihren Studien zu ähnlichen Ergebnissen gekommen sind, die sich aber aufgrund statistischer Schwankungen geringfügig unterscheiden. In diesem Fall ist es wünschenswert ein automatisiertes System für die Vereinigung der Diagramme zur Verfügung zu haben. Dadurch wird der Benutzer von der Notwendigkeit für manuelle Zusammenführung befreit.

Deshalb ist es Ziel dieser Arbeit, eine derartige Prozedur zu entwickeln. Im technischen

Teil wird der Open-Source-Reasoner dlvhex verwendet, für den ein Plugin entwickelt wird, das es zuerst ermöglicht überhaupt Entscheidungsdiagramme verarbeiten zu können, um diese schließlich in einem späteren Schritt zu vereinen. Das Vereinen selbst ist natürlich stark applikationsabhängig, das heißt es gibt nicht *einen* besten Algorithmus der in jedem Fall zum Ziel führt. Aus diesem Grund werden unterschiedliche Varianten implementiert die eine möglichst breite Palette von Szenarien abdecken.

Nutzen und Stärke des Plugins ist die Möglichkeit, ohne manuelles Zusammenführen unterschiedliche Strategien ausprobieren zu können. Wie man aus Erkenntnissen des Bereichs *Machine Learning* weiß, ist die Qualität des Endergebnisses stark von den Trainingssets, den Trainingsalgorithmen und - im Falle von Multi-Classifier-Systemen - den Aggregatprozeduren abhängig. Das Testen und Evaluieren der Möglichkeiten wird in der vorliegenden Arbeit teilautomatisiert und daher wesentlich vereinfacht. Die Software wird schließlich demonstriert, indem wir das Beispiel der DNA-Klassifizierung im Detail betrachten.

Contents

List of Tables

List of Figures

Danksagungen

An dieser Stelle möchte ich mich bei jenen Menschen bedanken, die zum raschen Abschließen dieser Arbeit beigetragen haben.

Mein Dank gilt vor allem meinem Betreuer Prof. Thomas Eiter, der sich in zahlreichen Meetings die Zeit nahm, mir beratend zur Seite zu stehen, wertvolle Verbesserungsvorschläge zu geben und meine Fragen zu beantworten. Ebenfalls mitgewirkt hat Thomas Krennwallner, der bei implementierungstechnischen Fragen stets kompetente Auskunft erteilen konnte.

Dem FWF (Fonds zur Förderung der wissenschaftlichen Forschung[1]) danke ich für die freundliche finanzielle Unterstützung des Projektes *Modular HEX-Programs* (P20841), sowie dem WWTF (Wiener Wissenschafts-, Forschungs- und Technologiefonds[2]) für die Förderung des Projektes *Inconsistency Management for Knowledge Integration Systems* (ICT 08-020), in deren Umfeld ich diese Arbeit schreiben durfte. Dadurch wurde es mir erst ermöglicht an diesem interessanten Thema zu arbeiten.

Weiters haben auch alle weiteren Mitglieder der Arbeitsgruppe für wissensbasierte Systeme ihren Teil dazu beigetragen, mein Interesse auf Logik und logikorientierte Programmierung zu lenken. Dies gilt vor allem für Uwe Egly, Hans Tompits und Michael Fink, deren spannende und kurzweilige Lehrveranstaltungen in den letzten Jahren für mich der Anlass waren, meine Abschlussarbeit letztendlich in diesem Arbeitsbereich zu schreiben.

Einen Beitrag haben auch meine Studienkollegen geleistet. Jeder auf seine Art und Weise. Sei es, indem sie mich durch ihren Ehrgeiz angespornt haben die Ziele hoch zu stecken und Schritt halten zu wollen, oder weil sie in vielen belanglosen Gesprächen für Spaß gesorgt und so das Studentenleben lockerer gemacht haben.

Nicht zuletzt bedanke ich mich natürlich auch bei meiner Familie, die es ertragen hat dass ich in den letzten Jahren trotz körperlicher Anwesenheit oft mit meinen eigenen Gedanken beschäftigt war. Vor allem gilt dieser Dank meinen lieben Eltern, Karl und Anita Redl, die mich während meiner Studienzeit sowohl finanziell wie auch moralisch unterstützt haben und ohne die der Weg weitaus beschwerlicher gewesen wäre.

[1] `http://www.fwf.ac.at`
[2] `http://www.wwtf.at`

More people are killed every year by pigs than by sharks, which shows you how good we are at evaluating risk.

Bruce Schneier

Chapter 1

Introduction

Many medical decisions are based on decision diagrams. For now we informally define them as acyclic graphs, where the inner nodes check some conditions and the leaves contain a proposal for the decision to make if we end in this node. They are often developed by expert groups after extensive studies and are published in clinical guidelines. Since they summarize the current state of the art of science, they are a great tool for medical doctors and health care personal.

Applications include the determination of the best medicine or intervention for a patient depending on his or her medical history and results of examinations. Another scenario is the computation of possible diagnosis for a patient given his or her symptoms as part of computer supported decision systems. For instance, [Mair et al., 1995] present a decision tree for early diagnosis of myocardial infarctions.

A very common use case, that is often implemented in clinical protocols, is the quantification of the degree of severity depending on the patient's condition. For instance, the TNM system classifies tumor diseases with respect to the size of the primary tumor and the spread of metastasis [Sobin et al., 2009]. The suggested kind of treatment depends on the stage.

Decision diagrams are not only relevant in clinical practice. They have also become popular in more basic forms of life sciences in the last decades. One example is presented in [Salzberg, 1995], where decision trees are used to decide whether a given DNA sequence is protein coding or non-coding. As input they use certain features that can be computed for sequences of bases which incorporate knowledge from molecular biology. For instance, certain triplets are known to be more frequent in coding sequences than in non-coding ones. If such a set of features has been computed, one can apply machine-learning algorithms to train classifiers for automatic classification.

However, decision diagrams are not only used in medicine and natural sciences. Applications in other fields can easily be found, for instance in economy and psychology. An economic application is the rating of a company's or country's liquidity depending on certain financial ratios, which is usually done by rating agencies. In psychology, typical tests for diagnosis of personality disorders or mental diseases can easily be organized as decision trees or diagrams. For some more applications see [Bahar et al., 1993].

In many cases there exist multiple decision diagrams for the same purpose. This occurs due to different institutes working on similar projects, similar but not equivalent meanings about correct decisions, statistical impreciseness or simply human errors. If one does not have any

preferences about the trustability of the different decision diagram providers, it is necessary to combine them somehow into one compact and coherent diagram.

As the main contribution of this thesis, we will develop and implement a tool which supports this process, i.e., the semi-automatic incorporation of multiple diagrams into a single one. This takes the burden of performing routine tasks from the user. Since we are going to use logic programming for the implementation, we first need to find a way to turn decision diagrams into "objects" which are accessible from the logic program. As we will see, this can be done by a straightforward encoding.

1.1 Existing Approaches

The incorporation of several *classifiers*, which is a generalization of decision diagrams, is called *ensemble learning* and comes from the field of machine learning. It has already been studied and well working methods have been developed. Two of the most successful strategies are *boosting* and *bagging*. For an overview about ensemble learning methods see for instance [Dietterich, 2000], [Maclin and Opitz, 1997] or, for a quick introduction, [Polikar, 2009].

We will shortly describe boosting and bagging. In bagging (*bootstrap aggregating*), n different classifiers are trained independently. The underlying annotated training set is used to randomly draw a subset for each of the training passes. Thus, the classifiers will be trained on partially overlapping and partially differing training sets. After training has been done and a new element needs to be classified, this is first done independently by each classifier. Then the results are combined by a simple majority voting rule.

Bagging is very similar. The only difference concerns the selection of the training set for the individual classifiers. While bagging draws them randomly from the overall training set, boosting makes sure that when classifier C_i is trained, half of its training samples are correctly and half of them are incorrectly classified by C_{i-1}. Informally this strategy tries to select *interesting* samples, namely those where the results of existing classifiers are moderate and leave room for improvement.

1.2 Going Beyond

All of the methods presented in Section 1.1 have some underlying assumptions that we will put into question. First of all, in most cases the merging algorithms assume that the underlying data, which was used when the input decision diagrams were created, is still available. But this may be not the case because of different reasons like business secrets or privacy. But even if the original training set is still available, it is sometimes simply undesired to train a new classifier. Maybe one prefers to combine some of the most successful and well-tested diagrams directly.

Additionally, many approaches do not actually merge decision diagrams into a standalone one. Instead they create a decision *procedure* which refers to the source diagrams and summarizes the results, e.g., by applying a majority voting rule as we have seen before. In clinical practice this solution would not be acceptable since a decision aid must be intuitively understandable and easily applicable. Users will in general not be knowledge engineers that are used to work with a variety of sources they need to combine.

A further difference to our approach is that most merging algorithms convert decision diagrams into rule-based systems before they are combined. We will directly work with diagrams and skip this intermediate representation formalism.

To summarize, this thesis is intended to develop a procedure which actually creates a standalone decision diagram from multiple ones. It will be based on an extended version of answer

set programming using dlvhex, which is another essential difference to existing approaches that are mainly based on machine learning techniques rather than logic programming. We are not going to develop one specific merging algorithm, but a fairly *flexible framework*, which can be parameterized depending on the needs of a certain application.

1.3 Intention of the Approach

The merging of decision diagrams can also be done by hand. So what is the advantage of using a merging framework? In many settings it is not clear from the beginning which merging strategy or which combination of strategies gives the best result. Thus, one has to try out several operations and measure the performance of the result. This is suboptimal since the user has to deal with routine tasks instead of the optimization of the merging strategy.

Therefore, a framework that allows the declarative specification of merging plans is highly desirable. Then the user can spend more efforts in the development and improvement of merging operators while the technical details of the merging process are managed automatically.

The framework is useful even if the merging strategy is already fixed. It is well-known that the performance of classifiers strongly depends on the selected training data and the correct selection of parameters. Modified start conditions will lead to classifiers that behave differently. It is tedious if the merging has to be done repeatedly each time one of the source classifiers is modified or exchanged.

Consequently, the intention of the proposed decision diagram merging procedure is to ease the process of finding appropriate parameters by performing routine tasks automatically and give the user the chance to focus on the interesting aspects of the procedure.

1.4 Applications and Experimental Results

We will apply our framework and its implementation on some practical applications of life sciences. Our most detailed case study deals with the classification of DNA sequences as being protein-coding or non-coding. This can be done by computing statistical features of the sequences of an annotated set, and training a decision diagram upon them. Subsequently this diagram can be used to classify new sequences.

As we will see, training *multiple* different diagrams and merging them in a post-processing step has several advantages over the training of a single classifier. The results of our experiments show, that combining classifiers trained by *different* algorithms may increase the accuracy; this reminds on the principle of recombination in nature. We could also observe that the total size of the training set which is necessary to reach a certain accuracy seems to decrease when multiple diagrams are trained and merged.

Clearly, the quality of the final diagram depends on the quality of the training set, the selection of the training algorithms and the merging procedure in use. Some combinations increase the quality, while others have no influence or even decrease it. But this exactly shows why our framework is useful. Instead of wasting time by manually incorporating diagrams after each change of parameters, the user may focus on the evaluation of the result, while the technical details of the merging are managed automatically.

Other applications of our framework include medical screening tests, which can elegantly be represented by a decision diagram, or, more technically, multidimensional index structures in database systems. A classical application of medical informatics are decision support systems, which may also be based on decision diagrams.

Ah! That is a beautiful assumption; it explains many things.

Pierre-Simon Laplace

Chapter 2

Preliminaries

This chapter summarizes the preliminaries for the work in later chapters. A more detailed description of many topics of this chapter is given in [Redl, 2010]. We first give a short introduction to logic programming under the answer-set semantics. This is a modern declarative formalism that has been widely accepted and used. Nevertheless it is by far not the only one developed.

We then continue with HEX programs as an extension of this semantics since this will be the formalism we work with in this thesis. The reasoner in use will be dlvhex[1] which uses DLV[2] in the background and extends it with new features.

Finally we give a short introduction into the mergingplugin which is an extension of dlvhex and has been developed as part of another master's thesis [Redl, 2010] since we build upon this plugin and extend it further.

2.1 Answer-Set Programming

The very beginnings of logic programming were dominated by resolution-based methods, see for instance [Kowalski, 1974]. But because of several drawbacks, where the most obvious one is the lack of a possibility to derive negative information, completely new semantics were developed subsequently. This includes the least fixed point semantics [Fitting, 1999] and the stable-model semantics [Gelfond and Lifschitz, 1988]. The latter one was finally extended to the answer-set semantics [Gelfond and Lifschitz, 1991].

In the following sections we will restrict our discussion to aspects which are relevant with respect to the answer set semantics. In particular, we are going to skip historical methods like resolution-based reasoning and the stable model semantics. A more detailed and stepwise introduction of the answer set semantics is given in [Redl, 2010] or the underlying literature.

Least Fixed Point Semantics

To overcome the problems of resolution-based methods, a completely new semantics, named the *least fixed point semantics*, for logic programs has been introduced. We are going to follow

[1] `http://www.kr.tuwien.ac.at/research/systems/dlvhex`
[2] `http://www.dbai.tuwien.ac.at/proj/dlv`

[Fitting, 1999] and keep working with programs of the following type for now.

Definition 2.1. A classical logic program P is a set of Horn rules, where a Horn rule r is of the form

$$r = H \leftarrow B_1, \ldots, B_n.$$

with B_i $(1 \leq i \leq n)$ being atoms and H being an atom or empty.

Intuitively the evaluation procedure can be described as follows. The facts (a rule with $n = 0$) are initially *true* since they have no premises. Then a rule $H \leftarrow B_1, \ldots, B_n.$ enforces us to set H to *true* whenever all of B_i are *true*. Thus, the set of *true* variables is continuously expanded. The procedure is repeated until no more atoms can be added. Then we say that the *least fixed point* is reached.

Formally we define the operator $\Gamma(P, A)$, where P is a positive logic program and A a set of atoms.

Definition 2.2. The semantics of $\Gamma(P, A)$ is another set of atoms A', s.t. $H \in A'$ iff there exists a ground instance of a rule $H \leftarrow B_1, \ldots, B_n.$ in P and $B_i \in A$ $\forall 1 \leq i \leq n$. The semantics of a program P is the least fixed point of this operator, i.e., $lfp(\Gamma(P, \emptyset))$.

Example 2.1. Consider the program

$$\begin{aligned} P = \{a. & \\ b &\leftarrow a. \\ c &\leftarrow a, b.\} \end{aligned}$$

Obviously $lfp(\Gamma(P, \emptyset)) = \{a, b, c\}$.

Note that these definitions basically remain the same when we allow the use of literals (atoms or strongly negated atoms) instead of atoms. We will presuppose corresponding definitions in the following subsections.

Grounding

In the remaining part of this chapter we will further assume that all programs are free of variables. This happens without loss of generality, since programs with variables can be easily transformed into a variable-free version by a procedure called *grounding*. That is, a variable occurring in a rule is simply treated as shortcut for all the rules that can be constructed by replacing the variable by arbitrary domain elements, see Definition 2.4 and 2.5. Note that for function symbols, the grounding is possibly infinite. However, this is not relevant for our purposes since DLV and dlvhex do not support function symbols. For an illustration of grounding see Example 2.2.

Definition 2.3. A first-order signature $\Sigma = \langle \Sigma_v, \Sigma_p, \Sigma_c \rangle$ consists of a set of variables Σ_v, a set of predicate symbols Σ_p and a set of constant symbols Σ_c from a first-order vocabulary ϕ.

The set of terms over Σ is denoted as $\mathcal{T}$.

Definition 2.4. A term $t \in \mathcal{T}$ is called *ground* iff it contains no variables.

Definition 2.5. The grounding of a ground term (a ground program) is the term (the program) itself. The grounding of an arbitrary term $t \in \mathcal{T}$, denoted as $ground(t)$ (an arbitrary program P, denoted as $ground(P)$), is the set of all variable-free versions that can be obtained by replacing variables by arbitrary ground terms.

Example 2.2. Let's consider the non-ground program

$$\begin{aligned} P = \{ & f(a). \\ & g(b). \\ & f(X) \leftarrow g(X).\} \end{aligned}$$

Then the according grounded program is

$$\begin{aligned} P_{grounded} = \{ & f(a). \\ & g(b). \\ & f(a) \leftarrow g(a). \\ & f(b) \leftarrow g(b).\} \end{aligned}$$

Answer-Set Semantics

Before we can discuss the answer set semantics we need to introduce some preliminary definitions.

Definition 2.6. The Herbrand universe of a program P, denoted as $HU(P)$, is the set of all ground terms occurring in P.

Definition 2.7. The Herbrand base $HB(P)$ of a program P is the set of all ground atoms over the Herbrand universe and the predicate symbols in P.

Definition 2.8. A Herbrand interpretation $I \subseteq HB(P)$ is any subset of the Herbrand base.

Definition 2.9. A literal L is either an atom A or a strongly-negated atom $\neg A$.

The following definition will introduce the syntax of the kind of logic programs we are going to use in later chapters.

Definition 2.10. An extended logic program P consists of rules of the form

$$H_1 \vee \cdots \vee H_k \leftarrow B_1, \ldots, B_n, \text{not } B_{n+1}, \ldots \text{not } B_m.$$

where H_i is the possibly empty set of head literals and B_i are the body literals.

In contrast to the stable model semantics, the answer set semantics uses two types of negation. We will use the symbol $\neg$ for classical (strong) and "not" for default negation (negation as failure). Further we will denote the set of default-negated body literals $\{B_{n+1}, \ldots, B_m\}$ of a rule r as $B^-(r)$ and the set of non-default-negated (but possibly strongly negated) literals $\{B_1, \ldots, B_n\}$ as $B^+(r)$.

The semantics of extended logic programs is called the *answer-set semantics*, which is an extension of the stable model semantics. In the stable model semantics, strong negation does neither occur in the program nor in the answer of the program, i.e., the answer of a program is expected to be a set of ground *atoms*.

In contrast to that, extended logic programs evaluate to sets of ground *literals*, called *answer sets* [Gelfond and Lifschitz, 1991].

Formally we can define answer sets as follows.

Definition 2.11. Let P be an extended logic program and $M \subseteq HB(P)$ be a Herbrand interpretation. Then the *reduct* P^M is the program obtained from P by

- discarding all rules r with $B^-(r) \cap M \neq \emptyset$

- removing all default-negated literals from the bodies of the remaining rules

If the unique minimal Herbrand model of P^M coincides with M, it is an answer set of P.

Note that this definition allows us to use disjunction in the rule heads. So called *disjunctive logic programs* were first described by [Przymusinski, 1991] under the stable model semantics and then adopted by Gelfond and Lifschitz for the answer-set semantics.

Example 2.3. The program

$$\begin{aligned} P = \{p &\leftarrow \text{not}\, q. \\ q &\leftarrow \text{not}\, p. \\ \neg r &\leftarrow p.\} \end{aligned}$$

has two answer sets, namely $AS_1 = \{p, \neg r\}$ and $M_2 = \{q\}$ because AS_1 is the (unique) least model of $P^{AS_1} = \{p.,\ \neg r \leftarrow p.\}$ and AS_2 the least model of $P^{AS_2} = \{q.,\ \neg r \leftarrow p.\}$. In contrast, $M_3 = \emptyset$ is not an answer set of the program since it does not coincide with the least model of $P^{\emptyset} = \{p.,\ q.,\ \neg r \leftarrow p.\}$, which is $\{p, q, \neg r\}$.

2.2 Intoduction to HEX programs and dlvhex

The open-source software dlvhex is a reasoner for a generalization of logic programs under the answer-set semantics called HEX programs. The latter generalize extended logic programs in the following two ways.

First they bring support for higher-order atoms. In contrast to first-order atoms, the predicate name is not necessarily a constant but can also be a variable. For instance, $X(a, Y)$ is a higher-order atom since X is a variable whereas $z(a, Y)$ is an ordinary atom. In general a higher order atom is of form

$$Y_0(Y_1, \ldots, Y_n)$$

which can also be written as $n + 1$-ary tuple $(Y_0, Y_1, \ldots, Y_n)$. HO-atoms collapse to first-order atoms after grounding, therefore the semantics is equivalent to extended logic programs.

The second enhancement, which is more important in our context, is the support of external atoms. They allow a bidirectional communication between the HEX program and an external source of computation. In this section we explain them with focus on practical usage, for a theoretic discussion we refer to the above literature.

An external atom $\&g$ is of the form

$$\&g[A_1, \ldots, A_m](O_1, \ldots, O_n),$$

where $A_1, \ldots, A_m$ are the *input parameters* (we use A for *arguments* instead of I to avoid confusions with interpretations) and $O_1, \ldots, O_n$ are the *output parameters*, A_i can either be a constant or a predicate name that needs to be passed to the external source. In case of a constant, it is passed just as is. In case of a predicate name, all atoms in the current interpretation that a built upon this predicate are passed. This is called *restricted interpretation*. Then the source is called, which needs to compute a set O of n-ary output tuples (based on the input parameters). Finally the computation of the logic program continues where $\&g[A_1, \ldots, A_m](O_1, \ldots, O_n)$ evaluates to *true* iff $(O_1, \ldots, O_n) \in O$.

In practice, external atoms are implemented in form of C++ classes that are compiled as shared object libraries and placed in the plugin directory of dlvhex. There they can be found and loaded on startup such that the atom can evaluated as needed.

Example 2.4. The program

$$\begin{aligned} P = \{ & s(\text{“}hello\ \text{”}). \\ & s(\text{“}dlvhex\ \text{”}). \\ & result(Z) \quad \leftarrow \quad s(X), s(Y), \&concat[X, Y](Z). \} \end{aligned}$$

has an answer set with 4 atoms over *result*, namely:

$$\begin{aligned} AS_1 = \{ \quad & result(\text{“}hello\ hello\ \text{”}), result(\text{“}dlvhex\ dlvhex\ \text{”}), result(\text{“}hello\ dlvhex\ \text{”}), \\ & result(\text{“}dlvhex\ hello\ \text{”}), s(\text{“}hello\ \text{”}), s(\text{“}dlvhex\ \text{”})\} \end{aligned}$$

The external atom &*concat* is implemented in the string plugin and computes the concatenation as usual.

For a more detailed and formal introduction of HEX programs see [Redl, 2010] or directly the underlying basic literature, e.g., [Eiter et al., 2006].

2.3 Intoduction to the mergingplugin

The mergingplugin is a plugin for dlvhex which was developed as part of the master's thesis [Redl, 2010]. Basically it consists of two components: a set of external atoms and the command-line tool mpcompiler.

Nested HEX Programs

The external atoms implemented in the mergingplugin allow HEX programs to call other HEX programs, let them compute their result independently from the calling program, and continue computation in the host program afterwards. This is best explained with an example.

Example 2.5. In [Eiter et al., 2005] a logic program P for demonstrating the new features of HEX programs in comparison to extended logic programs under the answer-set semantics is shown. It computes a randomly selected group of two or three persons among John's relatives and invites them. Let's assume that P returns atoms over the binary predicate *invites*, for instance

$$invites(john, sue)$$

to represent that *sue* is invited. Each answer set of the program will contain exactly one possible selection of relatives, i.e., two or three atoms of the given form.

Now suppose that we are not interested in the invited persons themselves, but only in the number of possible selections. This combinatorial problem cannot be solved in pure HEX programs because it is not possible to *count* the number of answer sets, since at each point during reasoning, we can only "see" the contents of the currently computed answer set. What we need is a mechanism for calling sub-programs, i.e., reasoning on the level of *sets of answer sets*. But this is not possible without special external atoms.

The mergingplugin implements exactly such external atoms, namely:

- &*hex*[*Prog*, *Args*](*A*)
  This executes a program (given as string literal *Prog*) with certain command-line options (*Args*). The result is an integer value *A* that serves as *handle*[3].

[3] A handle is similar to a pointer: the numeric value is irrelevant, but it can be used to access this answer later on.

- $\&hexfile[File, Args](A)$
  This executes a program stored in the file with name *File* (a string literal) with certain command-line options *Args*. The result is again a handle A.
- $\&answersets[H](AS)$
  This atom takes the handle H to a program's answer and returns zero or more handles AS to the answer sets contained in this answer.
- $\&predicates[H, AS](Pred, Arity)$
  $\&predicates$ takes the handle AS to an answer set within a program's answer H. Its result is a list of predicates occurring in this answer set (*Pred*) paired with their arities (*Arity*).
- $\&arguments[H, AS, Pred](I, ArgIndex, Value)$
  $\&arguments$ takes the handle AS to an answer set within a program's answer H as well as some predicate name *Pred*. Its result is a list of triples of the following form. The first element I is a running index that states with triples belong together (namely all with the same index). The second and third argument state the *Value* of the argument with index *ArgIndex* in the I-th occurrence of the given predicate.

  The special index s for *ArgIndex* denotes the *sign* of the atom, where the possible values are 0 (for positive) and 1 (for strongly negated).

This is demonstrated with an example which was taken from the end-user documentation of the mergingplugin. For more details see the cited thesis, which includes the user guide in its appendix.

Example 2.6. The program

$$\begin{aligned} P = \{val(Pred, I, ArgIndex, Value) \quad \leftarrow \quad & \&hex[\text{“}p(a,b).\ \neg p(x,y).\ q(f).\text{”}, \text{“”}](H), \\ & \&answersets[H](AS), \\ & \&predicates[A, AS](Pred, Arity), \\ & \&arguments[A, AS, Pred] \\ & \quad (I, ArgIndex, Value).\} \end{aligned}$$

will have one answer set, namely

$$\begin{aligned} \{&val(p,0,s,0), val(p,0,0,a), val(p,0,1,b), \\ &val(p,1,s,1), val(p,1,0,x), val(p,1,1,y), \\ &val(q,0,s,0), val(q,0,0,f)\} \end{aligned}$$

The inner (nested) program obviously has just one answer set, namely $\{p(a,b), \neg p(x,y), q(f)\}$. The result of the outer program (host program) expresses that in the 0-th occurrence of p, the *s*ign is positive (0), the 0-th parameter is a and the 1-st parameter is b.

Similar for the 1-st occurrence of p, where the sign is negative (1), the 0-th argument is x and the 1-st one is y.

q occurs only once (positively) and has just one parameter which is f.

Example 2.7. The following example solves the problem of counting the number of answer sets returned by program “invitations.hex” (continuation of Example 2.5).

$$\begin{aligned} P = \{as(AH) \quad \leftarrow \quad & \&hexfile[\text{“}invitations.hex\text{”}, \text{“”}](HP), \\ & \&answersets[PH](AH). \\ number(D) \quad \leftarrow \quad & as(C), D = C + 1, \text{not } as(D).\} \end{aligned}$$

If just figures out the highest value D such that $D-1$ is a valid handle to an answer set and D is not.[4] Or in other words, we compute D such that it is the number of answer sets of P.

Operator Applications

The mergingplugin allows the specification of so called *merging operators*. Formally, an n-ary operator maps n sets of answer sets over some signature Σ plus m additional parameters to a new set of answer sets.

Definition 2.12. An n-ary operator with m additional arguments of types $\mathcal{D}_i$ $(0 \leq i < m)$ is a function

$$\circ^{n,m} : \underbrace{\left(2^{\mathcal{A}(\Sigma)}\right)^n}_{belief\ \ bases} \times \underbrace{\mathcal{D}_0 \times \ldots \times \mathcal{D}_m}_{additional\ \ parameters} \rightarrow 2^{\mathcal{A}(\Sigma)}$$

where $\mathcal{A}(\Sigma) = 2^{Lit_\Sigma}$.

The implementation of operators is done via an external atom of the following form:

$$\&operator : String \times A \times P \rightarrow \mathbb{N}$$

Its input is a string literal that selects the operator to apply, a sequence $A = \langle a_1, \ldots, a_n \rangle, a_i \in \mathbb{N}$ of answers to be passed to the operator, and m additional parameters of kind key-value pairs, given as sequence $P = \{(k_1, v_1), \ldots, (k_m, v_m)\}, k_i, v_i \in String$.

Operators are in general user-defined (even though the mergingplugin comes with a few predefined ones). They need to be implemented in form of a shared object library that provides a function with the following signature:

```
extern "C" std::vector<dlvhex::merging::IOperator*> OPERATORIMPORTFUNCTION()
```

This function must return a vector containing pointers to all operators provided by this library, where each operator is a class that implements the interface IOperator defined in the mergingplugin. Such a shared object library can either be installed into the dlvhex plugin directory, where it is automatically found on startup, or in a user defined directory that is explicitly added to the search path using the command-line option `--operatorpath`.

This informs the plugin about the existence of certain operators. Later on, they can be used within *merging plans*, which is the topic of the following section.

Merging Plans

Merging plans generalize the concept of merging operators by organizing them hierarchically. This means that the result of an operator application can be passed as argument to a further operator. This gives a tree-like structure that is similar to syntax trees for arithmetic expressions, where the leaf nodes are formed by knowledge bases and the inner nodes by merging operators.

Example 2.8. The following snippet shows a merging plan that is intuitively readable. The knowledge bases *kb2* and *kb3* are first unioned and finally subtracted from *kb1* (using set minus operation).

```
[merging plan]
{
        operator: setminus;
        {
                bb1
```

[4] Handles start with value 0.

```
        };
        {
                operator: union;
                {
                        bb2
                };
                {
                        bb3
                };
        };
}
```

Without going into detail and explaining how this works, a merging plan like this can be translated into a semantically equivalent HEX program by the use of the command-line tool mpcompiler[5] that is installed as part of the mergingplugin. The resulting program can then be executed by dlvhex, which will give the result of the topmost operator application of the merging plan. In the example, this would be the result of *setminus*.

This concludes our introduction to the mergingplugin. For more information see [Redl, 2010] where this plugin is developed step by step. We now take a closer look at decision diagrams and their applications in medicine. In Chapter 4 we come back to merging plans and use them for diagram merging.

[5] merging plan compiler

A weak man has doubts before a decision; a strong man has them afterwards.

Karl Kraus

Chapter 3

Decision Diagrams in Biomedicine

Decision diagrams and especially decision trees are an important means for decision making in clinical practice. [Shortliffe et al., 1979] introduces clinical algorithms in form of flowcharts. Even though they may consult human experts in cases with exceptional circumstances, such protocols are an important tool as basis for efficient patient handling. Medical personal can simply follow the description in standard cases and concentrate on the few cases that need special attention.

Since then, many clinical guidelines were published that contained decision diagrams, and some of them were implemented in clinical expert systems. For instance, [Mair et al., 1995] present a decision tree that supports the early diagnosis of myocardial infarctions under certain circumstances.

An interesting approach is presented in [Althoff et al., 1998]. It is called *INRECA approach* with the well-known underlying idea that a current medical case can be solved (that is, an accurate medication or therapy is selected) by comparing it to past cases. Past cases consist of symptoms and the therapy that finally has lead to success. If a the actual case is similar to a certain past case, it is very probable that a similar therapy will do the job.

But how can medical cases be compared? For that purpose they introduce a new data structure called the *Inreca-tree*, which is essentially a decision tree. In its inner nodes, certain parameters like blood values are queried, where there is an outgoing edge for each possible result plus the special value *unknown*. The existence of such an else branch in each node makes sure that the tree can deal with incomplete knowledge. The leaf nodes are references to past cases that are similar to the current one. Thus, something like a decision tree is used to quickly find cases that are of interest for the current patient (even though the *Inreca-tree* finally leads to a *class* of cases rather than a single classification as in case of pure decision trees).

A further application of decision trees comes from the field of tumor staging. The *TNM system* [Sobin et al., 2009] classifies tumor diseases into 4 stages, depending on the size of the primary tumor (T), the number of lymph node metastasis (N) and the existence or absence of metastasis in non-lymph organs (M). The distinction has direct influence on the suggested therapy. This system has been established as generic appraisal tool for determining the severity, where the details depend on the type of cancer. For instance, [Mountain, 1986] investigate certain types of lung cancer.

Even though most medical papers contain decision tables rather than decision trees (and sometimes redundancy), they can be easily converted into a tree. In case of the TNM system

this would even be the more natural representation formalism since not all combinations of values for T, N, and M are relevant. Some queries depend on prior answers. For instance, in case of existence of metastasis in non-lymph tissue (M), the final result will be *stage 4* (most serious), independent of the values of T and N.

3.1 Formal Definition of Decision Diagrams

After this informal description of decision diagrams we need to define them mathematically now. We talk about nothing more than an algorithmic representation of a discrete *decision function*, which assigns a class to each element of a certain domain. In detail we characterize decision diagrams similar to [Moret et al., 1980]. However, note that they consider decision *trees* whereas we consider decision *diagrams*, but the concept can easily be generalized. We will come back to the difference in later sections.

Definition 3.1. Let $\mathcal{D}$ be a domain and $\mathcal{C}$ a set of classes, i.e., sets of arbitrary elements which serve as domain elements resp. classes. Then a *classification function* c (*classify*) has the signature $c : \mathcal{D} \to \mathcal{C}$ and can be represented as decision diagram

$$D = \langle V, E, l_c, l_e \rangle ,$$

where V is the set of nodes and $E \subseteq V \times V$ the set of directed edges such that (V, E) is a directed acyclic graph and D has a unique root node r_D, i.e., $\exists n \in V$ s.t.

$$d^-(r_D) = 0 \text{ and } d^-(n) = 0 \Rightarrow n = r_D$$

(there is exactly one node with no ingoing edges)
The function

$$\mathit{leaves}(V) := \{v \in V \mid \nexists : (v, u) \in E\}$$

denotes the set of leaf nodes in V. Then, the function

$$l_c : \mathit{leaves}(V) \to \mathcal{C}$$

assigns a class to each leaf node and

$$l_e : E \to 2^{\mathcal{D}}$$

a subset of the domain to each edge.

Informally, $l_e(e)$ represents any condition that can either be satisfied or violated by a domain element. When the classification of a certain domain element x needs to be determined and a certain node v is reached, we follow the (unique) edge $e = (v, u)$ with $x \in l_e(e)$.

This is formalized by the recursive Algorithm 1. We will write $D(d)$ to denote the application of this algorithm to domain element d and the root element of decision diagram D.

In order to guarantee that the algorithm is deterministic and the result is unique, we need the following extra requirements:

(a) $\bigcup_{(v,u)\in E} l_e((v, u)) = In(v)$, *for all* $v \in V \setminus (\{root\} \cup \mathit{leaves}(V))$
where $In(v) = \bigcup_{(p,v)\in E} l_e((p, v))$ and $root = v \in V$ s.t. $\nexists p \in V : (p, v) \in E$
This states that at any point during the execution of Algorithm 1, if a node is reached for element d, there must be an outgoing edge for this element, i.e., computation can always continue.

Algorithm 1: classify(d, $root$)

```
Input: d ∈ 𝒟, root ∈ V
Output: c ∈ 𝒞
if d ∈ leaves(V) then
    return l_c(d)
else
    for u : (root, u) ∈ E do
        if d ∈ l_e((root, u)) then
            return classify(d, u)
    /* This point is never reached due to requirements (a) and (b)   */
```

(b) $\bigcup_{(root,v)\in E} l_e((root, v)) = \mathcal{D}$
We need a similar condition for the root node. The root must have an outgoing edge for each domain element.
Together with the previous requirement, this one guarantees that the decision tree can classify all domain elements.

(c) $\forall v \in V\ \forall e_1 = (v, u) \in E\ \forall e_2 = (v, w) \in E : l_e(e_1) \cap l_2(e_2) \neq \emptyset \Rightarrow u = w$
Successors of decision tree nodes must be unique at any point of computation. Thus the conditions of the edges must be pairwise disjoint.

These abstract definitions are illustrated with the following example.

Example 3.1. Assume a domain:

$$\mathcal{D} = \{1, 2, \ldots, 15\}$$

and classes:

$$\mathcal{C} = \{prime, not\ prime\}$$

Then a possible decision diagram is:

$$D = \langle \{r, p, n\}, \{(r, p), (r, n)\}, l_c, l_e \rangle$$

with leaf classes

$$l_c(p) = prime, l_c(n) = not\ prime$$

and conditions

$$l_e((r, p)) = \{2, 3, 5, 7, 9, 11, 13\},\ l_e((r, n)) = \{1, 4, 6, 8, 10, 12, 14, 15\}$$

If an element needs to be classified, e.g., $e = 7$, we start at node r and check whether $e \in l_e((r, p))$ or $e \in l_e((r, n))$. In this case $e \in l_e((r, p))$ holds, which directs us to leaf node p. Therefore the classification for $e = 7 = l_c(p) = prime$.

Since we are not going to talk about decision diagram *evaluation* but about decision diagram *merging*, we will have to deal with arbitrary diagrams over a certain signature in the later chapters. Thus we need one more definition that allows us to specify the necessary operators.

Definition 3.2. The set of all decision diagrams over domain $\mathcal{D}$ and classes $\mathcal{C}$ is denoted as $\urcorner_{\mathcal{D},\mathcal{C}}$.

This concludes the basic definition of decision diagrams. Even though the remaining part of this chapter will not lead to gain in expressiveness, it simplifies upcoming definitions.

3.2 Query Language

In Section 3.1 we defined decision diagrams as quadruple containing in one component:

$$l_e : E \to 2^{\mathcal{D}}$$

Edges are assigned sets of domain elements, such that Algorithm 1 can decide in a top-down manner which way to go.

Observe that in practice it is cumbersome to construct a decision diagram this way, since one would need to decide for each domain element if is shall be included in an edge's label or not. It is much more convenient to use a query language.

Thus we insert an intermediate formalism "between" edge labels and sets of domain elements. Formally, this means that we change the definition of a decision diagram with

$$D = \langle V, E, l_c, l_e \rangle$$

and

$$l_e : E \to 2^{\mathcal{D}}$$

Definition 3.3. A decision diagram under a query language Q is a quintuple

$$D = \langle V, E, l_c, l_e, \mathcal{M}_Q \rangle$$

with V, E and l_c as usual and

$$l_e : E \to \Sigma_Q$$

and

$$\mathcal{M}_Q : \Sigma_Q \to 2^{\mathcal{D}},$$

where Σ_Q is the set of syntactically correct queries over an application-specific query language. A query can be evaluated using the meaning function $\mathcal{M}_Q$, which maps syntactic expressions to sets of domain elements, namely those that satisfy the expression.

All concepts from above remain valid. The only modifications affect the access of edge labels, which changes from $l_e(e)$ to $\mathcal{M}\left(l_e(e)\right)$ and gives the developer of a certain application the freedom to design a query language which is suitable for a specific use case. Note that in fact this is nothing more than syntactic sugar, since the only thing that changes is that the domain elements assigned to edges are now given implicitly by the query rather than explicitly.

Using a first-order like query language called F, we could rewrite the decision diagram from example 3.1 as follows.

Example 3.2. Let

$$\mathcal{D} = \{1, 2, \ldots, 15\}, \mathcal{C} = \{prime, not\ prime\},$$

and

$$D_F = \langle \{r, p, n\}, \{(r, p), (r, n), \}, l_c, l_e, \mathcal{M}_F \rangle$$

with

$$l_c(p) = prime, l_c(n) = not\ prime$$

and

$$l_e((r, p)) = f_1 = \nexists x, y : x > 1 \wedge y > 1 \wedge x \cdot y = z$$
$$l_e((r, n)) = f_2 = \exists x, y : x > 1 \wedge y > 1 \wedge x \cdot y = z$$

with the meaning function $\mathcal{M}_F(f) = \{n \in \mathcal{D} : I_n(f) = true\}$ where I_n is a first-order interpretation with $I_n(z) = n$.

The meaning function $\mathcal{M}_F(f)$ is applied to one of the edge conditions f. Then this function constructs a first-order interpretation I, that maps the free variable z of the conditions to a domain element that shall be classified. The function will return the set of all domain elements which satisfy the condition.

Observe that

$$\mathcal{M}_F(f_1) = \{2, 3, 5, 7, 9, 11, 13\} = \mathcal{D} \setminus \mathcal{M}_F(f_2)$$

which is equivalent to our initial version of the decision diagram. However, this does not need to be explicitly defined now but is hidden in the definition of the meaning function and the semantics of first-order logic.

Designing a Query Language

In principal it would be possible to design an application specific language for each decision diagram. However, this is a lot of work and additionally introduces a lot of problems when diagrams shall be merged in Chapter 4. It is more reasonable to design *one* language that is expressive enough for a large variety of applications.

The design of the following language is greatly inspired by typical *if-then* statements in well-known programming languages. Basically there are four types of scales: *nominal scales*, *ordinal scales*, *interval scales* and *ratio scales*. This classification was introduced in [Stevens, 1946] and since then cited many times.

Table 3.2 summarizes the main properties of these types as described by Stevens.

Type	Allowed operations	Examples
nominal	testing for equality	colors, employees
ordinal	additionally: testing for $<$, $>$	marks in school
interval	additionally: computing differences	dates
ratio	additionally: computing quotients	temperatures, durations

Table 3.1: Classification of scales

Nominal values can only be tested for equality. For instance, if X and Y are colors, we can clearly check if they are equal. But it makes no sense to say that one of them is greater than the other one.

This is only possible with ordinal scales. Marks in school are a classical example. Comparability is the underlying idea of all grading schemes. Thus it is possible to say that mark 2 is better than mark 4, but it is still useless to compute the difference between two marks. Even though some grading schemes try to incorporate this possibility to some extent (for instance, in the United States grade F is failed and thus *much* worse than D, which is expressed by the missing grade E), there is in general no defined unit for expressing differences.

Using interval scales overcome this restriction. In case of dates this is clearly possible. Expressions like "one week later" or "10 days before Christmas" are used everyday. But there is still no absolute zero point and, consequently, ratios are meaningless.

Finally, ratio values are the most general scale. They allow for all the operations we considered and additionally computing quotients. Thus they allow for statements like "This project lasts twice as long as the last one".

We now come back to the design principles of a reasonable query language. If we think about usual programming languages, the scale types of interest are clearly nominal and ordinal values.

In conditional blocks we normally check whether a variable has a certain value or whether it is greater or smaller than another one.

Also the other two scales are of interest sometimes. Consider for instance the following snippet:

```
if (x < 2 · y){
        ...
}
```

Such an expression is only reasonable if x and y encode ratio values. However, this observation is a consequence of a semantic analysis of the program. On the syntactic level there are basically two terms, namely x and $2 \cdot y$, that are compared.

Another example, that demonstrates the use of nominal values, is given by the following snippet:

```
if (weather = "rainy"){
        ...
}
```

The allowed operations are just $=$ and $\neq$. As demonstrated by these examples, all our query language needs to support is comparing two terms using an operator

$$\circ \in \{=, \neq, <, \leq, >, \geq\}$$

plus a special query of type *else*. We can further restrict this definition to a comparison of single *values* (variables or constants) rather than *terms*, since each term can be stored in a temporary variable before the condition is evaluated.

We come to the conclusion that each condition can be represented as triplet consisting of two values and one operator. This observation was already made by [Quinlan, 1987].

Definition 3.4. A query q consists of two values that are compared using an operator $\circ$:

$$q = (V_1, \circ, V_2)$$

with $\circ \in \{=, \neq, <, \leq, >, \geq\}$ or is of kind:

$$q = else$$

We call such conditions *cmp queries* (*comparison*) and the set of all valid queries, i.e., the query language, Σ_{cmp} (according to Definition 3.3).

Expressiveness of Σ_{cmp}

Note that until now we have completely ignored that conditions can be composed of several sub-conditions that are connected by propositional operators like $\wedge$ and $\vee$. Consider the following example:

Example 3.3. A program with a composed query using an $\wedge$ connective:

```
if (weather = "rainy" ∧ cinema = "closed"){
        go_to_bed();
}
```

However, it can be easily argued that our query language that allows only triplets of the form introduced above is completely sufficient even for composed queries. The snippet can be rewritten such that only queries $\in \Sigma_{cmp}$ occur:

```
if (weather = "rainy")
        if (cinema = "closed"){
                go_to_bed();
        }
}
```

A similar rewriting is possible for negations and disjunctions.

Example 3.4. A program with a composed query using $\neg$ and $\vee$ connectives:

```
if (leg_broken ∨ weather ≠ "sunny"){
        stay_at_home();
}else{
        go_to_beach();
}
```

It is equivalent to:

```
if (leg_broken){
        stay_at_home();
}else{
        if (weather = "sunny"){
                go_to_beach();
        }else{
                stay_at_home();
        }
}
```

Clearly, since $\wedge$ and $\neg$ as well as $\vee$ and $\neg$ are functional complete (i.e., they are sufficient to express any propositional formula), this is also true for all three connectives, and we can rewrite any conditions such that Σ_{cmp} suffices. Therefore we will use exactly this query language in the remaining part of the thesis, even though other languages could be designed that are possibly more user-friendly depending on the concrete application scenario.

This concludes our formal introduction of decision diagrams and we come to the point where we formally define the merging task.

Mathematics may be defined as
the subject in which we never
know what we are talking about,
nor whether what we are saying
is true.
Bertrand Russell

Chapter 4

Task Definition and Variants

In the last section we have introduced decision diagrams and gave some motivating examples. Now we come to the task of merging multiple diagrams.

In the biomedical literature often several decision trees or diagrams for basically the same purpose are given. Probably the suggestions from different sources will coincide in most cases if serious scientific methods and studies have been applied. But there can be differences in detail.

If one has no further information or preferences about the level of trust of the information providers, it is necessary to decide how to merge the *belief bases*, which we call knowledge sources from now on. In general, multiple bases will partially coincide, complement one another or contradict each other. The generation of one consistent decision diagram out of this variety of inputs is the informal description of what we understand under *decision diagram merging*.

4.1 General Definition

Formally we define our merging problem as follows. Let $D_1, D_2, \ldots, D_n$ be decision diagrams over domain $\mathcal{D}$ and classes $\mathcal{C}$ as shown in Section 3.1. All n decision diagrams *should* represent the same underlying reference classification $c_{ref} : \mathcal{D} \to \mathcal{C}$. However, due to different results from different research groups, modifications of existing decision trees, measurement errors, rounding errors or simply human errors, decision diagrams can differ in practice.

We defined $\urcorner_{\mathcal{D},\mathcal{C}}$ to be the set of all possible decision diagrams over domain $\mathcal{D}$ and set of classes $\mathcal{C}$. At the end of the day, an n-ary merging operator is required that combines all D_i into a single D_m (*merged*).

Definition 4.1. An n-ary decision diagram merging operator

$$\circ^n : \underbrace{\urcorner_{\mathcal{D},\mathcal{C}} \times \urcorner_{\mathcal{D},\mathcal{C}} \times \cdots \times \urcorner_{\mathcal{D},\mathcal{C}}}_{n \text{ times}} \to \urcorner_{\mathcal{D},\mathcal{C}}$$

maps n input classifiers (over $\mathcal{D}$ and $\mathcal{C}$) to a new diagram.

Observe that this definition does not yet predetermine *how* the operator actually works, i.e., how it decides which part to take from what diagram. Clearly there can be a large variety of

different properties we desire depending on the application scenario. An intuitively reasonable requirement is:

$$\forall d \in \mathcal{D}, \forall c \in \mathcal{C} : (\forall i D_i(d) = c) \Rightarrow \circ^n(D_1, D_2, \ldots, D_n)(d) = c$$

Informally, this expression states that if all decision diagrams agree upon the classification of an element d, then this classification should be unchanged after the merging process.

Note that this is not a *hard* requirement, but just one of many possible properties that a merging operator can have. One could also argue that a more powerful optimization is possible only if the requirement is dropped and the final diagram may deliver a different answer for a certain element, even if the input classifiers agree upon its class.

4.2 Task Variants

The previous sections introduced the task definition for the most general problem, namely the merging of decision diagrams without any further assumptions. If the problem is investigated in detail, several different but related tasks appear. Now we want to look closer at the problem and formally define the possible variations of the task.

Trees versus Diagrams

First of all, when reading related biomedical literature, one soon recognizes that many authors talk about decision *trees* rather decision *diagrams* (e.g., [Mair et al., 1995] or [Althoff et al., 1998]). Even though diagrams are more general, since trees are a special case of diagrams, it seems intuitively plausible that the work with trees is somehow simpler than with diagrams. To illustrate this, one supporting argument is that many graph algorithms are much easier to apply on trees, or graphs that are *almost trees*, than on general graphs. This is what the term *treewidth* is used for, see for instance [Bodlaender, 1993].

However, not only the algorithmic runtime is important. Since medical guidelines are often made for humans, they are preferred over diagrams since they are simply easier to read and understand.

If we restrict the merging task to trees, the formal definition given in Section 4.1 essentially remains the same with the only difference that the arguments and the return value of an operator will now come from the set of all trees over $\mathcal{D}$ and $\mathcal{C}$ rather than from the set of general directed acyclic graphs. Formally, let

$$\circ^n : \underbrace{\mathcal{T}_{\mathcal{D},\mathcal{C}} \times \ldots \times \mathcal{T}_{\mathcal{D},\mathcal{C}}}_{n \text{ times}} \rightarrow \mathcal{T}_{\mathcal{D},\mathcal{C}}$$

be the operator, where $\mathcal{T}_{\mathcal{D},\mathcal{C}}$ is the set of all decision trees over domain $\mathcal{D}$ and classes $\mathcal{C}$.

This idea can even be refined. We can define operators that take arbitrary diagrams as input but ensure that the result is a tree. In this case, the operator would include a kind of *simplification* procedure. Or the other way round, they require the inputs to be trees, but deliver general diagrams.

Node Degrees

Sometimes it is desirable to restrict the maximum number of outgoing edges for inner nodes. This makes the implementation of many algorithms much easier, as we will see in Chapter 6. Additionally such trees are in general easier to understand.

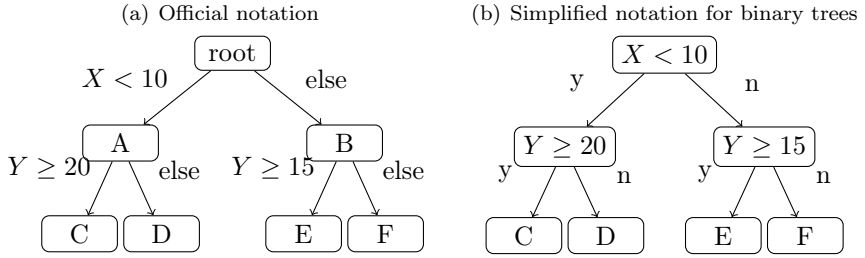

Figure 4.1: Comparison of notations

A straightforward restriction is therefore to work only with *binary* trees (i.e., each node has at most two children). Especially if one additionally assumes that each node has exactly one *conditional* outgoing edge (the other one is an *else* edge), it is obvious that the alternatives are disjoint and exhaustive.

Ordering

Now we consider a criterion that cannot be checked by looking at individual nodes or edges, but which concerns a path from the root to a leaf. Further assume that we work with binary trees only, where each inner node has one conditional and one else edge as discussed above.

Then we can say that in each inner node one certain variable is queried, though strictly speaking, the conditions belong to the edges rather than the nodes (see Definition 3.3). In this case the edges are labeled with *yes* or *no*. This is illustrated with the following example.

Example 4.1. The binary tree in Figure 4.1(a) can also be drawn as in 4.1(b). This is against our official definition, but this point of view simplifies the explanation below.

When the queries are thought of as node labels one can talk about the *order* in which the variables are queried on a certain path through the tree. In the above example, on the path $root \to A \to D$ the variables are evaluated in order $X \to Y$.

Now a very natural requirement is that the variables are requested in a predefined order on *every* path through the diagram. If this condition is satisfied by a diagram we call it *ordered*, otherwise it is *unordered*. This is similar to *Ordered Binary Decision Diagrams (OBDDs)* (see [Bryant, 1992]), which may simplify the merging later on but enlarges the tree in general.

Consider Example 4.2 where X needs to be queried prior to Y on every path through the tree (if it is evaluated at all). The diagram in Figure 4.2(a) is unordered. Figure 4.2(b) shows a semantically equivalent but ordered decision tree.

Example 4.2. Figure 4.2(a) shows an unordered tree that is transformed into an equivalent ordered one in Figure 4.2(b). The variable ordering is $X \to Y$.

Each non-ordered tree can be converted into an semantically equivalent ordered one (see for instance [Fujita et al., 1991] for an overview about ordering algorithms). While in the previous example the size of the tree stayed the same, note that in general reordering to a given variable ordering enlarges the tree exponentially since subtrees may need to be copied. Is is again stressed that this is true for the case that a *predefined* variable ordering must be achieved (as this is the case if one wants to make the ordering in two trees equivalent). There also exists an optimal ordering for each tree with respect to it's size, and of course the tree will get smaller or stays

equivalent if a transformation to this ordering is made (however, finding the optimal ordering is NP-complete [Bollig and Wegener, 1996]).

We will come back to this point in the next chapter where a formal operator is defined that implements a reordering algorithm.

4.3 Contradicting Diagrams

In the general part of this chapter we said that a merging operator maps n input decision diagrams onto a new one. The only requirement on this operator until now was that the output diagram should deliver the same classification of a domain element, if the input diagrams agree upon its class (and even this was not an absolutely necessary condition, as we said).

We illustrated conflicting diagrams in the example in the last subsection. But what we skipped so far is a formal definition of conflicts.

Example 4.3 shows a typical conflict situation. The trees are obviously similar but not equal. Domain element $(X = 8, Y = 12)$ is classified as C_2 by the tree in Figure 4.3(a) and as C_1 by the one in 4.3(b). And this is exactly the characterization of conflicts, namely domain elements that get different classifications, depending on the diagram in usage.

Example 4.3. The decision diagrams in Figures 4.3(a) and 4.3(b) are contradicting for all input tuples

$$\{(X, Y) \in \mathbb{R}^2 \mid X < 10 \wedge 10 < Y \leq 15\}$$

We will refer to this first example during the sections in the next chapter to illustrate the semantics of the proposed operators.

Definition 4.2. Two diagrams D_i and D_j over domain $\mathcal{D}$ and classes $\mathcal{C}$ are said to be *contradicting* iff

$$\exists d \in \mathcal{D} : D_i(d) \neq D_j(d)$$

Definition 4.3. A set of diagrams $\{D_1, \ldots, D_n\}$ is contradicting, iff it contains at least two contradicting diagrams.

Definition 4.4. The *conflict set* of a set of diagrams $\Delta = \{D_1, \ldots, D_n\}$ is defined as

$$\gamma(\Delta) = \{d \in \mathcal{D} \mid \exists i, j : D_i(d) \neq D_j(d)\}$$

4.4 Summary

Table 4.4 shows the attributes introduced in the previous sections. In theory, all combinations (and probably much more variants) of these attributes are possible, even if some of them are either unimportant in practice or difficult to realize.

Input graph type	general directed acyclic graph, tree
Output graph type	general directed acyclic graph, tree
Input node ordering	arbitrary, ordered
Output node ordering	arbitrary, ordered

Table 4.1: Task attributes

Observe that it would be necessary to implement merging operators suitable for the most general task version in order to solve all attribute combinations. That is, we have to implement the merging operators for *arbitrarily ordered decision diagrams*. This can be very tough since we cannot make any restricting assumptions about the input. Therefore it seems easier to solve the problem stepwise.

Instead of solving very general and complicated variants, it might be easier to reduce tricky instances to simpler ones. For instance, we could translate decision diagrams into decision trees (which is possible, as we will see) and merge them. This requires two operators to be applied in sequence. First we apply a unary *unfolding* operator on the input diagrams, then we use a n-ary merging operator to incorporate the resulting trees.

The same idea can be applied to the case of unordered decision trees. We could introduce a unary operator for ordering. Subsequently we can safely assume that all input diagrams are not only trees but are additionally ordered. If this is not the case, we can convert them using the unary operators.

This concludes the the formal basics we are going to use later on. We will shortly come back to the characterization of different task variants at the end of the next chapter where we will see that all variants can be solved by the proposed set of operators. Now we are going to take a closer look at the operators and introduce conflict resolution strategies.

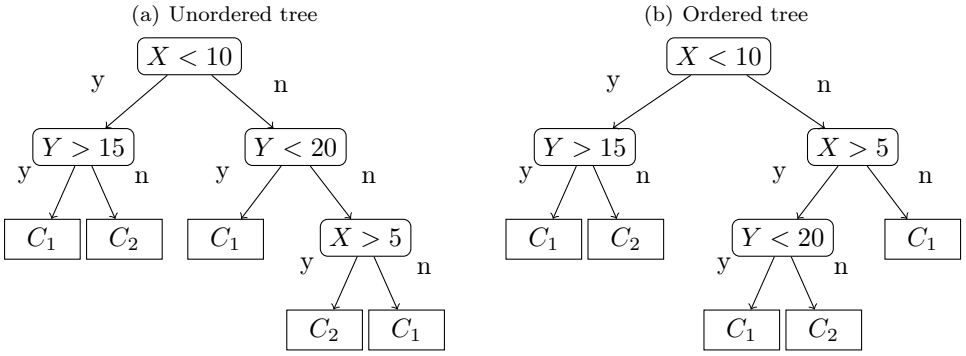

Figure 4.2: Comparision of ordered and unordered trees

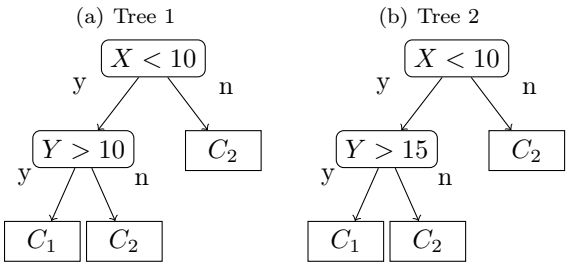

Figure 4.3: Conflicting trees

I mean the word proof not in the sense of the lawyers, who set two half proofs equal to a whole one, but in the sense of a mathematician, where half a proof is zero, and it is demanded for proof that every doubt becomes impossible.

Carl Friedrich Gauß

Chapter 5

Formal Operator Definition

In the previous chapter we argued that there exist several versions of the initially presented task description. The differences concerned the assumptions about the input diagrams. We further have defined conflicts formally, but what we have not discussed until now is what to do in case of disagreement.

In Example 4.3, all elements in the conflict set $\{(X < 10, 10 < Y \leq 15)\}$ can essentially be arbitrarily classified by the merged tree. However, in a certain use case there may be a desired solution other than random classification.

Maybe one has more trust in one of the two trees and gives precedence to it. For instance, one could imagine that a more restrictive condition is more trustful than a general one, since the original author may had better evidence if a special case was investigated rather then if the diagram comes from a fairly general study, where the class could be simply the result of a *classification-as-failure*.

Another idea is that one wants to use some kind of average operation, such that each tree can assert its classification in half of the cases. This is discussed in detail in Section 5.2. Thus, the next task and the topic of this chapter is the evaluation of possible conflict resolution strategies.

Preliminary remark: The operators in discussion are intended for showing the potentials of the framework and serve only as *examples*. Therefore they will not always be implemented in detail. It is up to the user to select among these operators and adapt them or implement new ones that satisfy the individual needs. Thus, some of the assumptions that will be made can and should be put into question, depending on the application in mind. We will show some application scenarios in Chapter 7.

5.1 Unary Modification Operators

We are now going to introduce a few unary operators that do not merge several but only modify single diagrams. They will simplify the work for the second main section of this chapter, where actual merging operators will be shown.

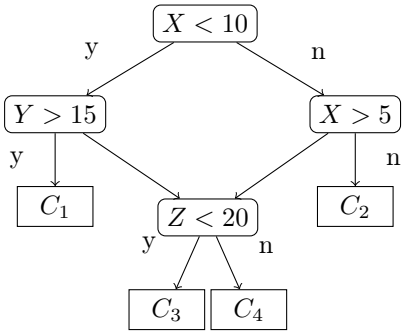

Figure 5.1: Decision diagram with node sharing

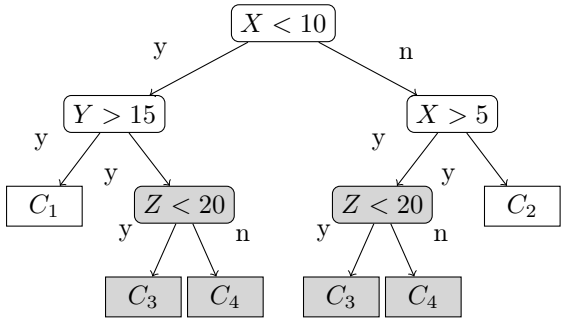

Figure 5.2: Decision diagram from Figure 5.1 without node sharing

Making Trees out of Diagrams

It is intuitively plausible that working with trees is easier than with diagrams. Therefore we introduce a unary operator for this purpose and call it $\circ_{uf}$ (*unfold*). The idea of *unfolding* decision diagrams can best be explained with Example 5.1.

Example 5.1. The input diagram in Figure 5.1 contains two nodes $Y > 15$ and $X > 5$ that share a common subnode $Z < 20$.

In the unfolded diagram in Figure 5.2, the common subnode needs to be duplicated such that each parent gets its own copy.

It is straight forward to convert this diagram into a tree. We "simply" have to duplicate the common subtree. The nodes that shared a successor node got separate copies of it (gray background). This procedure needs to be applied recursively because common subtrees can contain nodes that are again involved in another node sharing. Of course this may lead to an exponential overhead in the worst case.

Definition 5.1. The unary unfolding operator $\circ_{uf}$ is defined as follows.

$$\circ_{uf}(D)(d) = D(d)\ \forall d \in \mathcal{D} \text{ and } \circ_{uf}(D) \text{ is a tree.}$$

Input: $D \in \urcorner_{\mathcal{D},\mathcal{C}}$ with root node $r \in V$
Output: $D' \in \urcorner_{\mathcal{D},\mathcal{C}}$ with $D(d) = D'(d)\ \forall d \in \mathcal{D}$ and D' is a tree
if $r \in \mathit{leaves}(V)$ **then**
  return *new Node(r.classification)*
else
  `/* make a copy of` r `*/`
  $r_{new} = r$;
  `/* for all outgoing edges` e `*/`
  for $e = (c, r, \mathit{subnode})$ **do**
    `/* recursively unfold the diagram */`
    $\mathit{subnode}_{new}$ = unfold($\mathit{subnode}$);
    add edge from r_{new} to $\mathit{subnode}$ with condition c;

Algorithm 2: Computation of operator $\circ_{uf}$

Proposition 1. $\circ_{uf}$ can be computed by Algorithm 2, which runs in $O(\delta^{|V|})$ where δ is the maximum out-degree and $|V|$ the number of nodes in the input diagram.

Proof sketch. Correctness. It can be easily seen that the algorithm basically performs a depth-first search (DFS), bundled with a copying of the graph. Since each visit of a node is copied separately, it is clear that the resulting graph does not contain any shared nodes, i.e., it is a tree.

The semantic equivalence of the input and the output diagram is obvious.
Termination. The only loop in the algorithm iterates over the edges, which is a finite set. Thus the algorithm terminates for sure.
Complexity. Proof by induction on the number of nodes in the diagram. If our diagram is a single leaf node, the runtime is obviously restricted by $O(\delta^{|V|}) = O(1)$.

Induction step: Our diagram consists of $|V|$ nodes with a root r. Each of the root's child nodes is the root of a sub-diagram consisting of at most $|V| - 1$ nodes. A recursive call of the algorithm on one of these sub-diagrams is possible in $O(\delta^{|V|-1})$ by induction hypothesis. Since the maximum out-degree of r is δ, we have at most δ such calls, leading to complexity $O(\delta \cdot \delta^{|V|-1}) = O(\delta^{|V|})$. □

Basically the algorithm works as follows. If a leaf node is observed we just make a flat copy since it is unfolded anyway, i.e., it is already a legal tree. In case of inner nodes we recursively call the method for each child.

Note that there are several possibilities for implementing $\circ_{uf}$. The given algorithm is just one of them, but it is a straightforward solution.

Binary Decision Trees

At this point we are able to translate each general decision diagram into a tree. Now we make another simplification. A tree that has nodes of arbitrary out-degree is still very inconvenient for later sections. Thus we will show now how we can convert an arbitrary tree into a binary one.

The underlying idea is simple. In a general tree we can have a case distinction with up to n outcomes in each inner node. To reduce this number to two, we simply just check one of them and redirect the else edge to a new intermediate node, where the next condition is checked. This process reduces the number of cases by 1 in each iteration. It is continued recursively until we finally have checked all conditions. This needs n steps (we further assume that each node must have one conditional and one else edge; if the last node is allowed to have two conditional edges, $n - 1$ steps are sufficient).

The procedure is illustrated by Example 5.2.

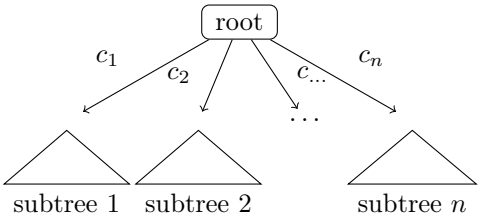

Figure 5.3: An n-ary tree

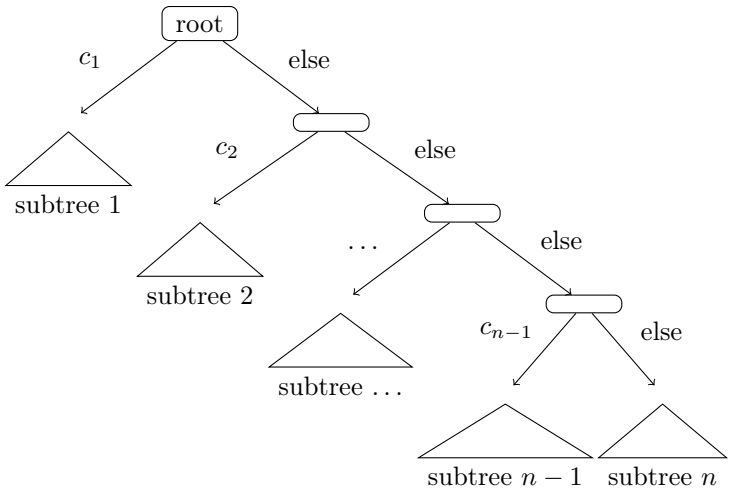

Figure 5.4: Binary version of the tree in Figure 5.3

Example 5.2. The n-ary decision tree in Figure 5.3 can be converted into the binary one in Figure 5.4

This leads to the following formal definition.

Definition 5.2. The unary operator $\circ_{tb}$ (*to binary*) is defined as:

$\circ_{tb}(D)(d) = D(d)\ \forall d \in \mathcal{D}$ and $\circ_{tb}(D)$ is binary decision tree, i.e., its maximum node degree is 2.

Remark: Clearly, one could design algorithms for creating of balanced decision trees, which are computationally advantageous. But since our operator serves only as example anyway we skip this.

Proposition 2. $\circ_{tb}$ can be computed in $O(|V| + |E|)$ by Algorithm 3.

Proof sketch. Correctness. We show that Algorithm 3 indeed generates a binary decision tree from a general one. The proof is done by induction on the depth of a tree.

Induction basis: Leaf nodes, which are processed in the outer if-block, are trivially binary trees.

Induction step: All sub-trees of an arbitrary inner node v have a depth which is smaller than the depth of the tree with root v, denoted as T. Therefore they are binary trees after the recursive application of the algorithm by induction hypothesis.

Input: $T \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with root node $r \in V$
Output: $T' \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with $D(d) = D'(d)\ \forall d \in \mathcal{D}$ and T' is a binary tree
if $r \in \mathit{leaves}(V)$ **then**
  return r
else
  if r *has more than two subtrees* **then**
    Let s_1 be r's first subtree with condition c_1;
    $s_2 = \mathit{toBinary}(T \setminus s_1$ with root $r)$;
    return *new Node*$(c_1,\ s_1,\ s_2)$
  else
    return r

Algorithm 3: Computation of operator $\circ_{tb}$

We need to show that also T is a binary tree after the algorithm was applied. We have two cases: (i) T was a binary tree even prior to the execution and (ii) T has out-degree greater than 2.

(i) The algorithm will execute the else-branch within the outer else-block, where the node is returned unmodified. Thus it still has degree ≤ 2 after the run.

(ii) The only case that requires modifications concerns inner nodes with more than 2 sub-nodes. In this case, the input diagram simultaneously checks conditions $c_1, \ldots, c_n$. We first check only condition c_1 and redirect the else-branch of the node to a new intermediate node v', where the other conditions $c_2, \ldots, c_n$ are checked.

Note that v' is the root of a tree T' with *the same* depth as T, but the degree of v' is smaller by 1 than of v. If this algorithm is applied recursively on v', it is easy to see that we will finally end up in one of the trivial cases, i.e., either the induction hypothesis (leaf node) or in (i).

Now we show that the resulting diagram is semantically equivalent to the input diagram. Suppose we end up in sub-tree n in our input tree for some element e. Then e satisfies only c_n but none of c_i with $i \neq n$ by our precondition in Section 3.1 that forces a unique result for all domain elements. But then we end up in the same sub-tree in the modified diagram, since all c_i with $i < n$ are unsatisfied, which makes us following the else-edges. This can be seen in Figure 5.4.

Termination. The algorithm does not contain any loops, the recursion will eventually end since the degree of a newly introduced node is always smaller than of the original node. Therefore the algorithm will always terminate.

Complexity. Obviously, each node which was already contained in the input diagram is visited exactly once (DFS). Since each newly generated node decreases a (formerly too high) node degree by 1, its number is bounded by the sum of the node degrees exceeding 2. Thus, $O(|V| + |E|)$ suffices in the worst case. □

Leaf nodes and inner nodes with up to two sub-nodes (actually *exactly* two for a reasonable diagram) do not need to be modified. In case of more than two conditions, the algorithm recursively splits off one after the other by introduction of intermediate nodes.

As in the definition of the unfolding operator, the algorithm does not need to be part of the definition since there exist other strategies that will also produce binary trees. Even though this is probably one of the most straightforward ones, it may be more desired to make the procedure deterministically, i.e., eliminating the *random* selection of the first condition to test. More about such technical details are depicted in Section 6.

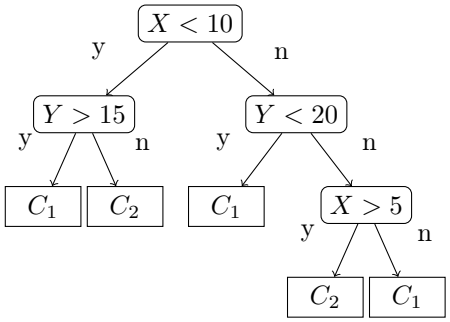

Figure 5.5: An unordered decision tree

Ordering Trees

Next we will consider how we can order arbitrarily ordered decision trees. We assume the input to be a binary tree. The presented approach is similar to [Fujita et al., 1991] in the sense that the global ordering is obtained by local exchanges of two neighbored variables. But in contrast to the algorithm in [Fujita et al., 1991], we will *not* distinct several cases depending on the local structure of the diagram (e.g., the number of children of the exchanged nodes). This makes the exchange procedure a bit more complicated, but calling it recursively becomes simpler.

First we define ordered decision trees formally.

Definition 5.3. Let $T = \langle V, E, C, l_c, l_e \rangle$ be a binary decision tree with root $r \in V$ and let $[v] = x$ denote the (unique) variable x that is queried in inner node $v \in V$. Then the tree is called ordered, iff for each path from the root to a leaf node $r, v_1, \ldots, v_n$ ($v_n \in \mathit{leaves}(V)$), the following holds:

$$[r] <_l [v_1] <_l \cdots <_l [v_{n-1}]$$

where $<_l$ denotes the lexical ordering of the variables.

Example 5.3. Figure 5.5 shows an example where variable Y is queried prior to X on a path through the tree. Thus the tree is *not* ordered. A semantically equivalent ordered tree is depicted in Figure 5.6. Gray elements show parts of the original (unordered) tree that have been removed. Dotted arrows show where certain parts of the final tree come from.

If we want to order a tree, we clearly have to change the variable query order somehow. But a simple node exchange without redirecting affected incident edges is not possible. Look at Figure 5.6.

On the path from the root towards the leaf nodes we first simply skip the node that queries variable Y. This means, the new *no*-child of the root is the node $X > 5$. However, after this query has been evaluated, we can not simply return one of the classes C_1 or C_2 since we ignored that variable Y is involved in the decision. We have to keep in mind that the result of this node is C_2 (in the *yes* subtree) or C_1 (in the *no* subtree) provided that $Y < 20$ evaluates to no (which has been skipped). Let's call this the *conditional result* of $X > 5$.

Therefore we need to catch up the formerly skipped query before a final decision can be made. The node that queries Y including its *yes* subtree needs to be inserted into both branches. In case that this condition evaluates to *no* (which was the precondition for our conditional result) we finally can fix and return the conditional result. In case that this condition evaluates to *yes*, our conditional result is irrelevant since $X > 5$ would have never been evaluated in the original tree and we can safely return C_1 in both subtrees.

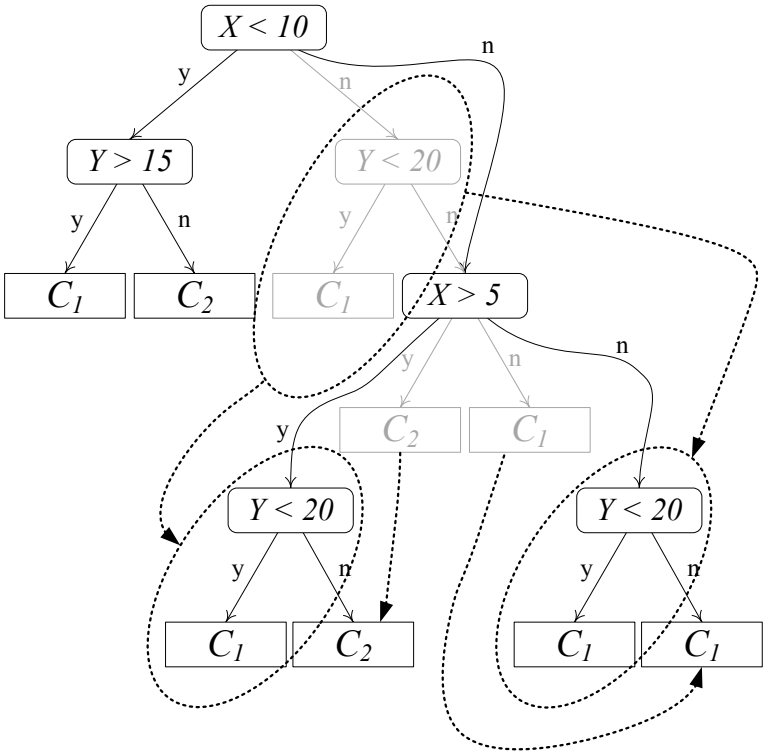

Figure 5.6: Ordered version of the tree in Figure 5.5

Note that the skipped node $Y < 20$ and its *yes* subtree needs to be copied for each branch of the node $X > 5$ in order to make sure that the tree actually remains a *tree* rather than a diagram. This, of course, again leads to an exponential increase in size in the worst case.

The two algorithms referenced in Definition 5.4 show a procedure that orders a tree according to the lexical ordering of the variables. Figure 5.7 shows an illustration of this algorithms.

Definition 5.4. The unary order operator $\circ_{ord}$ (*node ordering*) is defined as:

$$\circ_{ord}(T)(d) = T(d)\ \forall d \in \mathcal{D} \text{ and } \circ_{ord}(T) = \langle V, E, C, l_c, l_e \rangle \text{ with root } r \text{ s.t.}$$
$$[r] <_l [v_1] <_l \cdots <_l [v_{n-1}]\ \forall \textit{paths}\ r, v_1, \cdots, v_n\ (v_n \in \textit{leaves}(v))$$

Proposition 3. $\circ_{ord}$ can be computed by calling the procedure shown in Algorithm 4 and its sub procedure in Algorithm 5. The complexity is $O(2^n)$.

Proof sketch. Correctness. We show that Algorithm 4 orders the nodes of the diagram according to their lexicographic ordering. Suppose the nodes are not ordered after the execution of the algorithm. Then there exists a node v and a sub-node w s.t. $[w] <_l [v]$. But in this case, the algorithm would have exchanged the two nodes during sinking of v, which shows that the situation cannot occur.

The semantic equivalence of the input and the output diagram can be shown as follows. Let v be an arbitrary leaf node of the input diagram and $C = \{c_1, \ldots, c_n\}$ the set of conditions that need to be satisfied to end up in node v. It can be easily verified by looking at Figure 5.7

Input: $T \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with root node $R \in V$
Output: $T' \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with $T(d) = T'(d)\ \forall d \in \mathcal{D}$ and T' is an ordered tree
if $R \notin leaves(V)$ **then**
  `/* R is an inner node, iterate through all child nodes */`
  for $(R, u) \in E$ **do**
    remove edge (R, u);
    $order(T$ with root $u)$, let u' be its new root;
    add edge (R, u') with the same condition as before;
  $sink(T$ with root $R)$, let R' be its result;
  return R'

Algorithm 4: order

Input: $T \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with root node $r \in V$
Output: $T' \in \mathcal{T}_{\mathcal{D},\mathcal{C}}$ with $T(d) = T'(d)\ \forall d \in \mathcal{D}$ and T' is an ordered tree
if R *needs to be exchanged with one of its children* **then**
  Let *exchange* be the child with the lexically smallest variable queried (condition: c_{ex}) and *sibling* the other child (condition: c_{sib}); let further $resttree1$ and $resttree2$ be the subtrees of *exchange* (with conditions c_{rest1} and c_{rest2} respectively);
  $newroot = exchange$;
  remove all edges from *root* (old root) to its successors and predecessors;
  remove all outgoing edges from *exchange*;
  make copies of *sibling* and *root*, denoted with primes($'$);
  connect *newroot* to *root* and *root*$'$ with conditions c_{rest1} and c_{rest2};
  connect *root* to *sibling* and $resttree_1$ with conditions c_{sib} and c_{ex};
  connect *root*$'$ to *sibling*$'$ and $resttree_2$ with conditions c_{sib} and c_{ex};
  $rootsuccessor_1 = sink(T$ with $root)$;
  $rootsuccessor_2 = sink(T$ with $root')$;
  connect *newroot* with $rootsuccessor_1$ and $rootsuccessor_2$;
  return n
else
  return R

Algorithm 5: sink (see reference Figure 5.7)

that the satisfaction of the same conditions in the output diagram will lead to a leaf node with the same label, independently from the (possibly unnecessarily checked) additional conditions. This is true for all leaf nodes v. Therefore a single exchange of neighbors does not modify the semantics of the diagram.

By induction, which corresponds to the recursive calls in the algorithm, it can be proved that the algorithm as a whole does not change the semantics either.
Termination. Algorithm 5 implicitly contains a loop that iterates over the children of a node, but this number is clearly finite. Algorithm 4 iterates both over the edges and the nodes of the diagram, but this obviously terminates as well.
Complexity.

Lemma 1. Procedure *sink* has complexity $O(2^{|V|+L})$ where L is the length of the longest variable name used in the decision diagram.

Proof of Lemma 1. We first prove that subprocedure *sink* has complexity $O(2^{|V|+L})$. This is done by induction on the number of nodes. Clearly, for a leaf node ($|V| = 1$) the procedure

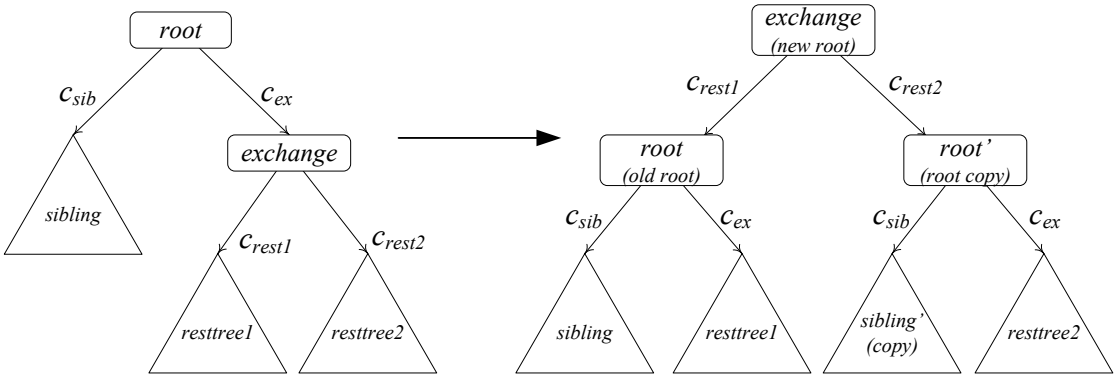

Figure 5.7: Illustration of the operator from Definition 5.4

immediately terminates (induction basis).

In case of an inner node we need to determine the lexicographically smallest of its two child nodes, which is possible in time $O(L)$. We further make copies of some subgraphs, which runs in $O(|V|)$ (actually $O(|V| + |E|)$, but all node degrees are equal to 2 by assumption, restricting the number of edges to $O(|V|)$).

The algorithm calls itself recursively on the child nodes of the root, which have at most $|V| - 1$ nodes. The complexity of those two calls is in $O(2 \cdot 2^{(|V|-1)+L}) = O(2^{|V|+L})$ by induction hypothesis.

Therefore *sink* has an overall complexity of $O(L) + O(|V|) + O(2^{|V|+L}) = O(2^{|V|+L})$. □

The proof of the complexity of the ordering algorithm is again done by induction on the number of nodes in the diagram. For the induction basis ($|V| = 1$), the diagram consists of a single leaf node. Then the complexity is clearly restricted by $O(2^{|V|+L}) = O(1)$.

Induction step: Our diagram consisting of $|V|$ nodes has root r with two sub-diagrams S_1 and S_2. First, the algorithm removes an edge (constant time), before it calls itself recursively within the for loop. Clearly the number of nodes of S_1 and S_2 is at most $|V| - 1$. Thus, by induction hypothesis, the application of the algorithm on those diagrams is bounded by $O(2 \cdot 2^{(|V|-1)+L})$. The insertion of the new edge is again possible in constant time. The loop runs exactly 2 times by assumption that the input tree is binary.

Finally, the algorithm makes another call of *sink*. This requires $O(2^{|V|+L})$ by Lemma 1. Therefore the overall complexity is $O(2^{|V|+L})$. □

Remark. In real world applications L is usually constant. If the decision diagram is trained by machine-learning algorithms, the dimensionality of the feature vector is fixed a-priory. But then we can also assign fixed variable names to the dimensions. Consequently, also the maximum length of variable names, L, is constant.

Eventually it boils down to a reordering of the nodes such that conditions c_{rest1} and c_{rest2} are *always* (on every path) evaluated before c_{sib} and c_{ex}. As demonstrated in Figure 5.5, we simply skip c_{sib} and c_{ex} first and go directly to the evaluation of c_{rest1} and c_{rest2}. This brings us either to node *root* or to *root*$'$ (a copy of *root*). At this point we have the conditional result *resttree*1 in *root* and the conditional result *resttree*2 in *root*$'$. Both results are only correct if c_{ex} evaluates to *true*. Thus, in order to compute the final result, we need to check this condition

now. Then we either go into the according rest tree or forget the conditional result and evaluate *sibling* (in this case the interpretation of the conditions c_{rest1} or c_{rest2} was unnecessary).

Since the output tree is exponential in the input, there is no hope for an efficient (polynomial) algorithm. Nevertheless there have been algorithms developed that are at least polynomial in the product of the input and the output size [Bern et al., 1996], [Tani and Imai, 1994]. Some of them work by local exchanges of variables (like the algorithm above), others use a global rebuilding strategy.

Note the similarity of this algorithm with Heap-sort which was presented in [Williams, 1964]. And in fact, the idea of sinking inner nodes in reverse order until they have no "larger" child (according to the variable in the condition) has been inspired by this sorting algorithm. However, the runtime is very different. While Heap-sort has a worst-case complexity of $O(n \cdot log(n))$, the suggested procedure for tree ordering is exponential. This comes from the fact that subtrees have to be duplicated during node exchange in order to keep the semantics equivalent. This is the crucial difference to Heap-sort.

5.2 Merging Operators

After these simplifications steps we come to the actual merging operators. While some of them can be applied to any decision diagrams, advanced operators like *average* are much easier to implement if we assume that the input diagrams are ordered binary trees. If this is not the case, the unary operators from the previous section are applied first to satisfy these preconditions.

User-Preferences

One merging approach that was already shortly mentioned and that is very simple from a programmers point of view is the usage of user preferences. In this case, not the algorithm but the *user* decides what to do in case of inconsistency. This is inspired by the field of *social choice theory* [Dasgupta et al., 1979], though a difference is that we assume that the user specifies globally valid preferences rules rather than a separate set for each agent.

The underlying idea is simple. In many applications, not all wrong decisions are likewise serious. One wrong decision can cause higher or lower costs than another one. For instance, in medical screening tests, a *false positive* is usually much less serious than a *false negative*. While in the former case, a second and more precise (and more expensive) test will just reveal that the patient is in fact healthy, the latter will have the consequence that a disease is undiscovered for a longer period of time and can progress in the meantime.

In such cases it is easy to argue that one of the possible decisions is preferred over the other one in case of doubt. In the context of decision diagram merging, this means that the algorithm will chose the preferred classification if the sources are inconsistent. This is demonstrated in Example 5.4.

Example 5.4. Imagine there are two possible classifications C_1 and C_2 and two classifiers D_1 and D_2 s.t. $D_1(d) = C_1$ and $D_2(d) = C_2$ for some domain element d. The final decision will depend on the user ranking of classes C_1 and C_2. So for instance if $C_1 > C_2$ (our notation for "C_1 is preferred over C_2"), the final decision will be C_1.

We can push this idea further. Let's assume that a third belief source D_3 with $D_3(d) = C_2$ is added to the last example. Then we still have a contradiction and we still prefer C_1 over C_2. However, the number of votes for the different classes have changed. While we had $1 : 1$ with two belief bases, we now have $1 : 2$, and thus it has become more probable that C_2 is the correct decision. This could be an additional criterion to check.

In general we can say that some classification C_i is preferred over another one C_j, *if* the voting difference is equal or higher than some treshold n:

$$C_i > C_j \text{ if } |C_i| - |C_j| \geq n,$$

where $|C|$ is our notation for the number of votes for a class C. We will denote such a *user preference condition* as follows:

$$C_i >^n C_j$$

The previously introduced idea of preferring C_i over C_j *in any case* will be denoted as:

$$C_i \gg C_j$$

Definition 5.5. A *user preference condition* over classes $\mathcal{C}$ is a tuple

$$C_i \gg C_j$$

or a triple

$$C_i >^n C_j$$

with $n \in \mathbb{N}$ and $C_i, C_j \in \mathcal{C}$

Such a definition seems natural, but the implementation and evaluation can be tricky. First, we have to make sure that the user preferences are consistent in all cases. For instance, the set of conditions

$$\{C_1 \gg C_2, C_2 \gg C_3, C_3 \gg C_1\}$$

is obviously inconsistent in all cases where we have votes for at least two different classes. Because of cycles in this set, each choice is inferior to some other possibility and thus there does not exist a "best" classification. The second problem is somehow the contrary of inconsistent preference rules. While contradicting rules are "too strong" since they kill all possible classifications, we can also construct examples where more than one possibility survives. For instance

$$\{C_2 \gg C_1, C_3 \gg C_1\}$$

will not deliver a unique result in case of $|C_1| = |C_2| = |C_3| = 1$. We just know that either C_2 or C_3 should be selected, but not which of them. One could easily construct other, less obvious contradicting examples. Especially conditional preference rules can quickly become confusing.

Both phenomena introduce new troubles. This problem is somehow related to common problems of *voting systems*. For instance, the *Condorcet paradox* is an example where several voters, each delivering a consistent set of preference rules, lead to cyclic preferences when united [Gabbay et al., 2009]. (The connection to to our problem can be seen if we assume each of our preference rules to come from a different voter.)

To avoid problems like these, and to keep things simple for now, a straightforward solution (which was chosen by us) is to use *sequences of rules* rather than *sets*. This is also very natural from a programmer's point of view since it is an algorithmic formalism. We just start with the lexically smallest of all possible classifications (which is as good as any other startup criterion) and then evaluate one of the user's preference rules after the other, where each overwrites the result of the previous one iff it is applicable. This works similar to the way like *access control lists* (for instance in file systems or computer networks) are processed. The procedure is illustrated by example 5.5.

Example 5.5. Let

$$R = \langle C_2 \gg C_1, C_3 >^2 C_2, C_4 \gg C_2 \rangle$$

be our sequence of rules. Assume the voting results are

$$|C_1| = 2, |C_2| = 1, |C_3| = 2, |C_4| = 0$$

Then the algorithm starts with the smallest of the classifications of C_1, C_2 and C_3 (C_4 is discarded since there are no votes for this class), which is C_1. The first rule is applicable since C_2 is *always* preferred over C_1 (also in case that the total number of votes is smaller), changing the intermediate result. The next rule, $C_3 >^2 C_2$ is not applicable, since the voting difference between C_3 and C_2 is 1 and thus less than 2. Also the last rule, $C_4 \gg C_2$ is not applicable. C_4 *would* be preferred over C_2, but none of the classifiers voted for this class.

This leads to the following formal definition.

Definition 5.6. The operator $\circ_{up}$ (*user preferences*) is defined as

$$\circ_{up} : (\urcorner_{\mathcal{D},\mathcal{C}})^n \times R \to \urcorner_{\mathcal{D},\mathcal{C}}$$

where R is a sequence of preference rules of form

$$R = \langle r_1, \ldots r_k \rangle$$

with $r_i \in \mathcal{C} \times \{\{\gg\} \cup \{>^n \mid n \in \mathbb{N}\}\} \times \mathcal{C}$.

The result of an operator application

$$\circ_{up}(D_1, \ldots, D_n) = D_{up}$$

is a new decision diagram D_{up} s.t. its classifications correspond to the result of Algorithm 7.

Input: source classifiers $D_1, \ldots, D_n \in \urcorner_{\mathcal{D},\mathcal{C}}$, rule sequence R, domain element $d \in \mathcal{D}$
Output: classification $D_{up}(d) \in \mathcal{C}$
let c be the lexically smallest $c \in \mathcal{C}$ s.t. $\exists i : D_i(d) = c$;
for $r \in R$ **do**
  if $c = C_j \wedge r = C_i \gg C_j$ *or* $r = C_i >^n C_j \wedge |C_i| - |C_j| \geq n$ **then**
    $c = C_i$;
return c

Algorithm 6: Definition of operator $\circ_{uf}$

Note that this procedure describes only one of the possibilities how we could implement a user-preference operator. Many details, like the syntax and semantics of the preference rules could be discussed and modified. But this is strongly application dependent (the same is true for the other operators that will be described). This chapter is intended to show the possibilities and demonstrate them with examples.

Observe that Algorithm 6 only *defines* the user-preferences operator. It is applied when a domain element d needs to be classified. But it does not provide a procedure that actually computes the resulting diagram $D_{up} = \circ_{up}(D_1, \ldots, D_n)$. This is done by Algorithm 7.

Input: source classifiers $D_1, \ldots, D_n \in \urcorner_{\mathcal{D},\mathcal{C}}$, rule sequence R
Output: result of $\circ_{up}$ when applied on $D_1, \ldots, D_n$
$S = \{D_1, \ldots, D_n\}$;
while $|S| > 1$ **do**
  Insert $S.D_{|S|}$ into each leaf node of $S.D_{|S|-1}$;
  $S = S \setminus \{D_{|S|}\}$;
$R = S.D_1$;
for *all leaf nodes* v_l *in* R **do**
  Let $P = (v_1, \ldots, v_l)$ be the path from the root to v_l;
  for each class label $c_i \in \mathcal{C}$, count the number of votes $|c_i|$ for c_i along P;
  let c be the lexically smallest c_i s.t. $|c_i| > 0$;
  for $r \in R$ **do**
    if $c = C_j \wedge r = C_i \gg C_j$ *or* $r = C_i >^n C_j \wedge |C_i| - |C_j| \geq n$ **then**
      $c = C_i$;
  Set the class label of v_l to c;

Algorithm 7: Computation of operator $\circ_{uf}$

Proposition 4. Algorithm 7 computes a diagram D_{up} that corresponds to the result of operator $\circ_{up}$ according to Definition 5.6. It runs in time $O(\prod_{i=1}^{n} |V_i|)$ where $|V_i|$ is the number of nodes of diagram D_i.

Proof sketch. Correctness. Obviously the algorithm produces a single diagram D_{out}, since the while loop runs as long as at least two classifiers exist. We have to show that for all domain elements d, $D_{out}(d) = (\circ_{up}(D_1, \ldots, D_n))(d)$ in order to prove that the algorithm actually computes the proposed operator.

Let $d \in \mathcal{D}$ be an arbitrary domain element and let c_{exp} be the expected class label returned by Algorithm 6.

By inserting $S.D_{|S|}$ into each leaf node of $S.D_{|S|-1}$ until a single diagram remains, we obviously generate one leaf for each combination of individual results. When the resulting diagram is used to classify d, we end up with a path $P = v_1, \ldots, v_l$ that exactly encodes the set of satisfied conditions, including the classifications from the input diagrams. The algorithm uses these values to set the class label of this leaf node. But as one can see, the computation of this label (the inner for loop) is equivalent to our definition of the operator. Therefore the diagram computed by Algorithm 7 classifies d exactly as Algorithm 6 does. Since this is true for all $d \in \mathcal{D}$, our algorithm for computing D_{up} is correct.

Termination. In each iteration the number of diagrams is decremented by 1 by fusing two of them. Since this can only be done a finite number of times, the while loop eventually terminates. The outer for loop iterates over the finite set of leaf nodes and the inner one over the finite set of preference rules, thus they terminate as well.

Complexity. We show by induction on the number of diagrams that the incorporation is possible in time $O(\prod_{1}^{n} |V_i|)$.

Induction base ($n = 2$): Clearly, for inserting D_2 into each leaf of D_1 we need to make a number of copies that is restricted by $O(|V_1|)$ (since the number of leaves in a binary tree is approximately half of the total number of nodes, as one can show by induction). This is possible

in time $O(|V_1| \cdot |V_2|) = O(\prod_1^n |V_i|)$ and results in a number of leaves that is restricted by the same upper bound.

Induction step: The incorporation of diagrams $D_1, \ldots, D_{n-1}$ is possible in time $O(\prod_1^{n-1} |V_i|)$ by induction hypothesis. This results in a single diagram, which can be fused with D_n in time $O((\prod_1^{n-1} |V_i|) \cdot |V_n|) = O(\prod_1^n |V_i|)$ by induction hypothesis (since $2 < n$). □

Average

Another solution is not to completely trust one of the classifiers, but give some credibility to each of them. That is, we want to compute some kind of average operation. This operator is mainly inspired by [Hall et al., 1998]. First we summarize the results of this paper.

Existing Approach for Parallel Learning

Despite the fact that Hall et al. had another application scenario than belief merging in mind, the procedure is well-suited as merging operator. Instead of learning *one* decision tree from a large data set, multiple trees are learned from random subsets of the training data due to complexity reasons. Then the individual results are combined. First they transform the decision trees into sets of production rules (a step that we are going to skip), where each leaf node gets assigned one rule. Then the sets of rules are merged into one set, where conflicts (i.e., a domain element is classified differently by different rule sets) are basically resolved using two strategies.

First, rules that query more variables are preferred to rules that incorporate less. In terms of trees this means that the path from the root to the leaf is longer in one tree than in the other. The idea behind this is that a more specific case is considered to be more trustful since it applies to special cases, whereas general rules can more easily result in a default classification because of the lack of better evidence.

Their second strategy merges conflicting rules that use the same number of variables. In this case they will contain at least one contradicting condition $X \leq c_1$ vs. $X \leq c_2$ with $c_1 \neq c_2$ (otherwise the rules were equivalent). Then they use the greater of the two values in the final rule set, while they use the smaller one in case of a $>$ comparison.

Semantics of a Belief Merging Operator

We will implement their second strategy but adopt it such that the average is used rather than one of the two comparison values. A similar approach was developed in [Williams, 1990], but with the difference that Williams works with examples from a training set for computing the averaged decision boundary. That is, he computes the mean value of the element nearest to the first classification boundary and the element nearest to the second. We will rather work with the numeric comparison values in the decision tree since we assume that we have no training set.

Of course the idea could even be expanded such that the user can specify all the details about which decision boundary to select in an additional operator parameter, but since it is only thought to be a demonstrating example, a simple version of Hall's rule is sufficient.

As a motivating example, consider Example 5.6. It is intuitively clear how the result should look like. Since one diagram queries $Y > 10$, and the other one $Y > 15$, the merged one contains the condition $Y > 12,5$. This shows our understanding of the term *average*. Not the final classifications are averaged (which is not possible, since we work with discrete class labels), but

the conditions within the diagram. Formally this means that the *conflict set* (as introduced in Definition 4.4) is equally partitioned, such that each of the input diagrams can assert its classification in half of the cases. This is illustrated in Figure 5.8.

Example 5.6. The merged version of the trees from Example 4.3 after application of $\circ_{avg}$ is shown in Figure 5.9.

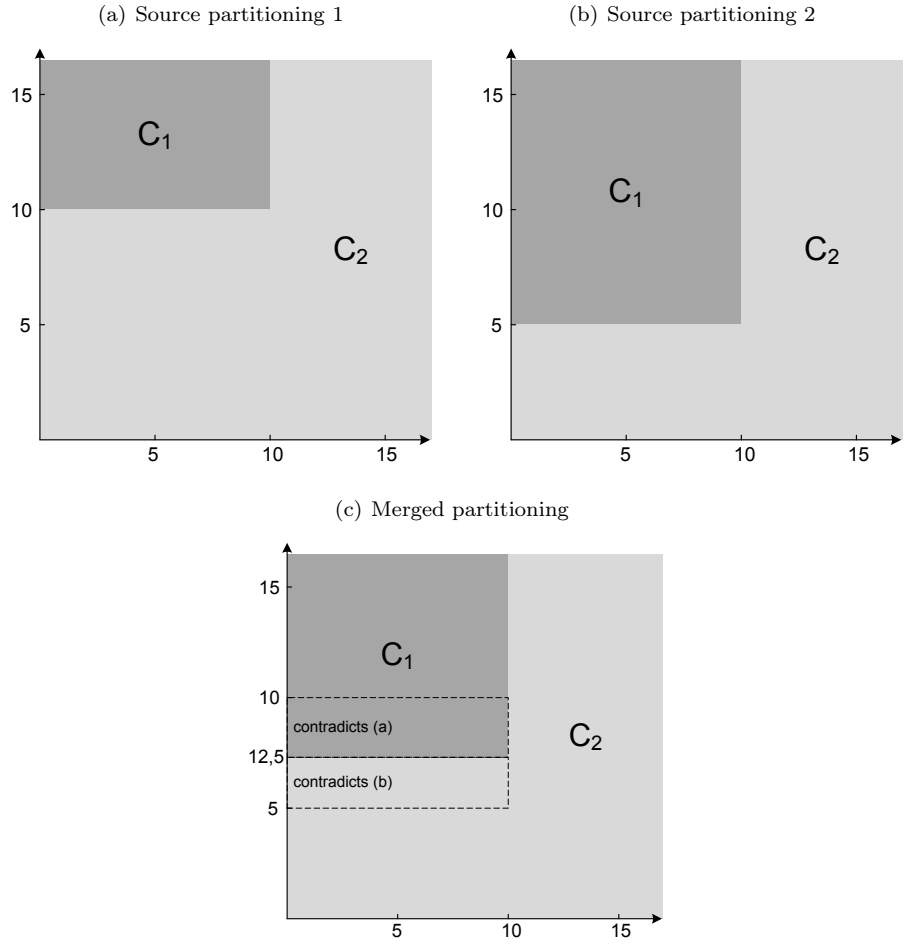

Figure 5.8: Class partition of the trees shown in Figures 4.3(a) and 4.3(b) (of Example 4.3) and the averaged diagram

The conflict set (shaded region) is partitioned such that for half of the cases, source 4.3(a) is considered to be correct and source 4.3(b) as false and vice versa.

It is tricky to formally define the concept of an *average diagram*. Consider for instance two diagrams that reason over completely different sets of variables. In this case, it is not trivial to see what *average* means. Or even more obvious, consider two trees that both consist only of one leaf node but with different labels. How can we compute the *average* in such cases?

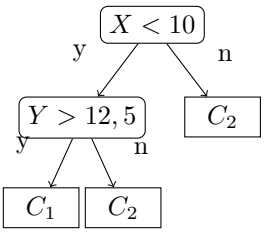

Figure 5.9: Result of $\circ_{avg}$ applied on the diagram from Example 4.3

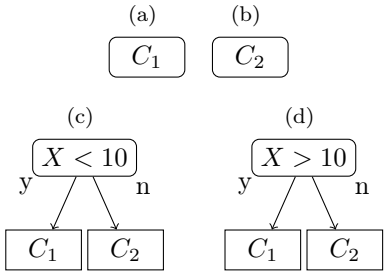

Figure 5.10: Mean computation is impossible

What actually happened in Example 5.6 is that the mean of the *conditions* in the inner nodes was computed. This is of course only possible if both conditions query the same variable. Example 5.7 shows several situations where it is impossible to compute a mean value.

Example 5.7. The two pairs of trees shown in Figure 5.10 (5.10(a) and 5.10(b) as well as 5.10(c) and 5.10(d)) show situations where the computation of a mean value is impossible.

In case 5.10(a)/5.10(b) the sources come to a different classification. In 5.10(c)/5.10(d) the comparison operator is different.

Now consider Example 5.8. At first glance we could mean that this situation is similar to (2) since different variables are queried. However, in this case it is actually possible to take the union of the diagrams. It is even not required to compute something like a mean value since the sources can be perfectly integrated. We simply *insert* 5.11(a) into every leaf node of 5.11(b). That is, we first classify by 5.11(b) and after it has come to a conclusion, we start 5.11(a). Then, if both diagrams agree, the final result is fixed. Otherwise they are contradicting (labeled with question marks).

Example 5.8. In case of the two trees shown in Figure 5.11(a) and 5.11(b), mean computation is possible and results in the tree in Figure 5.11(c).

Now the formal definition of this operator needs to be given. We will assume that the input diagrams are *ordered trees*. That is, the variables are requested in a predefined ordering $X_1 < X_2 < \ldots < X_n$.

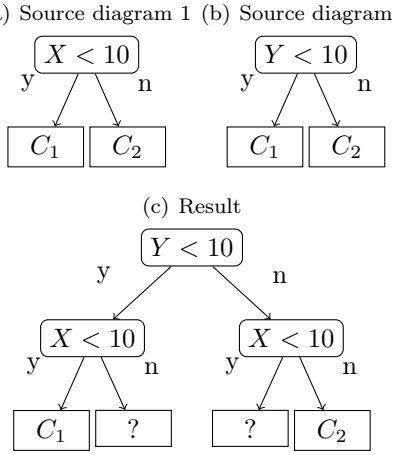

Figure 5.11: Mean computation

Definition 5.7. Let D_1 and D_2 be two decision diagrams with root nodes R_1 and R_2. Let us further denote an inner node N with condition $X < V$ and subnodes S_l and S_r as $N(X < V, S_l, S_r)$. Then the diagram $D_{avg} = \circ_{avg}(D_1, D_2)$ ($\underline{average}$) is computed by Algorithm 9.

Input: source classifiers $D_1, D_2 \in \urcorner_{\mathcal{D},\mathcal{C}}$ with roots R_1 and R_2
Output: diagram D_{avg}
if *R_1 and R_2 are leaves* **then**
  if *R_1 and R_2 both encode the same class* **then**
    `/* just carry over this classification */`
    return R_1
  else
    `/* two leaves contradict each other */`
    return *contradiction*
else
  `/* if both nodes query the same variable, merge the conditions */`
  if *R_1 and R_2 are both inner nodes which query the same variable* **then**
    $R_1 = (X < c_1, s_{1l}, s_{1r})$;
    $R_2 = (Y < c_2, s_{2l}, s_{2r})$;
    `/* merge the subtrees */`
    $s_{ml} = average(s_{1l}, s_{2l})$;
    $s_{mr} = average(s_{1r}, s_{2r})$;
    return *new* $Node(X < \frac{c_1+c_2}{2}, s_{ml}, s_{mr})$
  else
    `/* otherwise: just merge` R_2 `with both subtrees of` R_1 `recursively */`
    Let R_1 be the inner node; if both are inner nodes, let R_1 query the lexically smaller variable (exchange trees if necessary);
    return *new* $Node(X < c_1, merge(s_{1l}, R_2), merge(s_{1r}, R_2))$

Computation of operator $\circ_{avg}$

Informally, this definition states that two leaf nodes are merged directly. If they have the same label, the resulting node in the merged diagram will carry over this label. Otherwise we

discovered a contradiction and the result is undefined.

In case of an inner node, we first check whether both conditions query the same variable. If this is the case we simply average the comparison value and apply the recursive algorithm on the left and right subtree. If the conditions contain different variables we take the condition containing the smaller variable (according to the given total ordering) and pass the other condition (with the greater variable) recursively into both subtrees.

Note that, in contrast to the previous operators, this one is defined by providing an algorithm that actually computes it. Therefore we do not need to prove the correctness since the output of the algorithm is equivalent to the operator's result by definition.

Remember our initial requirement from Section 4.1 stating that if all input classifiers agree, we carry over its result into the merged diagram. Observe that $\circ_{avg}$ does *not* fulfill this requirement, as illustrated by Figure 5.12.

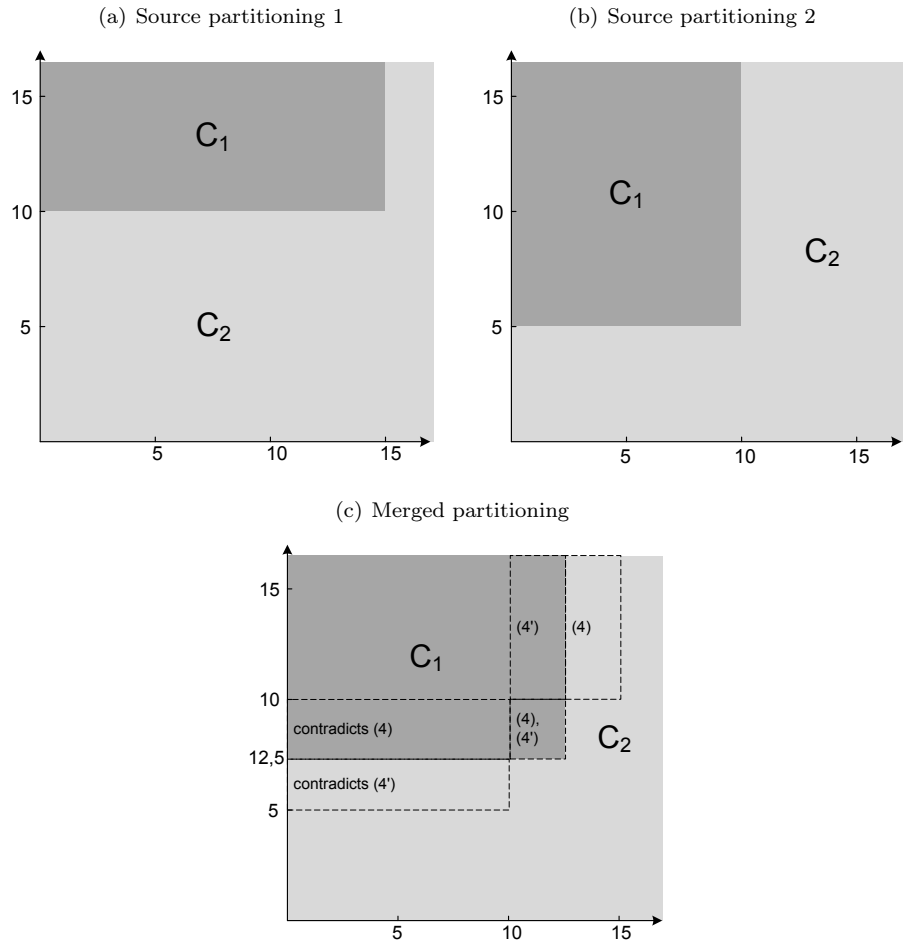

Figure 5.12: The partitions of the merged classifier contain a region that contradicts both inputs

In fact the merging procedure computes the average boundary independently for each dimension. Therefore there will in general emerge regions that contradict *multiple* input classifiers.

Solving this problem is possible but beyond the scope of this chapter. It is up to the developer

of a certain application to select appropriate operators, adjust them and define them in detail. Nevertheless we give some ideas how a solution could look like:

- Add an additional check and apply the average diagram only in cases where the input diagrams differ.
- Use an advanced algorithm for computing the decision regions. This is a task from the field of *machine learning* and beyond the scope of this thesis. For a survey of the topic see [Zhu, 2005] or [Dietterich, 2000].

Majority Voting

A very natural kind of merging is majority voting. That is, for domain element d we let it be classified by each of the input decision procedures. Then we count the number of procedures that returned class c_i for each $c_i \in \mathcal{C}$. We will call this the *number of votes* for a certain class. The merged diagram D_{maj} will return the c_i with the highest number of votes. This procedure was inspired by *boosting* [Breiman, 1996].

This leads to the following formal definition.

Definition 5.8. The n-ary operator $\circ_{maj}$ (*majority voting*) is given by

$$\circ_{maj}(D_1, \ldots, D_n)(d) = \arg(\max_{c \in \mathcal{C}}\{|D_i : D_i(d) = c|\})$$

where arg returns the c that accounts for the according maximum value. In order to get a unique class even in cases where we have multiple maximum values, we define formally:

$$\arg(\max_{c \in \mathcal{C}}\{S\}) = c_i \text{ s.t. } c_i = \max_{c \in \mathcal{C}}\{S\} \wedge \nexists j < i : c_j = \max_{c \in \mathcal{C}}\{S\}$$

That is, if we have several classes with the same number of supporting decision diagrams, c_i has a higher priority than c_j iff $i < j$.

Note that this defines only the properties of the resulting diagram. But in contrast to the previously defined operators, it includes no procedure to actually compute them. This has no special reason except that it was simply easier to define the other operators algorithmically. The implementation of $\circ_{maj}$ is fairly straightforward.

Example 5.9. Consider three decision diagrams in Figures 5.13(a), 5.13(b) and 5.13(c). After application of operator $\circ_{maj}$ we get the result shown in Figure 5.13(d).

Breiman came to the conclusion that this method is an easy way to gain increased accuracy. Enhancements of the methods include weights for the source classifiers. They allow the user to incorporate the trustability of the different classifiers into the final decision [Kolter and Maloof, 2003]. We will use an extended version of this operator in our case studies in Chapter 7.

The implementation of an algorithm for computing the result of $\circ_{maj}$ is very similar to Algorithm 7. Once more, we fuse diagrams to a single one by inserting the latter into each leaf of the former one. The only difference concerns the computation of the class labels of the leaf nodes. Instead of evaluating the sequence of preference rules, we just compare the absolute values of the number of votes for the different classes. Hence, the algorithm is even simpler. The idea of the correctness proof is the same as we had it for Proposition 4.

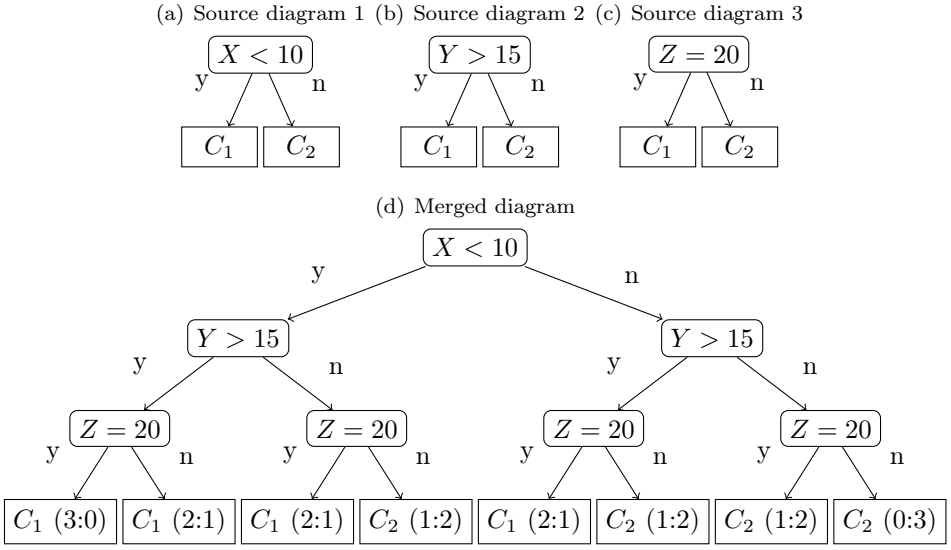

Figure 5.13: Demonstation of operator $\circ_{maj}$

5.3 Simplifying Diagrams

We have developed operators for unfolding diagrams, converting them into binary trees and ordering the nodes with respect to the queried variables. This makes working with the diagrams easier in the sense that the actual merging operators are simpler since they can make more assumptions about their input.

But all of these procedures make the diagrams themselves more complicated. It has been shown that their application will in general lead to an exponential blow-up. This is not only an algorithmic problem, but it makes interpreting and using them also very inconvenient for people. Thus it is desired that we reduce the diagram again in a final step after actual merging was completed.

Note that there exist many different methods how a tree can be simplified. Most of them use a kind of *pruning*. That is, sub-trees are cut off and are replaced by leaf nodes if this introduces only few false classifications while reducing the number of conditions to be checked enormously. Some methods for such strategies are depicted in [Quinlan, 1987]. In contrast, we will stick with *equivalence preserving* simplification techniques that modify only the structure of the diagram without any consequences for the classifications. For this purpose we introduce the operator $\circ_{simp}$.

Basically we apply two strategies for simplification that were proposed in a similar way by [Bryant and Bryant, 1992] for binary decision diagrams (actually they defined three reduction rules since they defined the rule for unifying common subtrees separately for terminal nodes, we will make just one definition that considers inner *and* leaf nodes).

Unifying Common Subtrees

We defined the unfolding operator (Section 5.1) in order to guarantee that the input is a tree rather than a diagram. This procedure duplicated subtrees in order to avoid node sharing. In

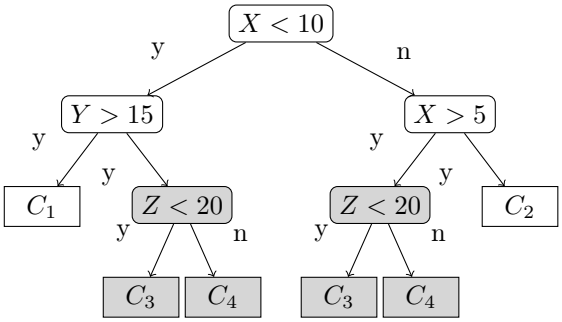

Figure 5.14: Diagram containing two equivalent subtrees

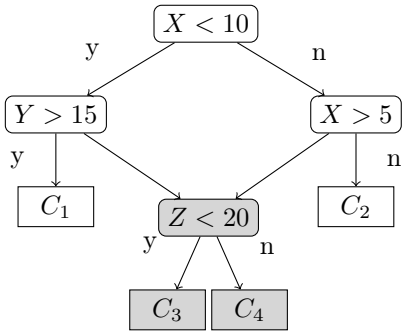

Figure 5.15: Diagram from Figure 5.14 after common subtrees were eliminated

the final result, this kind of redundancy is highly undesired since it makes diagrams look more complicated than they actually are.

Thus we need to unify common subtrees. This step is obviously equivalence preserving and somehow the contrary of unfolding. To demonstrate this, we revisit Example 5.1. All that needs to be changed is that we read it in reverse order.

Example 5.10. We are given the diagram in Figure 5.14. Obviously it contains equivalent subtrees (shaded nodes). One of these subtrees can be eliminated if we redirect its ingoing edge to the other one. The result is shown in Figure 5.15.

Eliminating Branches

After merging tasks were performed or as a result of the previous simplification step we will sometimes end up with a node that has only one distinct successor. Therefore the next state is fixed as soon as this node is entered. Consequently, querying a variable and checking conditions is completely unnecessary. In such cases we can simply cut out the node and redirect its ingoing edge directly to the appropriate child. Let us revisit Example 5.11.

Example 5.11. We are given the diagram in Figure 5.16(a). Since both edges from the node with condition $C \leq 20$ lead to the same successor, the node can be eliminated as shown in Figure 5.16(b).

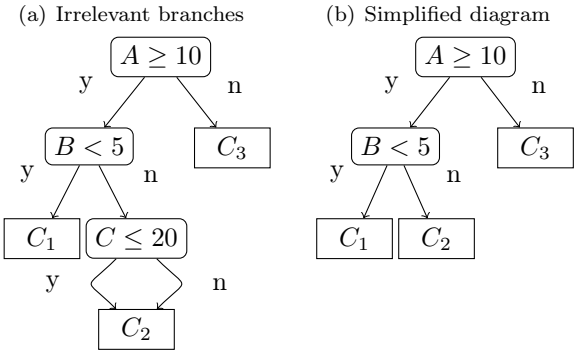

Figure 5.16: Elimination of irrelevant branches

Clearly this strategy can be applied iteratively. Thus, if a subtree uses only one kind of leaf node, the final decision is fixed as soon as the subtree is entered and none of the branches will have an effect. By iterative application of the previous and this simplification rule, this subtree will eventually collapse to a single leaf node.

Operator Definition

We come to the formal definition of this operator.

Definition 5.9. The unary operator $\circ_{simp}$ (*diagram simplification*) is defined as:

$$\circ_{simp}(D)(d) = D(d)\ \forall d \in \mathcal{D} \text{ and } \nexists D', D'' \subset \circ_{simp}(D) \text{ s.t. } D' \neq D'' \text{ and } D'(d) = D''(d)\ \forall d \in \mathcal{D}.$$

Informally, this states that there do not exist any common subdiagrams that represent the same classifier.

We will again suggest an algorithm that implements this operators, namely Algorithm 10.

Proposition 5. $\circ_{simp}$ is computed by Algorithm 10 in time $O(|V|^4 + |V|^3 \cdot |E|)$.

Proof sketch. Correctness. We first prove that the output diagram is semantically equivalent to the input. If we can show that this is an invariant of the while loop, it clearly follows that the statement is also true for the whole algorithm.

Let I be an arbitrary input diagram and let I' be the diagram after execution of the first if block. If the condition is not satisfied, $I = I'$. Otherwise there exists a node v in I which was removed from I'. But then, for each edge (w, v) that leads to node v in I, there will be an edge (w, v') in I' s.t. v and v' are the roots of semantically equivalent diagrams (otherwise the if condition would not be satisfied). Therefore the final result will be the same.

Let I'' be the result after execution of the second if block. Again, if the condition is not satisfied, $I' = I''$. Otherwise there will be a node v in I' which was removed from I''. But then $w_i = w_j$ for all conditional edges (v, w_i, c_i), (v, w_j, c_j) in I', otherwise the if condition was not satisfied. We denote the unique successor of v as w. But then we will end up in node w independently from the satisfaction of the conditions. Since v was removed from I'', and all input edges were redirected to w, the final result will obviously be the same in all cases.

Input: Input: $D \in \urcorner_{\mathcal{D},\mathcal{C}}$ with root node $R \in V$
Output: Output: $D' \in \urcorner_{\mathcal{D},\mathcal{C}}$ with $D(d) = D'(d)\ \forall d \in \mathcal{D}$ and D' does not contain common subtrees
$D_{new} = D$;
while *changes* **do**
  `/* unify common subtrees */`
  if $\exists D', D'' \subset D_{new}$ *s.t.* $D'(d) = D''(d)\ \forall d \in \mathcal{D}$ **then**
    let r' and r'' be the roots of D' resp. D'';
    replace all edges (v, r'') by (v, r');
    delete $D'' \setminus D'$;
  `/* remove nodes with only one distinct successor s */`
  if $\exists n, s \in D_{new}.V$ *s.t.* $(\forall v)(n, v) \in D_{new}.E \Rightarrow v = s$ **then**
    replace all edges (v, n) by (v, s);
    remove node n;
return D_{neu}

Algorithm 10: simplify

No more simplifications are possible. Suppose there would be another simplification according to our proposed strategies which was not performed by the algorithm. This is impossible, because in this case one of the if conditions was satisfied and the algorithm would not have terminated.
Termination. We have shown that none of the if blocks changes the semantics of the diagram, therefore this is an invariant of the loop. Observe that both if blocks reduce the number of nodes in the diagram. This can clearly be done only a finite number of times, which guarantees that the algorithm will terminate.

Complexity. The first simplification strategy runs in time $O(|V|^3 + |V|^2 \cdot |E|)$. For the detection of equivalent sub-diagrams we first need to iterate through all pairs of nodes v/w which are the roots of possibly equivalent sub-diagrams. This runs in time $O(|V|^2)$. Checking for equality is possible in time $O(|V| + |E|)$, leading to a total complexity of $O(|V|^3 + |V|^2 \cdot |E|)$.

The second simplification strategy is much simpler. Obviously, detecting nodes with unique successors, as well as removing them, is possible in $O(|V| + |E|)$.

Therefore the complexity of one iteration is dominated by $O(|V|^3 + |V|^2 \cdot |E|)$. Since the whole algorithms runs as long as changes were made, and each modification eliminates at least one node, the overall complexity of the algorithm is $O(|V| \cdot (|V|^3 + |V|^2 \cdot |E|)) = O(|V|^4 + |V|^3 \cdot |E|)$. □

This operator will reduce much of the blow-up that was caused as a preparation step. This was empirically discovered during the experiments that are documented in Section 7. However, there is no claim for optimality. This operator is just a demonstration of the possibilities that arise when using the developed framework and a starting point for experimenting and including further algorithms. Decision diagram optimization is a different and complex field of research and practical approaches are mostly heuristics since the problem is NP-complete [Bennett, 1994]. For an approach with genetic algorithms see [Lenders and Baier, 2005].

5.4 Solving different Task Variants

Finally we can continue our discussion of different task variants from Section 4.4.

Input graph type	Output graph type	Node ordering	Conflict resolution	Operators needed
decision tree	decision tree	arbitrary	user preferences	user preference merging operator
decision tree	decision tree	ordered	majority voting	majority voting operator
decision diagram	decision diagram	ordered	average	unfolding, average merging operator
decision diagram	decision diagram	arbitrary	majority voting	unfolding, ordering, majority merging operator

Table 5.1: Task variants

Table 5.1 shows some of the most important combinations of tasks from the last chapter, paired with certain conflict resolution strategies from above. The last column lists the operators that are needed for certain variants.

Observe that despite the fact that we never defined an average operator for decision diagrams, we can solve this task by operator composition. Thus it is sufficient to provide a merging operator for a special case and offer additional unary operators for conversion of diagrams into trees. In other words, the merging step is the core of the problem and can be preceded or followed by an ordering or a graph to tree unfolding algorithm. This shows that a relatively small set of operators can be used for a large variety of different application scenarios. Before practical examples are shown in Chapter 7, we now discuss the implementation of the framework for dlvhex.

Theory is when you know
something, but it doesn't work.
Practice is when something
works, but you don't know why.
Programmers combine theory
and practice: Nothing works and
they don't know why.

Anoymous

Chapter 6

Using dlvhex for Decision Diagram Merging

In the previous two chapters we have formally defined the task of merging decision diagrams. We further have seen that a set of few operators is sufficient to solve all of the suggested task variants by composition.

This chapter puts focus on the actual implementation. We will see how the merging software can be realized using dlvhex and the mergingplugin developed in [Redl, 2010]. The modules explained in the following sections will be summarized in another plugin for dlvhex, named the decisiondiagramplugin. Basically it consists of the operators and a tool for converting decision diagrams between several file formats, especially between human and a machine-readable ones.

6.1 Representation Formats for Decision Diagrams

When decision diagrams are implemented, the first task is to select a suitable representation formalism. It is intuitively clear that some hierarchical data structure will be necessary.

Have a look at the diagram in Figure 6.1. Listing 6.1 shows a possible implementation, though the formal syntax would need to be defined before it can actually be used.

```
{
        if (A < 10){
                if (B < 10){
                        <ClassA>
                }
                else {
                        <ClassB>
                }
        }
        if (A > 20) {
                if (B < 16){
                        <ClassA>
                }
                else {
                        <ClassB>
                }
```

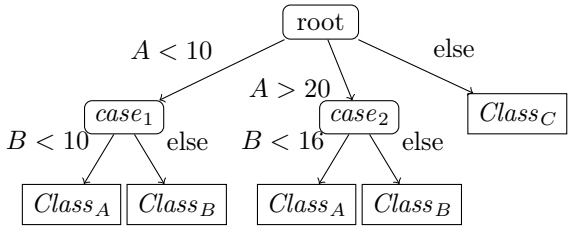

Figure 6.1: Example decision diagram

```
        }
        else{
                <ClassC>
        }
}
```

Listing 6.1: Example implementation of decision diagrams in Figure 6.1

Observe that this format is well-suited for humans. It is straightforward to interpret the definition even though we have not formally defined the syntax. But this definition cannot directly be parsed by dlvhex. A reasoner expects a logic program consisting of rules and facts. It should be clear that this is, on the other hand, difficult to read for humans.

Thus, whatever input format we will finally use, we have to provide a formal counterpart and a conversion mechanism between the two formats.

Representing Graphs Formally

Recall the formal definition of decision diagrams from Section 3.1. We need to represent nodes, directed edges (with or without conditions) and classifications for the leaf nodes. Conditions consist of two values and a comparison operator (we have argued that this is sufficient for all scale types presented in Section 3.2). We further need to store a reference to the (unique) root node of the diagram. This leads to a straight forward implementation. What we need is summarized by the following list.

- a unary predicate for storing the entry point

  $$rootnode(Node)$$

- a unary predicate to define inner nodes

  $$innernode(Node)$$

- a binary predicate for defining leaf nodes with assigned class labels

  $$leaf(Node, Class)$$

- a 5-ary predicate to define conditional edges

  $$conditionaledge(Parent, Child, Operand1, ComparisonOperator, Operand2)$$

  where *Operand1* and *Operand2* are alphanumeric strings, and *ComparisonOperator* is one of “ < ”, “ <= ”, “ = ”, “ > ”, “ >= ”

- a binary predicate for else edges

$$elsechild(Parent, Child)$$

Using these predicates, our motivating example can easily be converted into the formal definition in Listing 6.2.

```
innernode(root).
rootnode(root).

% Level 1
conditionaledge(root, case1, "A", "<", "10").
conditionaledge(root, case2, "A", ">", "20").
elseedge(root, elsecase).

% Level 2
conditionaledge(case1, case1a, "B", "<", "10").
elseedge(case1, case1b).

conditionaledge(case2, case2a, "B", "<", "16").
elseedge(case2, case2b).

elseedge(root, case3).

% Leaf nodes
leaf(case1a, "ClassA").
leaf(case1b, "ClassB").
leaf(case2a, "ClassA").
leaf(case2b, "ClassB").
leaf(case3, "ClassC").
```

Listing 6.2: Formalized decision diagram

This format is suitable for being processed by dlvhex. But it is obviously difficult to read for humans. This is in contrast to the dot file format introduced in the following section.

The dot File Format

In principle a file format like the one presented in Listing 6.1 or XML could be used as input format for the specification of decision diagrams, even though some details remain that need to be worked out first. For instance, the hierarchical structure from above is easy to use for decision trees. But if one works with general acyclic graphs, where two nodes may share a common subgraph, it becomes tricky to incorporate this in the syntax. One would need to provide an additional means for explicit edge definitions.

Another format that is appropriate for this task is the well-known dot format, see for instance [Koutsoos et al., 1993]. This is an open graph format with the major advantage that we have a direct possibility to visualize the resulting graphs using the *dot tools*[1]. Listing 6.3 shows the above example in the very intuitive dot format.

```
digraph G {
        root -> case1 ["A<10"];
        root -> case2 ["A>20"];
        root -> elsecase ["else"];
        root -> case3 ["else"];
        case1 -> case1a ["B<10"];
        case1 -> case1b ["else"];
```

[1] http://www.graphviz.org/

```
        case2 -> case2a ["B<16"];
        case2 -> case2b ["else"];
        case1a ["ClassA"];
        case1b ["ClassB"];
        case2a ["ClassA"];
        case2b ["ClassB"];
        case3 ["ClassC"];
}
```

Listing 6.3: Decision diagram 6.2 in dot format

Note that nodes are mostly defined implicitly by simply using them as endpoints for edges. This is common in dot graphs. Nevertheless one can also explicitly define them if this is necessary because one wants to set additional attributes like node labels (for instance for attaching class labels to leaf nodes).

Die formal syntax of the dot format is given in Appendix A. In the following we develop a converter from dot to our internal formal representation and vice versa.

Converting from and to dot

The graphconverter is a tool that was developed as part of the decisiondiagramplugin. It is a simple command-line application that is installed together with the plugin library.

If we assume that Program 6.2 is stored in the file "graph.hex", we can easily convert it into the dot format by calling:

```
graphconverter hex dot <graph.hex
```

The first parameter states the source format, the second one the desired destination format. In the other direction the call looks like this:

```
graphconverter dot hex <graph.dot
```

For details about the graphconverter we refer to the included online-help that can be read by entering:

```
graphconverter -help
```

Appendix B contains the user guide of this tool.

6.2 Architectural Overview

We have shown how the mergingplugin can be extended with custom merging operators (see Section 2.3). This feature will now be exploited for implementing the operators. The functionality was embedded in the decisiondiagramplugin that already contains the graphconverter from Section 6.1. This allows us to pack all the features that are necessary for the work with decision diagrams into a single package which is very user-friendly.

Interpreting the Input Diagram

At the beginning of this chapter, we have discussed that decision diagrams need to be converted from a human-readable data format into a pure HEX program consisting of facts over certain predicates. This allows dlvhex to read the input. However, after the input was parsed and converted into an instance of the class AtomSet[2] internally, it is very inconvenient to work with

[2] http://www.kr.tuwien.ac.at/research/systems/dlvhex/doc/

this representation. A set of facts is a *flat* data format, and even though it can be used to store the hierarchical structure of diagrams, this is far from optimal. The solution was only selected due to the fact that it is simply the only format that dlvhex can interpret.

A much better solution is to convert the AtomSet into an internal graph structure after dlvhex has parsed the input. In C++, objects and pointers can be elegantly used for representing tree-like structures. A good data structure simplifies the implementation of merging operators enormously. But it is clear that a back-conversion must be done after the operators were applied since dlvhex eventually must return an answer set rather than arbitrary C++ structures.

To summarize, we will work with three different data formats. The human-readable source format is dot. Using the graphconverter we translate it into sets of facts for the only reason that dlvhex expects its input to be a logic program. But due to the fact that a set of literals is inconvenient if one wants to write operators that work with diagrams, another conversion is done as soon as the decisiondiagramplugin comes into play. Before the actual operator implementation is called, the plugin will create C++ data structures for storing the diagram and eventually pass the diagrams in form of a pointer to this kind of representation to the operator.

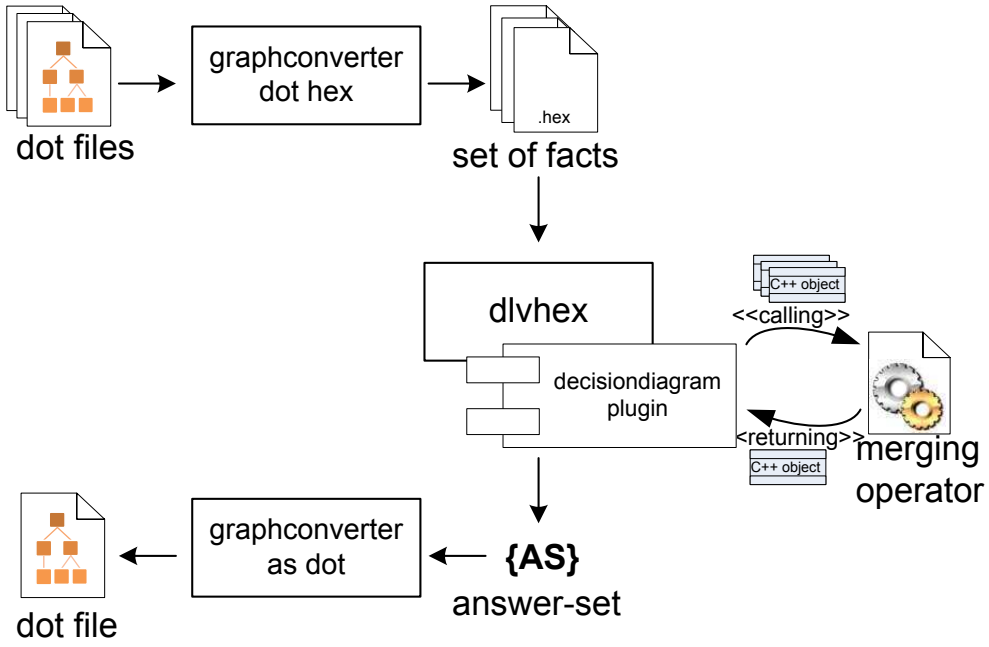

Figure 6.2: Used data formats and conversions between them

Then the result of the operator must be translated into an answer set representation before it can be output by dlvhex. After dlvhex has terminated, its console output can finally be directed through graphconverter once more to generate a human-readable diagram in form of a dot file. Figure 6.2 illustrates the process.

Internal Representation of Diagrams

Figure 6.3 shows the internal representation of decision diagrams. This is the format that is finally used to pass the input diagrams to the merging operators.

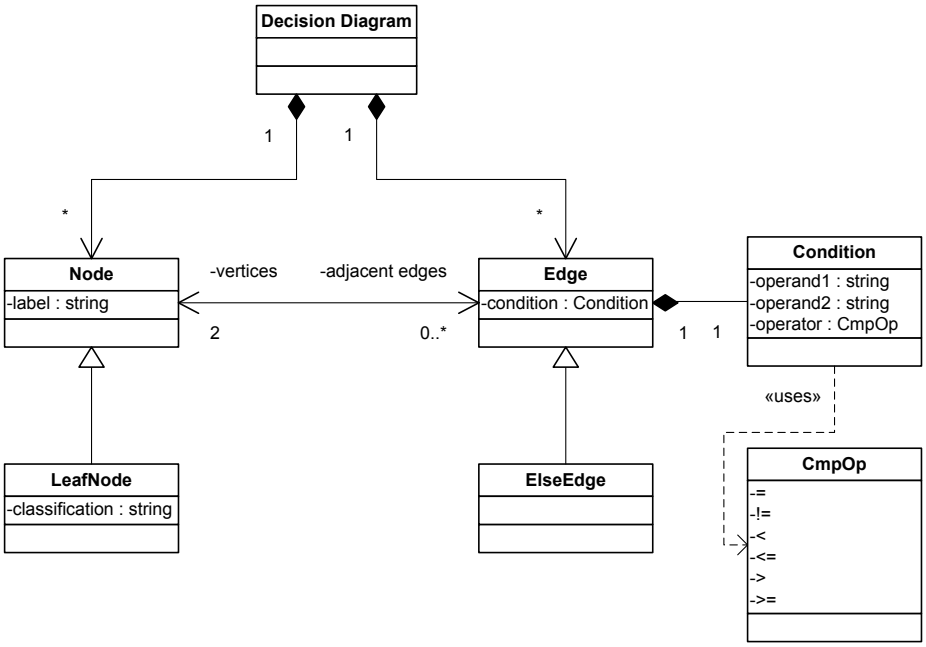

Figure 6.3: Internal decision diagram representation

In fact, the classes DecisionDiagram, Node and Edge additionally implement many well-known graph algorithms (like cycle detection) that simplify the work with the framework for users who write their own merging operators. But since these are technical details which do not change anything in theory, they were not shown in the diagram to keep things simple.

6.3 Operator Implementation

Recall the list of operators given in Chapter 5. We have already given algorithmic definitions for most of them, thus the actual implementation is straightforward and it is not necessary to discuss it separately for each operator. Nevertheless the principles will be explained.

Arity	Operation
1	ordering of decision diagrams
1	unfolding of decision diagrams into trees
1	converting arbitrary into binary decision diagrams
1	simplifying of binary decision diagrams by redundancy reduction
2	merging of ordered binary diagrams using range average
2	merging of ordered binary diagrams using majority voting
2	merging of ordered binary diagrams using user preferences

Table 6.1: Types of necessary operators

First look at the code snippet in Listing 6.4. It shows the implementation of a typical operator in the decisiondiagramplugin. The apply method first checks the arity. The operator *simplify* is

unary and expects a set that contains *exactly* one answer set.

```
string OpSimplify::getName(){
        return "simplify";
}

HexAnswer OpSimplify::apply(int arity, vector<HexAnswer*>& answers,
                            OperatorArguments& parameters) throw
                                (OperatorException){

        try{
                // Check arity
                if (arity != 1 || (*answers[0]).size() != 1){
                        // error handling
                }

                HexAnswer output;

                // Convert the AtomSet into an instance
                // of DecisionDiagram
                DecisionDiagram dd((*answers[0])[0]);

                // computation on dd
                ...

                // back-conversion into an answer set
                output.push_back(simplify(dd).toAnswerSet());

                return output;
        }catch(InvalidDecisionDiagram ide){
                        // error handling
        }
}
```

Listing 6.4: Implementation of a typical merging operator

Before the actual operator semantics is implemented, the atom set is converted into the internal diagram representation as discussed in Section 6.2. This is done by the constructor of class DecisionDiagram. After the computation, the diagram is back-converted into an atom set since this is the only format that can be returned by dlvhex.

This implements a merging operator as C++ class. But before it can finally be used we need to implement the operator import function. This method is expected to return a vector of all operators which are provided by the library (see Section 2.3). It is shown in Listing 6.5.

```
OpUnfold unfold;
...
OpSimplify simplify;

extern "C"
vector<IOperator*>
OPERATORIMPORTFUNCTION()
{
        vector<IOperator*> operators;
        operators.push_back(&unfold);
        ...
        operators.push_back(&simplify);
        return operators;
}
```

Listing 6.5: Operator import function of the decisiondiagramplugin

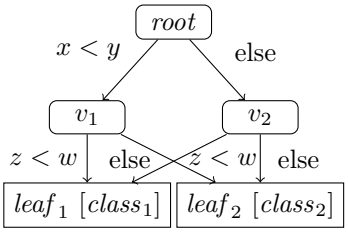

Figure 6.4: A decision diagram with node sharing

Note that the developed plugin does in fact *not* provide any external atoms for dlvhex but only merging operators. Nevertheless it is a normal dlvhex plugin that is installed in the system plugin directory. The mergingplugin will check for all libraries in this directory if an operator import function is found, and if this is the case, it is called. For more details see [Redl, 2010].

6.4 Demonstration

To conclude this chapter the implemented framework will be demonstrated with a simple example. More advanced and practical application scenarios will be discussed in the next chapter.

Assume we have the decision diagram shown in Figure 6.4. It needs to be unfolded, i.e., we need a decision *tree*:

The diagram is first encoded in human-readable dot file "input.dot", which looks as follows:

```
digraph {
    root [label="root"];
    v1 [label="v1"];
    v2 [label="v2"];
    leaf1 [label="leaf1 [class1]"];
    leaf2 [label="leaf2 [class2]"];
    v1 -> leaf1 [label="z<w"];
    v1 -> leaf2 [label="else"];
    v2 -> leaf1 [label="z<w"];
    v2 -> leaf2 [label="else"];
}
```

The dot file can either be written by hand, which is not very complicated, or by the use of a graphical user interface like the one provided by the X11 tool dot. Then we can use the command

```
graphconverter dot hex < input.dot > input.hex
```

to get the following HEX program:

```
root(root). innernode(root). innernode(v1). innernode(v2).
leafnode(leaf1, class1). leafnode(leaf2, class2).
conditionaledge(root, v1, x, "<", y). elseedge(root, v2).
conditionaledge(v1, leaf1, z, "<", w). elseedge(v1, leaf2).
conditionaledge(v2, leaf1, z, "<", w). elseedge(v2, leaf2).
```

Of course one could also write this program by hand, but this is not suggested since the flat file structure can quickly become confusing. Finally we need to write our merging plan "graph_to_tree.mp" before the merging procedure can be started:

```
[common signature]
predicate: root/1;
predicate: innernode/1;
predicate: leafnode/2;
```

```
predicate: conditionaledge/5;
predicate: elseedge/2;

[belief base]
name: kb1;
source: "input.hex";

[merging plan]
{
        operator: unfold;
        {kb1};
}
```

Basically, we load the input graph from file "input.hex" and call this belief base *kb1*. Then we just apply the *unfold* operator. The common signature is fixed for all decision diagrams. This merging plan file is passed through the merging plan compiler. Its output is a long HEX program that uses several external atoms and is therefore rather confusing. Thus we omit it and refer to [Redl, 2010]. The only thing of interest is that this HEX program will finally compute the desired result when we pass it to dlvhex:

```
mpcompiler graph_to_tree.mp |
   dlvhex --silent --filter=root,innernode,leafnode,conditionaledge,elseedge --
   > result.as
```

The filter is again a technical detail that is discussed in the cited thesis and can be ignored for now. But note that dlvhex is called in *silent mode* to restrict the output to the actual answer set (otherwise it would print some "about" information). This is necessary to make the output compatible with the expected input format of the graphconverter.

The result of dlvhex will be the final decision diagram, but in the rather confusing format of an answer set:

```
{root(root), innernode(root), innernode(v1), innernode(v2),
leafnode(leaf1, class1), leafnode(leaf2, class2),
leafnode(leaf1_1, class1), leafnode(leaf2_1, class2),
conditionaledge(root, v1, x, "<", y), elseedge(root, v2),
conditionaledge(v1, leaf1_1, z, "<", w), elseedge(v1, leaf2_1),
conditionaledge(v2, leaf1, z, "<", w), elseedge(v2, leaf2)}
```

Therefore we add a final call of the graphconverter to get a readable output diagram:

```
graphconverter as dot < result.as > result.dot
```

The output is in the intuitively readable dot format:

```
digraph {
     v2 -> leaf2 [label="else"];
     v1 -> leaf2_1 [label="else"];
     root -> v2 [label="else"];
     v2 -> leaf1 [label="z<w"];
     v1 -> leaf1_1 [label="z<w"];
     root -> v1 [label="x<y"];
     leaf2_1 [label="leaf2 [class2]"];
     leaf1_1 [label="leaf1 [class1]"];
     leaf2 [label="leaf2 [class2]"];
     leaf1 [label="leaf1 [class1]"];
     v2 [label="v2"];
     v1 [label="v1"];
     root [label="root"];
}
```

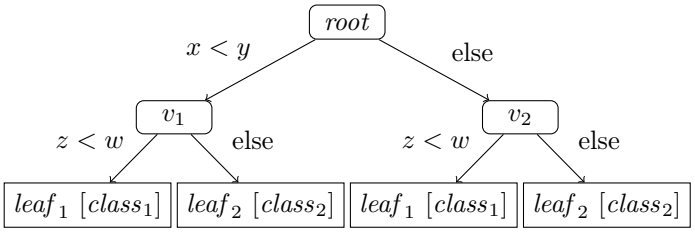

Figure 6.5: Decision diagram from Figure 6.4 unfolded

This can be visualized with the dot tools (dotty, dot, etc.) and looks like shown by Figure 6.5, which is just like expected.

Of course, this example shows only unfolding rather than merging. This was done to keep things simple. Merging works exactly in the same way in principle. The only difference regards the merging plan, where we need to define multiple belief bases and apply an n-ary operator rather than a unary one.

```
[belief base]
name: kb1;
source: "input1.hex";

[belief base]
name: kb2;
source: "input2.hex";

[merging plan]
{
        operator: majorityvoting;
        {kb1};
        {kb2};
}
```

This concludes our demonstration of the capabilities. In the following chapter we consider more sophisticated applications. But prior to this we summarize the main benefits of the proposed plugin.

6.5 Framework Benefits

Clearly, decision diagrams can also be merged by hand, therefore there must be advantages that justify the personal training overhead that arises when the plugin is introduced. As we have seen in Chapter 5, there exists a large variety of merging strategies and even more could be implemented if this is reasonable for a certain application.

It is often the case that we do not know right from the beginning which of the operators will behave best. Additionally we may want to experiment with different decision diagrams in the leaves of the merging plan. To perform the merging task manually each time when we wish to modify our setting this is a tedious waste of time. Therefore the actual benefit of the framework is the reduction of such routine tasks. It offers a set of merging operators that were proposed in the literature and leaves it up to the user to make case studies and finally select the merging strategy that delivers the desired result.

The plugin developed in this thesis is considered as an extension to [Redl, 2010]. Whereas the cited thesis shows the implementation of a *general* belief merging framework, where the sources can be of arbitrary type, this document assumes the inputs to be decision diagrams. We

have shown that the general belief merging framework can also be used for decision diagram merging in particular. This requires not only the implementation of appropriate operators, that are beyond general merging operators, but also pre- and post-processing steps in order to encode diagrams such that they can be processed by dlvhex.

If your result needs a statistician
then you should design a better
experiment.

Ernest Rutherford

Chapter 7

Case Studies

In this chapter we are going to use the developed framework to solve practice-orientated tasks. First we will consider biomedical problems from the fields of molecular biology, health care and administration. At the end of the chapter we go a step beyond and show that the framework can also fruitfully be used for non-medical applications.

Before we start with concrete examples, the goal of this chapter shall be pointed out. The scenarios are intended to demonstrate the flexibility of the framework. Clearly, merging of decision diagrams can also be done by hand. However, in many applications it is not known right from the beginning which of the merging strategies will lead to the best result. Therefore it would be necessary to try out several settings with different parameters. If one has to merge the diagrams manually in each run, this is a great waste of time. The framework takes this burden from the user.

All that needs to be done is to define the operators once. Then one can simulate several combinations and sequences of operator applications by automatically generating the merged decision diagram. Or in other words, the user does not have to perform routine tasks by hand and can focus on the development and optimization of the actual merging operations.

7.1 DNA Classification

A central task in automatic or semi-automatic generation of protein databases is the recognition of genes in DNA sequences. DNA[1] is the carrier of genetic information in every known living organism. Basically it is a huge molecule that consists of the four bases Adenine, Guanine, Cytosine and Thymine, that are arranged in a certain order, which is unique for each organism (except monozygotic twins).

The genetic flow of information is summarized as follows. The overall genetic information is stored in each cell of an organism. In eukaryotes, the nucleus encapsulates it, in procaryotes the DNA is stored as ring-shaped chromosome directly within the cytoplasm. Then, certain regions of the DNA, called *genes*, are *transcribed* into mRNA[2], which is in turn sent to cell organelles called *Ribosomes*. There they are *translated* into proteins consisting of amino acids, where 3

[1] Deoxyribonucleic acid
[2] Messenger RNA

bases in sequence (called *codon*) are mapped onto one amino acid. Finally the resulting proteins initiate metabolic processes in the cell.

However, only a minor part of the total DNA will ever be transcribed since most of it is so called *junk DNA*. That is, it does not encode any proteins but is rather useless. This phenomenon has different reasons, for instance that genes can become useless during evolution. In case of humans, about 97% of the total genetic information are junk DNA and only 3% are protein coding. Clearly, if one wants to construct a protein database like SWISSPROT, it is necessary to classify automatically sequenced DNA into *coding* and *non-coding* samples.

Task Description

This directly leads to the formal task description. We are given a sequence s over the alphabet A, G, C and T (for the four bases Adenine, Guanine, Cytosine and Thymine):

$$s \in \mathcal{D} = \{A, G, C, T\}^+$$

Our set of possible classifications is just coding and non-coding:

$$\mathcal{C} = \{coding, non\text{-}coding\}$$

As usual, a classifier c is a mapping from $\mathcal{D}$ to $\mathcal{C}$:

$$c : \mathcal{D} \to \mathcal{C}$$

DNA Features

In the literature many implementations based classifiers have been proposed. The underlying formalisms include neural networks, support-vector machines and decision trees.

But before we can use machine learning tools to train such classifiers, we need to answer one central question, namely what we consider as input. A straight forward approach would be to take the DNA sequence itself, but this does not work very well. The reason for this is that the sequence of bases varies too much from species to species, from individual to individual and from gene to gene. Thus, the difference between sequences reflects more the differences between genes than between the two classes coding and non-coding.

What we need is some kind of measurement that allows us to make a distinction between these classes. This leads to so called *DNA features*. Features are nothing more than numeric values that can be computed for a given sequence. They incorporate knowledge from molecular biology that allows us to distinct between the two classes. For instance, it is known that the predominant bases at the first codon position are purines (A and G are purines, C and T are pyrimidines), whereas in non-coding sequences, the distribution is rather random [Peng et al., 2005].

Example 7.1. Let

$$s = ATTGACAGGCTCCATGCA$$

Then we compute the feature

$$f_6 = \max_{i=1}^{3}(a_i + g_i)$$

where a_i (g_i) is the relative frequency of Adenine (Guanine) on the first position in reading frame i. Note that given a sequence, we *cannot* assume that the first codon starts with the first character. Therefore the codon boundaries are not fixed and we have 3 possible *reading frames*, namely:

$$s_1 = ATT - GAC - AGG - CTC - CAT - GCA$$

$$s_2 = A - TTG - ACA - GGC - TCC - ATG - CA$$
$$s_3 = AT - TGA - CAG - GCT - CCA - TGC - A$$

Therefore $a_1 = \frac{2}{6}$ since 2 of the 6 codons start with A and $g_1 = \frac{2}{6}$ since 2 start with G. Similarily $a_2 = \frac{2}{5}$, $g_2 = \frac{1}{5}$, $a_3 = \frac{0}{5}$, $\frac{1}{5}$. Then we can compute

$$f_6 = \max_{i=1}^{3}(a_i + g_i) = a_1 + g_1 = \frac{4}{6}$$

This is the numeric value of feature f_6.

As the name f_6 suggests, there exist numerous different features. Another one in use is the frequency of triplet ATG, which is more common in coding than non-coding sequences.

One of the most common feature sets was proposed in [Fickett and Tung, 1992] and refined in [Peng et al., 2005]. We will also use these features in our experiments. The DNA data in use was extracted by Fickett and Tung in 1992 from the Human Genome Project. It can be downloaded from `http://www.fruitfly.org/sequence/human-datasets.html`[3].

Existing approaches

Now that we have extracted DNA features, they can be used to train a classifier using machine learning techniques. We will follow [Salzberg et al., 1998] who implemented the MORGAN system using decision trees.

Basically, MORGAN is based on OC1. While OC1 uses a single decision tree, MORGAN trains multiple trees with randomization and combines the single trees as follows. The leaf nodes do not only store the classification, but also the frequency distribution in the training set. For instance, if 100 samples of the training set ended in a certain leaf node, where 70 where coding (abbr. c) and 30 were non-coding (abbr. n), the classification is c with the frequency distribution $\{c : 70, n : 30\}$.

When a new example needs to be classified, it is first put into each of the trees. Then the final classification is computed by adding the frequency distributions and taking the class with the highest number of training examples.

Example 7.2. Assume we have two decision trees T_1 and T_2 and a sequence s. The frequency distributions delivered by the trees are:

$$T_1(s) = \{c : 70, n : 30\}$$
$$T_2(s) = \{c : 40, n : 60\}$$

Then the combined distribution is:

$$T_1(s) + T_2(s) = \{c : 110, n : 90\}$$

Therefore, the final classification is c (coding). In other words, T_1 can insist on its classification. This is a consequence of the fact that T_1 has better evidences for c $(70 : 30)$ than T_2 has for n $(40 : 60)$.

Note that in the above example, both decision trees contained the same number of training samples in the leaf nodes in question. If this is not the case, Salzberg suggests weighting of the distributions according to the total number of samples in the node.

[3] visited on 2010-03-13

Implementation

We are going to use the DNA data from [Fickett and Tung, 1992], the features f_1 to f_{20} from [Peng et al., 2005] and the merging strategy from [Salzberg et al., 1998]. The first task when implementing such a system is the computation of the features for the training and test data.

This is a straight forward task. Basically one has to write a collection of string functions that count bases or triplets and perform some primitive operations. This leads to a set of annotated sequences consisting of the base sequence, a vector of 20 numeric features and a class label (coding or non-coding).

Next we need to construct decision trees for this set of sequences. This is best done by using some existing machine learning software. During the experiments performed in context of this thesis, RapidMiner[4] was used. This is an open-source data mining tool that implements a large repertoire of classification algorithms, among them decision trees.

In total, three different decision trees were trained. The variations concerned both the selected algorithm and the training samples. Figure 7.1 shows the trees. The first one was trained using the criterion "gain_ratio", the second one with "information_gain" and the third one with "gini_index". The training set of only 10 sequences was drawn randomly from a set of 4000 sequences (2000 coding and 2000 non-coding). This very small training set basically leads to trees that only look at the one or two most significant attributes, i.e., the attributes with the greatest variance between coding and non-coding sequences. The selected attributes are different depending on the training set and the selected learning technique.

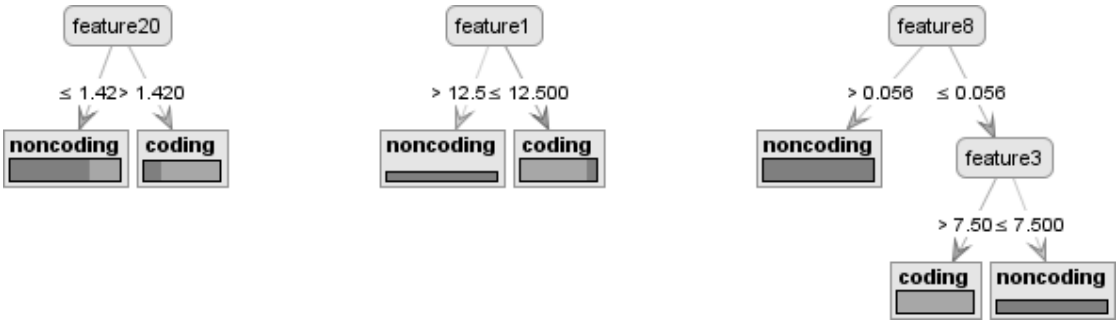

Figure 7.1: Individual source classifiers

The performance of these trees was tested with 2000 test instances (1000 coding and 1000 non-coding) different from the training set. As expected, the results were very poor due to the very small training set. Tables 7.1, 7.2 and 7.3 show the overall performance which is around 50%, or in other words, as good as random classification. An interesting observation is that the first classifiers tends towards *non-coding*, the second one tends towards *coding*, and the third one seems to be slightly better balanced, i.e., the ratio of false positives and false negatives is smaller.

To merge these trees, we need to export them from RapidMiner such that they can be loaded by dlvhex and processed by the plugin developed in Chapter 6. We have already shown how we can work with the human readable dot format. A tool for this purpose called graphconverter was developed in 6.1.

Unfortunately, RapidMiner does not support exporting decision trees in dot format. Instead it uses a proprietary XML format. However, the format is easy to understand. Therefore it was

[4] `http://www.rapidminer.com`

accuracy: 48,05%	true coding	true non-coding
predicted coding	175	214
predicted non-coding	825	786

Table 7.1: DNA classification tree 1

accuracy: 48,85%	true coding	true non-coding
predicted coding	854	877
predicted non-coding	146	123

Table 7.2: DNA classification tree 2

accuracy: 45,80%	true coding	true non-coding
predicted coding	262	346
predicted non-coding	738	654

Table 7.3: DNA classification tree 3

possible to extend graphconverter such that it can read and write RapidMiner's format by reverse engineering. This enables us to train decision trees with RapidMiner, export them, merge them using the decisiondiagramplugin and load the merged version again into RapidMiner. There the performance can be evaluated. Figure 7.2 shows the adjusted architecture from Figure 6.2 using the new capabilities of graphconverter.

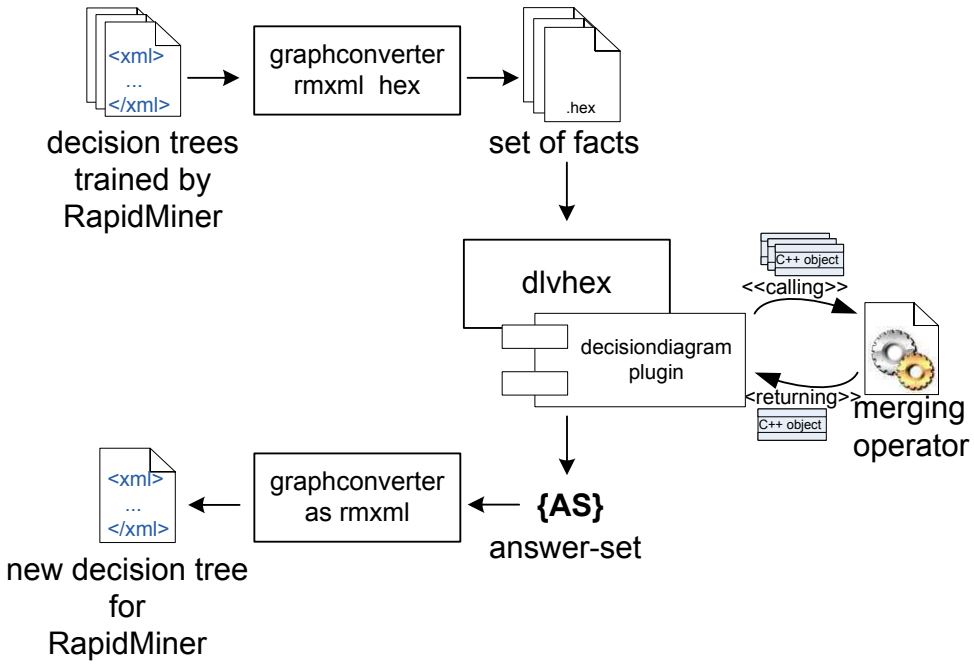

Figure 7.2: Instantiation of the abstract schema for DNA classification

Finally we come to the results. Figure 7.3 shows the merged decision tree according to Salzberg's approach.

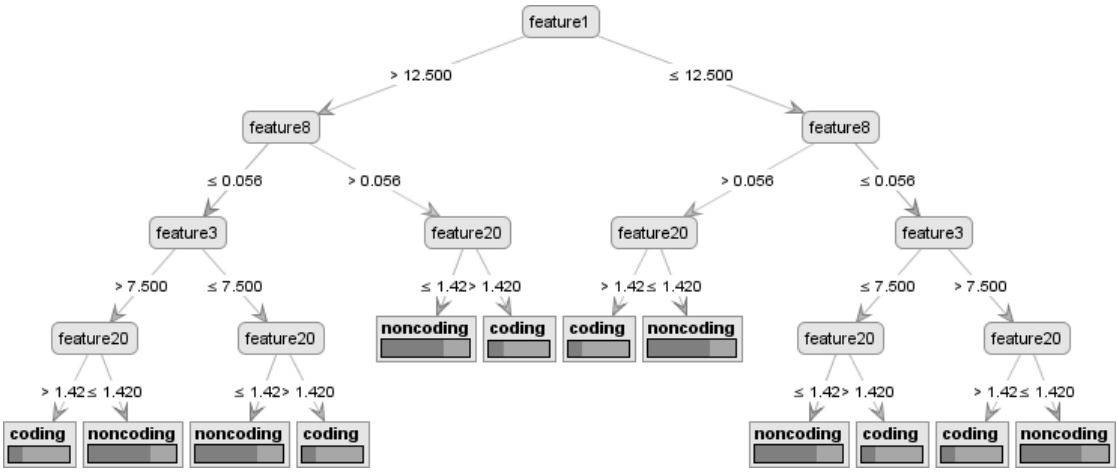

Figure 7.3: Merged decision tree

The evaluation results are surprisingly good. The overall performance was 65,25% accuracy, which is much better than the best result of the source classifiers (see Table 7.4). Remember that we used only very few training examples to train the individual decision trees. This accuracy cannot be enhanced much by using more training samples or source classifiers. Empirical experiments have shown that about 70% is the best one can expect. This comes from the limits of statistical features like those we used [Peng et al., 2005].

accuracy: 65,25%	true coding	true non-coding
predicted coding	565	260
predicted non-coding	435	740

Table 7.4: Merged DNA classification tree

Experiments have shown that about 1000-2000 training examples are needed to reach this accuracy with a single decision tree. Additionally, the single tree would have a depth greater than 3 (as the merged tree). A further advantage of this approach is that it is well-suited for parallel computing settings since the source classifiers can be trained simultaneously. Though the actual strength of the framework is that it offers the possibility to try out several strategies and evaluate the results in a very convenient way. For instance, if we like to check out the influences of the rule "in doubt, classify it as coding" in order to raise sensitivity (and consequently lowering specificity), this could easily done by using the user-preference operator instead (Section 5.2). The updated diagram can then be computed fully automatically. This allows us to compare the results without redoing manual merging between.

7.2 Multidimensional Indices

A common problem when working with multidimensional data is the creation and maintenance of efficient index structures. In the one-dimensional case quite efficient strategies exist. Balanced trees like AVL trees and B-trees enable database systems to insert and lookup arbitrary entries in time $O(\log(n))$ using binary search.

In the multidimensional case however, an obvious problem is that there does not exist a total ordering of the keys, because the relations can be different in the single dimensions. Consider for instance the tuples $a = (1, 3)$ and $b = (2, 2)$. Then a is greater than b in the second dimension, but b is greater than a in the first one. The apparently simple solution of ordering the tuples primarily by their first component, in case of equality by their second component, and so on, has the great disadvantage that it assigns different weights the single dimensions. This ordering only allows for an efficient binary search or answering of range queries in the first dimension, but not in the higher dimensions.

This problem makes it impossible to implement a structure that works in $O(\log(n))$ for *any* query and insertion, even though some structures and heuristics have a quite good behavior in average case.

Applications in Medicine

Even though multidimensional index structures are not only useful in medicine but also in general database systems, there are special medical application scenarios. One of them comes from the field of case-based reasoning.

Case-based reasoning can be summarized as follows. The current case is described by several information chunks about the patient. This includes symptoms, laboratory results, health care history as well as personal information like age, sex and life style. This results in a *multidimensional* description of the current case. If the same procedure was applied to previous cases and they are stored together with the applied treatment, one can be interested in those cases that were similar to the current one. The idea is simple. If patients had similar conditions, it is probable that also a similar treatment will lead to success.

This was studied in more detail in [Althoff et al., 1998] and [Wess et al., 1993]. We will now discuss some existing data structures that allow such applications in principle.

Existing Index Structures

R-Trees

One of these multidimensional data structures is the R-tree [Guttman, 1984]. We are given a set of d-dimensional objects:

$$S = \mathbb{R}^{2d}$$

Note that each element has not only a position in the d-dimensional space but also a *size*, thus we need $2d$ numbers to describe it.

Example 7.3. Let $d = 2$ be our dimensionality. The set of elements $s_i \in S$ is depicted in Figure 7.4.

Then the corresponding R-tree (with at most $m = 3$ elements per node) is shown in Figure 7.5.

Note that even though elements belong to exactly one parent node, they may overlap with some other node. In the example s_2 is contained in R_2. Nevertheless it also overlaps with R_1.

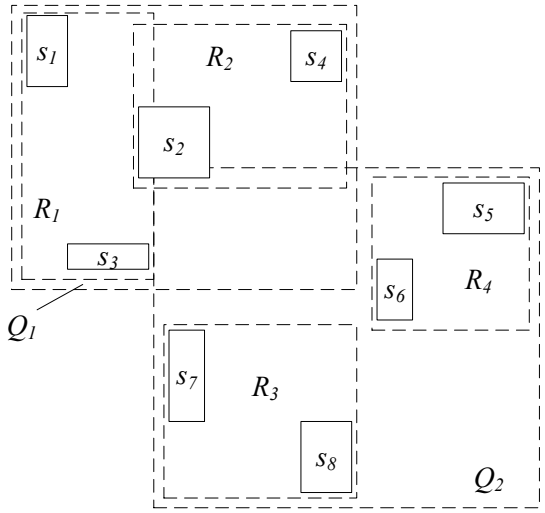

Figure 7.4: Two-dimensional data

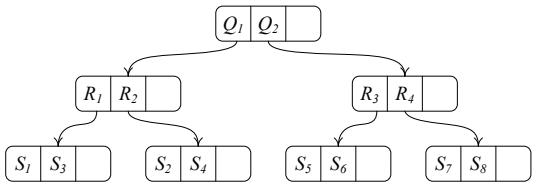

Figure 7.5: R-Tree

This shows that in general we need to search in *multiple* subtrees when we need to lookup an element.

A further difference to one-dimensional trees is that the structure of the tree is not unique. It is not clear a priori which elements to group in each step. A usual heuristic is for instance, to minimize the wasted space in each grouping step, i.e., to minimize the area enclosed by some node where no data elements are. This is driven by the idea that we want to minimize node dimensions in order to prevent overlappings whenever possible. This enhances the performance because overlappings can lead to the problem that multiple subtrees need to be traversed, as explained above.

The problem with R-trees is not only that the lookup of an element needs $O(n)$ in the worst case, but also that it is a static index. That is, if elements are inserted it can become unbalanced and a reinsertion of all elements into a new tree needs to be done from time to time in order to keep the tree balanced.

Nevertheless they have an important advantage, namely that they allow range queries. This is a query of form:

$$\{b_1 \leq X_i \leq b_2 \mid b_1, b_2 \in \mathbb{R}, 1 \leq i \leq n\}$$

Informally, we set a lower and an upper bound for each dimension and want to retrieve all elements that are in between.

k-d Trees

k-d-trees are a direct expansion of one-dimensional binary trees. As in the one-dimensional case, in each node one of the elements is selected such that half of them is smaller and half of them is greater (or equal). Then the elements are equally distributed into the left and right subtree of the node.

The difference in k-d-trees is that now we need to consider multiple dimensions. This is done in a straightforward way. In the root, we just look at the first dimension. That means we sort the elements by its first value and distribute them. In the next level, we only look at the second dimension and so on. In general, at level l we look at dimension $l \mod n$, where n is the total dimensionality.

Figure 7.6 shows a typical partition in 2-dimensional space as represented by a k-d-tree.

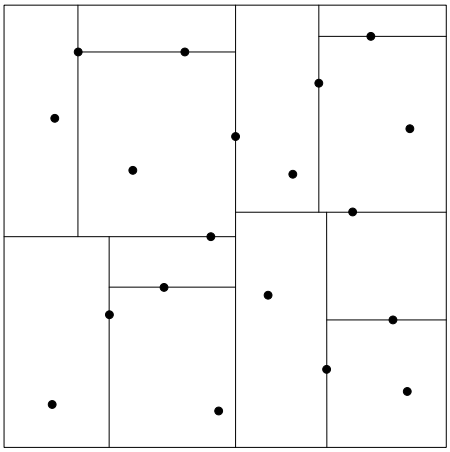

Figure 7.6: Partitioning by a k-d-tree

As with R-trees, the problem is still that the tree not necessarily remains balanced when additional elements are inserted. This leads to the last data structure we will mention.

Multidimensional Binary Trees

It turns out that the most straight forward implementation can have major advantages if implemented appropriately. In [Gonzalez, 2000] one treats the multidimensional elements as one-dimensional keys. This means, a vector of form

$$(v_1, v_2, \ldots, v_d)$$

is just interpreted as concatenation of its elements:

$$v_1 v_2 \cdots v_d$$

This allows us to use a one-dimensional balanced binary tree, like an AVL-tree, to store multidimensional data. Since such a tree allows all operations to be done in time $O(\log(n))$, the complexity of the naive implementation of the d-dimensional version is $O(d \cdot \log(n))$ because we need to compare up to d (sub)-keys in each node. In was shown that this complexity can be reduced to $O(d + \log(b))$ by using a more complex algorithm which prevents sub-keys from being compared multiple times on a path through the tree [Gonzalez, 2000].

The major disadvantage of this tree is that it does *not* allow range queries. Note that the elements are primarily ordered by their value in the first dimension. Only if they are equal, they are ordered by their second value and so on. Other strategies were proposed for the mapping of multidimensional keys to one dimension. For instance, instead of concatenating the v_i one could also interleave them bitwise, i.e.:

$$v_1[1]v_2[1]\cdots v_d[1]\cdots v_1[2]v_2[2]\cdots v_d[2]\cdots v_1[m]v_2[m]\cdots v_d[m]$$

where m is the number of bits in each subkey and $v_i[k]$ denotes the k-th bit of subkey v_i. This still gives a higher priority to a dimension i than to j when $i < j$, but it minimizes this undesired preference.

Merging of Case Repositories

In Section 7.2 we have already discussed the relevance of multidimensional data structures for case-based reasoning. Now we come back to the main topic of the thesis, the merging of decision trees.

An obvious scenario is the merging of case repositories. Imagine several health care institutes that maintain their own databases with historical records. However, for case-based reasoning we may increase the quality of the suggestions by merging the underlying repositories.

From the proposed data structures, multidimensional binary trees (Section 7.2) are best when range queries are irrelevant and k-d-trees (see 7.2) if they are relevant. Multidimensional binary trees and k-d-trees can be encoded as decision trees very easily. Each node contains only one (composed) key and has exactly two subtrees. In contrast to that, representing R-trees as decision diagrams leads to a lot of overhead, since we need to compare two boundaries for d dimensions for each subnode. For instance in the root node of Example 7.3 we would need to compare $2 \cdot 2 \cdot 3 = 12$ values. Due to the syntax of our query language (as introduced in Section 3.2), we cannot actually put 8 comparisons into one node, but we would need to create 8 nodes in sequence (or modify our query language).

A second reason why binary trees are suggested is the fact that insertions can be done in time $O(d + \log(n))$ when we implement the improved procedure from [Gonzalez, 2000]. This allows us to merge two trees T_1 and T_2 with n_1 and n_2 elements in time $O(n_2 \cdot (d + \log(n_1 + n_2)))$ by successive insertion of elements of T_2 into T_1. The result will be a balanced tree.

7.3 Aggregation of Hypothesis in Fault Diagnosis Tasks

To conclude this chapter we consider an application scenario that does *not* deal with decision diagrams and therefore is beyond the scope of this thesis. Nevertheless it is of medical interest, therefore we summarize it shortly. For a detailed discussion we refer to [Redl, 2010].

Propositional abduction problems deal with the finding of explanations for observed (fault) behavior. This is not only useful in technical systems where we deal with malfunctioning systems, but also in medicine where possible reasons for observed symptoms shall be derived. It is rarely the case that we have only one expert and mostly several opinions do not coincide. For instance, if we ask several medical doctors about their hypothesis, we will possibly hear differing answers. This comes from the fact, that different persons come with different expertise and in most situations they will "reason" with unequal theories in mind.

In such situations it is highly desirable to combine the individual decisions into a single consistent group decision. This aggregated judgment should be as similar to the single experts and at the same time it must explain the observations, i.e., it must still be a solution to the abduction problem.

More extended variants of the task deal with making rational group decisions even if the individuals potentially behave irrational. For a detailed discussion we refer to [Redl, 2010].

The best way to predict the
future is to invent it.
Alan Kay

Chapter 8

Conclusion and Outlook

8.1 Problem Statement

Decision diagrams are an important aid for clinical decision making. This mainly comes from the fact that they are intuitively understandable even for non-knowledge engineers.

We can even find application scenarios where we do not only deal with single decision diagrams but with several ones that are similar but not equivalent. As an example we considered DNA classification, where a given sequence needs to be labeled with "coding" or "non-coding", depending on certain statistical features that can be computed for each sequence. In this scenario, one can train several classifiers in parallel, where they differ from each other because of randomization in the learning procedure, random selection of the training set or application of different training algorithms. Clearly, at the end of the day we want just one diagram rather than a set of diagrams. It is therefore interesting to take a closer look at possible merging strategies.

Many people have observed that using different classifiers has advantages over using a single one, namely the possibility of using parallel computing architectures and accuracy enhancements. During our experiments we could observe that the merged tree possibly deliver better results than the individuals, if different training algorithms are combined. However, the quality of the resulting classifier depends on several influence factors, especially the parameters of the algorithms, the training set and the merging algorithm.

Of course, diagrams can also merged manually. But this has the serious disadvantage that this task needs to done again each time the input classifiers or the parameters of the merging operators are changed. Because of the fact that it is often not clear right from the beginning which strategy will behave best, and we want to make experiments with different settings, this can be very tedious. Therefore it is highly interesting to support this task with a tool that takes this burden from the user.

8.2 Solution

In this thesis we have developed a procedure for the semi-automatic incorporation of multiple decision diagrams into a single one. However, we did not hard-code one specific merging algorithm, but we rather developed a fairly *flexible framework*. This allows the user to specify the

desired merging strategy declaratively by selecting appropriate *merging operators* from a set of predefined ones, which can easily be extended by custom operators.

The main advantage of this flexibility is the support of rapid prototyping. This allows the user to experiment with different merging strategies, evaluate results empirically, and change the settings quickly. Without a tool for automatic incorporation, the merging, which is a routine task, would have to be done by hand after each modification; clearly this makes it difficult to focus on the interesting parts of the task.

Basically the framework consists of two parts. The first one is a utility that encodes decision diagrams as sets of facts. This is necessary to make them accessible from dlvhex. Without this translation mechanism, it would not be possible to compute upon an inherently hierarchical data structure like trees or general acyclic graphs.

After this has been done, we can start to merge several diagrams into a single one. For this purpose we use the mergingplugin, which adds support for belief merging tasks to dlvhex, and extend it with merging operators that are specialized to decision diagrams. The framework provides both unary modification and n-ary merging operators. While the n-ary operators with $n > 1$ actually support the merging process, the unary ones allow the modification and simplification of single diagrams. This allows us to first reduce complicated task instances to simple ones, which makes the development and implementation of the merging strategies much easier. For instance, by unfolding acyclic graphs to trees, we only need to develop merging operators for trees, but can still process general diagrams. The framework is extensible, i.e., it is very easy to refine the operators provided, and to implement additional, application dependent ones.

In Chapter 7 we have demonstrated the usefulness of the framework when we investigated a problem from molecular biology which was already briefly mentioned. We have trained several decision trees for DNA classification ("coding" or "non-coding") and merged them subsequently. This example has shown, that any changes in the merging strategy only requires minimal modifications in the formal task description, but no manual remerging of the underlying training data. During experimenting with different combinations of training algorithms, we could increase the accuracy of the final diagram. However, this strongly depends on the selected algorithms and the training set, i.e., not all combinations necessarily increase the quality; some even decrease it. But this exactly demonstrates the intention of the framework: one can try our different scenarios very quickly, investigate the results, and finally select the best one.

A further advantage when training *multiple* classifiers that are merged afterwards is the possibility of exploiting parallel computing. This is especially useful when working with genetic material, which usually comes in very huge files.

8.3 Future Issues

Several operators for decision diagram merging have been developed and implemented. It has been pointed out that they are only thought of as examples because the best suited procedure is application dependent. Nevertheless there is room for improvements by adding additional operators or by generalizing existing ones by introducing optional parameters.

Some of the developed operators can also be improved qualitatively. For instance, the simplification operator currently implements two strategies for decision diagram shrinking that preserve equivalence. One could extend this by allowing non-equivalence preserving transformations, so called *pruning strategies*. These cut subtrees if they contain only very few training samples which bears the risk of overfitting.

Further improvements concern interoperability. Currently, input decision diagrams are expected to be given in one of two supported input file formats: dot and the proprietary XML

format of the open-source tool RapidMiner. The support of additional file formats may allow processing of diagrams trained by other machine-learning tools.

The major benefit of the implementation of the framework is indisputably to provide a rapid prototyping tool for knowledge engineers where experiments and evaluations with different merging plans can be done in a convenient way.

Appendix *A*

The dot File Format

Table A shows the syntax of the dot file format.

Source: `http://www.graphviz.org/doc/info/lang.html` (visited on August 27, 2010).

graph	:	$[strict](graph\|digraph)[ID]''stmt\_list''$
stmt_list	:	$[stmt[';'][stmt_list]]$
stmt	:	*node_stmt*
		|*edge_stmt*
		|*attr_stmt*
		$\|ID' =' ID$
		|*subgraph*
attr_stmt	:	$(graph\|node\|edge)attr\_list$
attr_list	:	$'['[a\_list]']'[attr\_list]$
a_list	:	$ID['=' ID][','][a\_list]$
edge_stmt	:	$(node\_id\|subgraph)edgeRHS[attr\_list]$
edgeRHS	:	$edgeop(node\_id\|subgraph)[edgeRHS]$
node_stmt	:	$node\_id[attr\_list]$
node_id	:	$ID[port]$
port	:	$' :' ID[':' compass\_pt]$
		$\|' :' compass\_pt$
subgraph	:	$[subgraph[ID]]''stmt\_list''$
compass_pt	:	$(n\|ne\|e\|se\|s\|sw\|w\|nw\|c\|\_)$

Table A.1: Syntax of the dot file format

Appendix B

Command-Line Tool graphconverter

Decision diagrams can be stored in different file formats. While some of them are human-readable, others are better for automatic processing.

Supported Formats

dot

The dot file format[1] has an intuitive syntax and thus it is well-suited for humans. Additionally it is well suited for being visualized using the *dot tools*.

Consider the decision diagram depicted in Figure 6.1. The snippet in Listing B.1 shows its implementation as dot file.

```
digraph G {
        root -> case1 ["A<10"];
        root -> case2 ["A>20"];
        root -> elsecase ["else"];
        root -> case3 ["else"];
        case1 -> case1a ["B<10"];
        case1 -> case1b ["else"];
        case2 -> case2a ["B<16"];
        case2 -> case2b ["else"];
        case1a ["ClassA"];
        case1b ["ClassB"];
        case2a ["ClassA"];
        case2b ["ClassB"];
        case3 ["ClassC"];
}
```

Listing B.1: The decision diagram in Figure 6.1 encoded in dot format

Syntactically correct decision diagrams must

- have exactly one root node (which does not need to be explicitly mentioned, but which is implicitly identified by the fact that it has no ingoing edges)

[1] `http://www.graphviz.org`

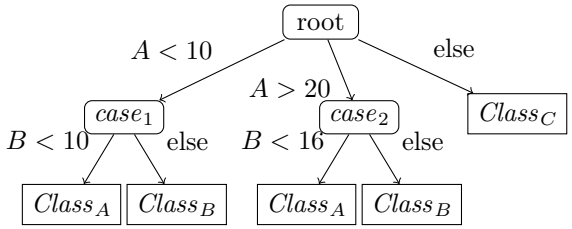

Figure B.1: Graphical representation of a decision diagram

- use only directed edges
- use only edges that are labeled either with

  $$else$$

  or with conditions of form

  $$X \circ Y$$

  where X and Y can be arbitrary strings and $\circ \in \{<, <=, =, >, >=\}$ is an operator
- have leaf nodes that are labeled with arbitrary strings that encode the classification in this node

Note that we talk about the expected syntax of input files to our converter. Not all convertible diagrams necessarily satisfy the semantic validity conditions discussed in Chapter 3.

Sets of Facts

However, dlvhex cannot directly load this format because its input must be a logic program. Thus a diagram must be represented using predicates.

We define the following predicates:

- *root*(X)
  To define that some constant X is defined as the root node
- *innernode*(X)
  To define some constant X to be an inner node
- *leafnode*(X, Y)
  To define that some constant X is a leaf node with label Y
- *conditionaledge*(X, Y, A, C, B)
  To define that a conditional edge with condition $A \circ B$ (where the operation $\circ$ is given by C) leads from node X to Y
- *elseedge*(X, Y)
  To define that an unconditional edge goes from X to Y

The above diagram can therefore be represented as follows.

```
root(root).
innernode(root).
innernode(case1).
innernode(case2).
leafnode(case3, "ClassC").
leafnode(case1a, "ClassA").
leafnode(case1b, "ClassB").
leafnode(case2a, "ClassA").
leafnode(case2b, "ClassB").
conditionaledge(root, case1, "A", "<", "10").
conditionaledge(root, case2, "A", ">", "20").
elseedge(root, case3).
conditionaledge(case1, case1a, "B", "<", "10").
elseedge(case1, case1b).
conditionaledge(case2, case2a, "B", "<", "16").
elseedge(case2, case2b).
```

Listing B.2: The above decision diagram as HEX program

Answer-Sets

A very simple and obvious translation from HEX programs into answer sets is to put all the facts simply as atoms into the answer set. The above diagram can therefore also be implemented as:

```
{root(root),
innernode(root),
innernode(case1),
innernode(case2),
leafnode(case3, "ClassC"),
leafnode(case1a, "ClassA"),
leafnode(case1b, "ClassB"),
leafnode(case2a, "ClassA"),
leafnode(case2b, "ClassB"),
conditionaledge(root, case1, "A", "<", "10"),
conditionaledge(root, case2, "A", ">", "20"),
elseedge(root, case3),
conditionaledge(case1, case1a, "B", "<", "10"),
elseedge(case1, case1b),
conditionaledge(case2, case2a, "B", "<", "16"),
elseedge(case2, case2b)}
```

Listing B.3: the above decision diagram as answer set

RapidMiner XML Format

RapidMiner[2] is an open-source data mining tool. It uses a proprietary XML file format to store decision trees. This format is also supported by the tool introduced in B.1. The details are not relevant for practical work and are skipped therefore. It is only important to know that the import and export functionality for this file format is necessary to process RapidMiner classifiers by the decisiondiagramplugin.

B.1 Conversion

For the conversion between the introduced file formats, the plugin installs a tool called graphconverter. It can be used to translate diagrams in any of the supported file formats into se-

[2] http://www.rapidminer.com

mantically equivalent versions in another format. Assume that the diagram is stored in the file "mydiagram.dot". Then the conversion into the corresponding HEX program is done by entering:

```
graphconverter dot hex < mydiagram.dot > mydiagram.hex
```

The result is a set of facts that can be loaded by dlvhex. After dlvhex has done its job, the output will be an answer set, which is ill-suited for human users. Thus the plugin also supports conversions in the other direction. Assume that the output of dlvhex is stored in the file "answerset.as" (using the *silent mode* such that the output contains the *pure* answer set without any additional information about dlvhex). Then the conversion is done by:

```
graphconverter as dot < answerset.dot > out_diagram.dot
```

Between the two converter calls, the diagram is given as HEX program "mydiagram.hex" that can be processed by dlvhex. Even though one can essentially do anything with this program that is computable, it is strongly intended to be used as part of the input for a revision task.

Note that graphconverter reads from standard input and writes to standard output. The graphconverter expects either one or two parameters. If one parameter is passed, it can be anything of:

- `--toas`
  Converts a dot file into a HEX program.
- `--todot`
  Converts an answer set into a dot file.
- `--help`
  Displays an online help message.

Note that `--toas` and `--todot` are only abbreviations for commonly used conversions. The more general program call passes two parameters, where the first one states the source format and the second one the desired destination format. Both parameters can be anything from the following list.

Format	Parameter name
dot graph	dot
HEX program	hexprogram or hex
answer set	answerset or as
RapidMiner XML	rmxml or xml

Bibliography

[Althoff et al., 1998] Althoff, K., Bergmann, R., Wess, S., Manago, M., Auriol, E., Larichev, O. I., Bolotov, E., Zhuravlev, Y. I., and Gurov, S. I. (1998). Case-based reasoning for medical decision support tasks: The inreca approach. In *Artificial Intelligence in Medicine 12*, pages 25–41. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.9763`.

[Bahar et al., 1993] Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., and Somenzi, F. (1993). Algebraic decision diagrams and their applications. *Computer-Aided Design, 1993. ICCAD-93. Digest of Technical Papers., 1993 IEEE/ACM International Conference on*, pages 188 –191.

[Bennett, 1994] Bennett, K. (1994). Global tree optimization: A non-greedy decision tree algorithm. In *Computing Science and Statistics*, pages 156–160. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.7.393`.

[Bern et al., 1996] Bern, J., Meinel, C., and Slobodova, A. (1996). Global rebuilding of OBDDs avoiding memory requirement maxima. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(1):131 –134.

[Bodlaender, 1993] Bodlaender, H. L. (1993). A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8755`.

[Bollig and Wegener, 1996] Bollig, B. and Wegener, I. (1996). Improving the variable ordering of OBDDs is np-complete. *Computers, IEEE Transactions on*, 45(9):993–1002. URL: `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=537122`.

[Breiman, 1996] Breiman, L. (1996). Bagging predictors. *Machine Learning*, 24(2):123–140. URL: `http://dx.doi.org/10.1007/BF00058655`.

[Bryant, 1992] Bryant, R. E. (1992). Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318. URL: `http://portal.acm.org/citation.cfm?id=136043`.

[Bryant and Bryant, 1992] Bryant, R. E. and Bryant, A. E. (1992). Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24:293–318. URL: http://portal.acm.org/citation.cfm?id=136043.

[Dasgupta et al., 1979] Dasgupta, P. S., Hammond, P. J., and Maskin, E. S. (1979). The implementation of social choice rules: Some general results on incentive compatibility. *Review of Economic Studies*, 46(2):185–216.

[Dietterich, 2000] Dietterich, T. G. (2000). Ensemble methods in machine learning. In *INTERNATIONAL WORKSHOP ON MULTIPLE CLASSIFIER SYSTEMS*, pages 1–15. Springer-Verlag. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.4718.

[Eiter et al., 2005] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2005). A uniform integration of higher-order reasoning and external evaluations in answer-set programming. In *In Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI-05*, pages 90–96. Professional Book. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.8944.

[Eiter et al., 2006] Eiter, T., Ianni, G., Schindlauer, R., and Tompits, H. (2006). dlvhex: A system for integrating multiple semantics in an answer-set programming framework. In *WLP*, pages 206–210. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.118.5386.

[Fickett and Tung, 1992] Fickett, J. W. and Tung, C. S. (1992). Assessment of protein coding measures. *Nucleic acids research*, 20(24):6441–6450. URL: http://view.ncbi.nlm.nih.gov/pubmed/1480466.

[Fitting, 1999] Fitting, M. (1999). Fixpoint semantics for logic programming - a survey. *Theoretical Computer Science*, 278:25–51. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.7641.

[Fujita et al., 1991] Fujita, M., Matsunaga, Y., and Kakuda, T. (1991). On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 50–54, Los Alamitos, CA, USA. IEEE Computer Society Press. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00206358.

[Gabbay et al., 2009] Gabbay, D. M., Rodrigues, O., and Pigozzi, G. (2009). Connections between belief revision, belief merging and social choice. *J. Log. Comput.*, 19(3):445–446. URL: http://people.stfx.ca/mimam/Stuff/Search%20Articles/October%203%20Search/GabbayPigozziRodrigues.pdf.

[Gelfond and Lifschitz, 1988] Gelfond, M. and Lifschitz, V. (1988). The stable model semantics for logic programming. *In Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.2912.

[Gelfond and Lifschitz, 1991] Gelfond, M. and Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.7150.

[Gonzalez, 2000] Gonzalez, T. F. (2000). Simple algorithms for the on-line multidimensional dictionary and related problems. *Algorithmica*, 28(2):255–267.

[Guttman, 1984] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 47–57. ACM. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.131.7887`.

[Hall et al., 1998] Hall, L., Chawla, N., and Bowyer, K. (1998). Decision tree learning on very large data sets. *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, 3:2579 –2584 vol.3.

[Kolter and Maloof, 2003] Kolter, J. and Maloof, M. (2003). Dynamic weighted majority: a new ensemble method for tracking concept drift. *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 123 – 130.

[Koutsoos et al., 1993] Koutsoos, E., Os, E. K., North, S. C., Compsparc, I., Sparccm, S., Sparcasemit, S. S., and Intnulld, I. I. (1993). Drawing graphs with dot. URL: `http://www.graphviz.org/Documentation/dotguide.pdf` (visited on August 27, 2010).

[Kowalski, 1974] Kowalski, R. (1974). Predicate logic as programming language. *in Proceedings IFIP Congress*, pages 569–574. URL: `http://www.doc.ic.ac.uk/~rak/papers/IFIP%2074.pdf`.

[Lenders and Baier, 2005] Lenders, W. and Baier, C. (2005). Genetic algorithms for the variable ordering problem of binary decision diagrams. In Wright, A. H., Vose, M. D., De Jong, K. A., and Schmitt, L. M., editors, *Foundations of Genetic Algorithms*, volume 3469 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg.

[Maclin and Opitz, 1997] Maclin, R. and Opitz, D. (1997). An empirical evaluation of bagging and boosting. In *In Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 546–551. AAAI Press. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.49.8401`.

[Mair et al., 1995] Mair, J., Smidt, J., Lechleitner, P., Dienstl, F., and Puschendorf, B. (1995). A Decision Tree for the Early Diagnosis of Acute Myocardial Infarction in Nontraumatic Chest Pain Patients at Hospital Admission. *Chest*, 108(6):1502–1509.

[Moret et al., 1980] Moret, B. E., , and R. C. Gonzalez, M. T. (1980). The activity of a variable and its relation to decision trees. *ACM Trans. Program. Lang. Syst.*, 2(4):580–595. URL: `http://portal.acm.org/citation.cfm?id=357114.357120`.

[Mountain, 1986] Mountain, C. F. (1986). A New International Staging System for Lung Cancer. *Chest*, 89(4 Supplement):225S–233S. URL: `http://chestjournal.chestpubs.org/content/89/4_Supplement/225S.short`.

[Peng et al., 2005] Peng, H., Long, F., and Ding, C. (2005). Feature selection based on mutual information: Criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1226–1238. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.63.5765`.

[Polikar, 2009] Polikar, R. (2009). Ensemble learning. *Scholarpedia*, 4(1):2776.

[Przymusinski, 1991] Przymusinski, T. C. (1991). Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.1434`.

[Quinlan, 1987] Quinlan, J. R. (1987). Simplifying decision trees. *Int. J. Man-Mach. Stud.*, 27(3):221–234.

[Redl, 2010] Redl, C. (2010). Development of a belief merging framewerk for dlvhex. Master's thesis, Vienna University of Technology, Institute of Information Systems, Knowledge-Based Systems Group, A-1040 Vienna, Karlsplatz 13.

[Salzberg, 1995] Salzberg, S. (1995). Locating protein coding regions in human dna using a decision tree algorithm. *Journal of Computational Biology*, 2:473–485. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.6046`.

[Salzberg et al., 1998] Salzberg, S., Delcher, A. L., Fasman, K. H., and Henderson, J. (1998). A decision tree system for finding genes in dna. *Journal of Computational Biology*, pages 667–680. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.6908`.

[Shortliffe et al., 1979] Shortliffe, E., Buchanan, B., and Feigenbaum, E. (1979). Knowledge engineering for medical decision making: A review of computer-based clinical decision aids. *Proceedings of the IEEE*, 67(9):1207 – 1224.

[Sobin et al., 2009] Sobin, L., Gospodarowicz, M., and Wittekind, C. (2009). *TNM Classification of Malign Tumors*. Wiley-Liss, 7 edition. URL: `http://www.uicc.org`.

[Stevens, 1946] Stevens, S. S. (1946). On the theory of scales of measurement. *Science*, 103(2684):677–680. URL: `http://web.duke.edu/philosophy/bio/Papers/Stevens_Measurement.pdf`.

[Tani and Imai, 1994] Tani, S. and Imai, H. (1994). A reordering operation for an ordered binary decision diagram and an extended framework for combinatorics of graph. *ISAAC'94*, pages 575–583.

[Wess et al., 1993] Wess, S., Althoff, K., and Derwand, G. (1993). Using k-d trees to improve the retrieval step in case-based reasoning. In *Stefan Wess, Klaus-Dieter Althoff, M. M. Richter*, pages 167–181. Springer-Verlag. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.5469`.

[Williams, 1990] Williams, G. J. (1990). *Inducing and Combining Decision Structures for Expert Systems*. PhD thesis, Australian National University, The Australian National University. URL: `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.4523`.

[Williams, 1964] Williams, J. W. J. (1964). Algorithm 232 heapsort. *Communications of the ACM*, 7(6):347–348.

[Zhu, 2005] Zhu, X. (2005). Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison. URL: `http://pages.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf`.