Antonina Danylenko

Decisions: Algebra and Implementation

Licentiate Thesis

Computer Science

2011

Linnæus University

A thesis for the Degree of Licentiate of Philosophy in Computer Science.

Decisions: Algebra and Implementation

Antonina Danylenko

Linnæus University
School of Computer Science, Physics and Mathematics
SE-351 95 Växjö, Sweden
<http://www.lnu.se/>

Abstract

Processing decision information is a constitutive part in a number of applications in Computer Science fields. In general, decision information can be used to deduce the relationship between a certain context and a certain decision. Decision information is represented by a decision model that captures this information. Frequently used examples of decision models are decision tables and decision trees. The choice of an appropriate decision model has an impact on application performance in terms of memory consumption and execution time. High memory expenses can possibly occur due to redundancy in a decision model; and high execution time is often a consequence of an unsuitable decision model. Applications in different domains try to overcome these problems by introducing new data structures or algorithms for implementing decision models. These solutions are usually domain-specific and hard to transfer from one domain to another.

Different application domains of Computer Science often process decision information in a similar way and, hence, have similar problems. We should thus be able to present a unifying approach that can be applicable in all application domains for capturing and manipulating decision information. Therefore, the goal of this thesis is (i) to suggest a general structure (Decision Algebra) which provides a common theoretical framework that captures decision information and defines operations (signatures) for storing, accessing, merging, approximating, and manipulating such information along with some general algebraic laws regardless of the used implementation. Our Decision Algebra allows defining different construction strategies for decision models and data structures that capture decision information as implementation variants, and it simplifies experimental comparisons between them.

Additionally, this thesis presents (ii) an implementation of Decision Algebra capturing the information in a non-redundant way and performing the operations efficiently. In fact, we show that existing decision models that originated in the field of Data Mining and Machine Learning and variants thereof as exploited in special algorithms can be understood as alternative implementation variants of the Decision Algebra by varying the implementations of the Decision Algebra operations. Hence, this work (iii) will contribute to a classification of existing technology for processing decision information in different application domains of Computer Science.

Key-words: Classification, decision algebra, decision function, decision information, learning

This thesis is based on the following refereed publications:

- Antonina Khairova, Jonas Lundberg, Welf Löwe. Decision Algebras for Capturing and Manipulating Decision Information (Best Doctoral Forum Poster Award). Tenth SIAM International Conference on Data Mining (SDM 2010).
- Antonina Danylenko, Christoph Kessler, Welf Löwe. Comparing Machine learning Approaches in Context-Aware Composition. Tenth International Conference on Software Composition (SC 2011).
- Antonina Danylenko, Jonas Lundberg, Welf Löwe. Decisions: Algebra and Implementation. Seventh International Conference on Machine Learning and Data Mining (MLDM 2011).

Related refereed publications which are not part of this thesis:

- Tobias Gutzmann, Antonina Khairova, Jonas Lundberg, Welf Löwe. Towards Comparing and Combining Points-to Analyses. Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009).

Acknowledgments

When I started my PhD studies, writing a licentiate thesis and finalizing it felt like an unreachable goal comparable to a journey to the base camp of the mountain Everest: sometimes you do not have enough equipment, skills, knowledge or oxygen, but nevertheless you must have an irresistible desire to reach the goal. Therefore, this thesis would not be possible without people who helped and supported me during this journey by providing the equipment, improving and developing my skills and knowledge, and supplying the oxygen when I needed it.

First of all I would like to express my gratitude to my supervisors who made an enormous contribution to my academic, research and even personal development and without whose tremendous support this thesis would not have been possible. I am sincerely thankful to Professor Welf Löwe for being my supervisor, sparking my interest to this research problem, sharing his knowledge and thus inspiring the course of this thesis. I am grateful for our informative, challenging and productive discussions and for his ability to change my mood for the whole working week by one supporting word. Special gratitude goes to Associate Professor Jonas Lundberg for his support, interesting discussions, suggestions and comments that helped me gain a different perspective to the research problem. Thank you for your good sense of humor and positive attitude that made the working environment pleasant and informal.

I would have never started my journey without endless enthusiasm and hard work of Associate Professor Tetyana Shatovska, Associate Professor Victoria Repka and Professor Marcelo Milrad who first started the university collaboration program between Kharkiv National University of Radioelectronics and Linnaeus University through which I learned about the opportunity to apply for PhD studies.

During this journey I was surrounded by outstanding and great people, such as my close friends and colleagues Tobias Gutzmann and Therése Rosdahl Nilsson, Bahtijar Vogel and Miranda Kajtazi, Rüdiger Lincke, Sadaf Salavati, Ekaterina Yurova, and Elena Soboleva. I appreciate their friendship and time that we spent together regardless all the hard work we usually had to do. Especially, I want to thank Tobias for proofreading my thesis and providing interesting and insightful comments. Without your help I would have missed plenty of "a"s and "the"s, and people would have a hard time reading this thesis. I also want to thank my best friend Olga Priadka simply for being there in my life. Additionally, I want to acknowledge all my colleagues and friends from DFM department for being great and supportive colleagues and for interesting discussions we usually have outside the university.

My parents Nina and Alik Khairovs and my grandparents are people who always stand by me and who always loved me regardless of my achievements, knowledge or degrees. Without your commitment and careful education that you gave me I would not write these acknowledgment words today. I am sorry, mom, for all your tears every time when I leave home and for our rare talks on Skype because of the lack of time. You all are those people whose support and love accompany me during all my life.

I want to thank the whole "Ukrainian community" for our friendship, and for feigning interest in my research topic when I raved about it over table conversations. I am thankful to Vlad Aleksakhin and Yuriy Tarasov for being good friends and for sharing the lonely moments of cold and rainy evenings of Växjö, to Alona and Kyrylo Myronovs, Yuriy and Yulia Glebovs for long friendship and trust that came over from Ukraine to Sweden. Thank you all Ukrainian students who came here for this study period and have become my good friends, without you my life would be boring.

However, there is a person in my life who is the most important of all to me and who always stands by me in both joyful and disappointing moments. This person is my beloved husband Oleg. Without his support, love and patience I would have never stepped on the path of PhD studies. Thank you for loving me, tolerating me, not complaining too much about my quite often bad mood and making sure that I have plenty to eat and to drink while working. You are the one who supplies me with oxygen during this "climbing", and without you I would have already gotten an Altitude sickness.

Contents

Abstract	iii
Acknowledgments	vii
1 Introduction	1
1.1 Research Goals	2
1.2 Research Goal Criteria	2
1.3 Outline of Intended Approach	3
1.4 Motivation	5
1.5 Thesis Outline	8
2 Decision Information: Motivation and Example	9
2.1 Decision Information	10
2.2 Classification	13
2.3 Points-to Analysis	17
2.4 Context-Aware Composition	29
2.5 Related Work	32
2.6 Conclusions	36
3 Decision Algebra based on Higher Order Functions	39
3.1 Decision Functions	40
3.2 Learning Decision Functions	42
3.3 Deciding Using Decision Functions	44
3.4 Decision Function Operations	45
3.5 Decision Lattice	48
3.6 Core Operations of the Decision Algebra	53
3.7 Summary	53
4 Algorithms using Decision Algebra	55
4.1 Decision Capturing	55
4.2 Decision Interpolation	56
4.3 k-approximation	57
4.4 Pruning	58
4.5 Conclusions	60

5	Decision Models: Possible Implementations	62
5.1	Decision Trees	63
5.2	Decision Graphs	66
5.3	Dispatch Tables	68
5.4	Conclusions	71
6	Probability- and Maximum Margin-based Decision Models Integrated into Decision Algebra	73
6.1	Decision Algebra: Generalization	73
6.2	Naive Bayes Classifier	77
6.3	Support Vector Machines	81
6.4	Conclusions	89
7	Decision Algebra: Implementation and Validation	90
7.1	Decision Graphs vs. Decision Trees	91
7.2	Decision Models in Context-Aware Composition	98
7.3	Summary	106
8	Conclusions	108
8.1	Review of the Goals and Goal Criteria	111
9	Future Work	115
9.1	Improving Theoretical and Practical Aspects of the Decision Algebra	115
9.2	Machine Learning in Software Engineering	116

Appendix

A	Context-Sensitive Strong Updates in Points-to Analysis	119
----------	---	------------

List of Figures

2.1	A general process of using decision relation in a problem domain	11
2.2	A tree representation of a decision function	12
2.3	Inference and Decision step in Decision Theory	13
2.4	Dataset and a corresponding decision tree for classifying physical activity	14
2.5	Example of replication problem in decision trees	15
2.6	A source code example	18
2.7	A source code example with corresponding control flow graph and SSA form	21
2.8	Tree and graph representations of χ -terms	23
2.9	Code example and corresponding control flow graph	25
3.1	Tree and graph representations of $x^1(x^3(1, 2), x^2(x^3(1, 2), 2))$. .	41
3.2	Equivalent decision functions: $df \equiv df'$	42
3.3	Approximation and k-approximation of $x^1(x^3(1, 2), x^2(x^3(1, 4), 2))$	46
3.4	Decision Lattice over a set of decision functions	51
6.1	An optimal hyperplane for classification in two-dimensional space for a linearly separable case [25]	82
6.2	An example of a hyperplane of an SVM gotten from the re-learning on support vectors of two previous SVMs	87
7.1	The percentage of reduced internal nodes and leaves compared to the total tree size (100%)	94
7.2	Times of learning and deciding based on a Decision Graph as % of Decision Tree (100%)	94
7.3	The accuracy gained by pruning Decision Trees and using k-approximated Decision Graphs	97
7.4	Learning and approximation times of Decision Graphs as % of Decision Trees (100%)	98
7.5	Homogeneous Quicksort and Context-Aware Sorting using Decision Graphs ("Opt Graph") and Dispatch Tables ("Opt Table"). The x -axis displays the array size, the y -axis the time in <i>msec</i>	105

List of Figures

A.1	Code example and corresponding control-flow graph	120
A.2	A simple control-flow graph representation for loop	122

List of Tables

2.1	Example of (a) a Decision Table and (b) a Dispatch Table . . .	30
7.1	Dataset Characteristics	92
7.2	Memory overhead of different classification models.	100
7.3	Decision overhead of different classification models.	100
7.4	Errors of different decision approaches.	102
7.5	Time overhead (in %) of different decision approaches.	104

List of Tables

Chapter 1

Introduction

A number of applications in different fields of Computer Science, such as Data Mining, Software Engineering, Artificial Intelligence, etc., are required to process decision information to adjust and alter applications' behavior. Decision information is an information which can be used to deduce the relationship between a certain context and a certain decision. A context is a set of values of properties/attributes that describes a particular situation or task, and a decision is an inference based on this context. As an example of decision information we can consider information that is used in classification problems, e.g., determining the disease based on patient's symptoms.

Decision information is usually represented by a decision model. A decision model is a set of rules that determines a target decision and can be implemented as Decision Tables, Decision Trees, etc. The selection of an appropriate decision model has an impact on application performance in terms of memory consumption and execution time. High memory expenses can possibly occur due to redundancy in the decision model; high execution time is often a consequence of the use of an unsuitable decision model. Choosing an appropriate decision model for a specific task requires a tradeoff between memory consumption, accuracy, and speed for construction and utilization. Additionally, every decision model requires a set of operations for processing decision information to adjust and improve application behavior with respect to specific domain requirements. Applications in different domains try to overcome these problems by introducing new data structures or new algorithms for implementing decision models. These solutions are usually domain-specific and it's hard to transfer achievements from one domain to another.

In this chapter we give an introduction to our chosen research topic in Sections 1.1 and 1.2. Section 1.3 presents an outline of our intended approach. Section 1.4 motivates our choice of research goals and Section 1.5 gives the thesis outline. All concepts and ideas presented here will be revised and further explored in later chapters.

1.1 Research Goals

Based on the observation that applications in different fields of Computer Science often process decision information in a similar way and, hence, have similar problems, we should be able to present a unifying approach that can be applicable in all domains for capturing and manipulating decision information. Therefore, the goal of this thesis is to present a generalized theory for capturing and manipulating decision information. This goal can further be divided into two sub-goals:

1. Theory: Develop a unifying theoretical framework for capturing and manipulating decision information. We will refer to this framework as a Decision Algebra.
2. Implementation: Develop a platform that allows comparing different instantiations of the framework, i.e., decision models implementing the Decision Algebra, and employing these decision models as alternative variants in applications from different domains where processing of decision information is essential.

1.2 Research Goal Criteria

The criteria for fulfilling our first goal are:

- 1.1 The Decision Algebra shall be complete: it shall provide a complete set of operations which is used by known decision models like Decision Trees, Decision Graphs, Decision Tables, Naive Bayes classifier, Support Vector Machines, and others for decision information processing. The complete set of operations shall be specified based on the observations made in concrete problem domains.
- 1.2 The Decision Algebra shall be efficient in terms of time complexity: the implementation of existing decision models as Decision Algebra instantiations shall not impose any complexity overhead compared to if they were implemented as stand-alone models. That is, no algorithm using Decision Algebra operations shall have a higher time complexity than if it was directly using specific decision model operations.
- 1.3 The Decision Algebra shall be efficient in implementation: it shall reduce the number of operations to a few core operations that are able to describe known decision models. This makes the Decision Algebra more reusable as it reduces the work required to implement a new decision model using this set of core operations.

The criteria for fulfilling our second goal are:

- 2.1 It shall be possible to compare different decision models under a common Decision Algebra interface regardless of implementation details.
- 2.2 It shall be possible to improve performance in one domain by using a decision model introduced by another application domain.
- 2.3 It shall be possible to assess non-functional properties for each decision model based on platform validation and deduce different performance profiles.

1.3 Outline of Intended Approach

Our approach to reach the two research goals can be outlined as follows. In order to reach our first goal, we have to provide uniform definitions of decision information, context, decision, and decision model that are valid for any application domain of Computer Science domain where this information is used.

By exploring a number of Computer Science fields where decision making is a part of application performance, we have to be able to show that the information they operate on corresponds to a decision information and, thus, is the same for all domains. As instances of Computer Science fields we take Data Mining with classification application domain and Software Engineering with points-to analysis and context-aware composition application domains. Each application domain has its own functional requirements corresponding to a number of operations to manipulate the stored information, and non-functional requirements corresponding to domain-specific performance specifications. These requirements have to be fulfilled by the decision model representing decision information. Therefore, each application uses a specific decision model with respect to its non-functional requirements. We will show that the set of operations are shared by all application domains. Due to this fact we can generalize a decision model to a unifying theoretical framework, denoted as Decision Algebra, that allows for evaluating different decision model implementations.

The Decision Algebra has to include a formal representation of a decision model that is valid for each application domain, and a set of generalized operations for information manipulation. We introduce higher order *decision functions* as a construct for mapping contexts to pre-computed decisions. Moreover, we define a *learning* phase that is responsible for capturing decision information into a decision function and a *deciding* phase that infers a decision/result by applying the decision function to an actual context. In order to manipulate the information represented by these functions, we

introduce general operations (signatures) for storing, accessing, merging, approximating, and manipulating such information along with some general algebraic laws that are independent of the used implementation. Based on this set of operations we define a minimal set of operations that serves as a basis for all other operations. At this point we should be able to reach research goal criterion 1.1. Furthermore, we introduce existing algorithms for decision model manipulation originated from three application domains and show that they can be implemented using a set of Decision Algebra operations. Several existing approaches for decision models, Decision Trees, Decision Graphs as explicit implementation of the Decision Algebra, and Decision Tables, come out as alternative implementations of our framework. In each decision model, we define an implementation of the minimal set of operations defined in the Decision Algebra along with some auxiliary operations that are specific to each decision model. Although these decision models are usually chosen based on the problem domain, sometimes even on the sample data, we can theoretically estimate non-functional properties, i.e., the bounds on memory consumption and decision time, for every alternative model regardless of the specific problem domain. These estimations will later be used when we assess non-functional properties of decision models in specific problem domains. Based on the discussed algorithms and specific implementations of decision model operations, it should be possible to satisfy research goal criterion 1.2.

However, there exist different types of decision models originated in the field of classification, for example, different types of linear classifiers including probability-based model - Naive Bayes classifier - and maximum margin model - Support Vector Machines. These models cannot represent the same kind of higher order decision functions as discussed in the Decision Algebra. The construction phase and interpretation of basic operations are different. Thus, we define a more abstract signature for these operations. Based on this abstraction we show that existing operations in these decision models have the same interpretation and follow our Decision Algebra representation, even though it becomes more general and, hence, less implementation-specific. By abstracting the Decision Algebra and, hence, abstracting core operations signatures formalizing decision information, we reach goal criterion 1.3.

To fulfill our second goal we explicitly implement the discussed decision models as Decision Algebra instances following one common interface in our platform. Most of the Decision Algebra operations for decision models are defined on a general decision function level regardless of their implementation. Therefore, the comparison between decision models should be rather fair showing advantages and disadvantages of the decision models instead of advantages and disadvantages of different implementations thereof. Thus, to enable a fair comparison, we need to bias:

- Accuracy of the decision, i.e., does a given decision model always decide for the optimal variant, and what is the impact of suboptimal decisions on the overall performance of the application processing decision information,
- Decision time and its impact on the overall performance,
- Memory size for capturing decision information in different decision models.

The validation of decision models is based on two series of experiments: (1) the first experiment compares two decision models based on higher order decision functions: Decision Graphs, explicitly implementing our Decision Algebra, and Decision Trees. In order to make a fair comparison, the construction approach for two decision models is the same. (2) For the second experiment we select an application in the context-aware composition domain which benefits in terms of memory consumption and decision speed from using different alternatives of decision models originated in other domains. The experiment evaluates these alternatives by invoking the best decision chosen based on an actual application context. Moreover, for each decision model we assess non-functional properties in this specific application domain. At this point we should be able to reach our second research goal: we should have a platform implementing different instantiations of our Decision Algebra. Moreover, by a number of experiments we show that our platform allows comparing these instantiations as well as using them in a certain application domain that requires decision information processing.

1.4 Motivation

In this section we motivate our choice of research goals, which follows directly from our research group's work on points-to analysis: our effort to improve points-to analysis precision [48, 22] became a starting point for developing a generalized theoretical framework for describing decision information.

Static program analysis is an analysis that tries to predict the dynamic behavior of programs without running them. Ideally, such analysis should be precise, i.e., the output result should usually be as small as possible, and, at the same time, conservative, i.e., not missing anything. Moreover, exact program analysis is (in all non-trivial cases) intractable and, hence, approximations must be accepted. In order to get better analysis results, we have tried to combine several analyses and thereby improve analysis precision. However, as our experiments showed, combining different analyses did not improve accuracy significantly [22].

Precise program analysis is quite expensive in terms of time and memory consumption, since it must take into account all possible execution paths in an analyzed program. Therefore, another way of improving analysis precision is to exploit the potential of handling these execution paths separately. Furthermore, the execution paths of a program represent a source of decision information that has to be captured in a proper non-memory-consumptive way. In this case, different branches in a path correspond to different contexts and lead to different analysis results, which correspond to different decisions. In order to capture such information in points-to analysis, a data-structure called χ -terms was developed. A χ -term is a compact representation of how different control-flow options, e.g., *if-else* statements, loops, etc., affect the analysis value in a given program point. It was first presented as a formalism by Martin Trapp in his dissertation [74] and was extended and adjusted to fit our needs.

The decision information, in general, is an information that keeps a number of different contexts, where each context corresponds to a set of values of several attributes, based on which we reach a certain result corresponding to a decision. Well-known application domains where decision information is used are classification and pattern recognition domains from the fields of Data Mining and Machine Learning. Data Mining is the process of extracting patterns from large data sets by combining methods from statistics and artificial intelligence with database management. Originally it is the process of the extraction of implicit, previously unknown, and potentially useful information from data [83]. While Machine Learning provides a technical basis for Data Mining, i.e., it deals with the issue of how to build computer programs that improve their performance through experience [49].

In general, capturing decision information in different decision models that map contexts to decisions can be memory consumptive, as they can grow exponentially with the number of attributes and attribute values. Additionally, there exist the problems originated in the classification domain: data replication (redundancy in the stored information) and model overfitting possibly leading to data fragmentation (the amount of information is too small to make a statistically significant decision) [39, 58].

Thus, an important choice to be made is to select a representation of a decision model: the representation of the function that the capturing algorithm, denoted as *learning* algorithm, will use to capture the information in. There exist a number of approaches to adjust existing decision models to specific problems. These approaches involve reducing the amount of required memory for decision information by reducing redundancy in the stored information, reducing the time for learning and deciding (getting the result based on the current context) by approximating decisions based on the captured decision information, etc. Often they improve the *learning* algorithms or the data structures capturing decision models. No single model has been

found to be superior to all others [25]. Issues such as model accuracy, capturing time, robustness, interpretability, and scalability must be considered and can involve tradeoffs further complicating the search for an overall decision model. In general, this choice of the model involves a tradeoff between (1) the expressiveness of the model, i.e., the representation should be as close as possible to the ideal function that covers all the captured data, (2) the memory required to keep this high-detailed decision model and (3) the time and initial data required in order to choose among the alternative decisions it can represent [49].

Introducing new learning algorithms is more or less a well-defined process, since there exists an algebraic framework for presenting decision algorithms and describing various splitting criteria [66]. The data structures most frequently used for keeping decision information (trees and tables) are also well-known, along with efficient implementations once we have specified their operations. However, special learning algorithms might require adding new operations which, in general, have a negative impact on the data structure memory requirements and performance.

In addition to the classification domain there exist a number of application domains in the Computer Science fields where the processing of decision information is a part of applications:

- Static program analysis [48, 22] where different branches in a program execution path lead to different analysis results,
- Context-Aware Composition [1] where components are bound (classified) dynamically based on context properties (attributes),
- Object-oriented languages with multiple dispatch where target methods are selected (classified) based on the dynamic types (attributes) of the actual parameters,
- Chip design, where the logical switching behavior of circuits (binary classification) depends on their input (binary attributes).

Because of this variety of application domains with decision problems (each coming with different notations and tailored implementations) we consider it worthwhile to introduce a generalized theoretical framework for a decision model representation and capturing the decision information, which is applicable in different fields of Computer Science. Due to this generalization, insights can be gained at an abstract level or reused between different application domains, paving the way for a deeper problem understanding and, possibly, for novel and more efficient data structures and algorithms.

The motivation of the second research goal follows directly from the motivation of the previous one and is also based on the variety of domains with

decision problems. The implementation of a framework allows comparing results from different domains, benefiting from the improvements across domain boundaries and giving the opportunity to inherit improvements from other application domains. It will also provide the possibility of improving application performance by replacing a usually used decision model for capturing decision information by an alternative model.

1.5 Thesis Outline

The remainder of this thesis is as follows: In Chapter 2 we present a notion of decision information and give a general introduction to a decision model that captures this information. Moreover, we give an overview of three application domains originated in the Software Engineering and Data Mining Computer Science fields that process decision information: classification, points-to analysis, and context-aware composition. For each domain we introduce a decision model and discuss the domain's non-functional requirements. We conclude the chapter with a related work section followed by a section with conclusions.

Chapter 3 presents the Decision Algebra for capturing and manipulating decision information. Chapter 4 gives an introduction to several algorithms from the presented application domains and presents the descriptions of the algorithms based on the operations denoted by the Decision Algebra.

Afterwards, Chapter 5 and Chapter 6 give a detailed description of the five existing decision models that can be considered as instances of the Decision Algebra and discuss implementation and interpretation of basic operations for each instance. Moreover, Chapter 6 abstracts the defined Decision Algebra in order to formalize different types of decision models. The chapters also provide theoretical estimations of non-functional properties (memory consumption and decision time) for each decision model.

In Chapter 7, we present a platform for our second research goal and show the results of two experiments that support this goal's criteria: (1) a comparison of a straight-forward implementation of the Decision Algebra with one of the existing instantiations, (2) an evaluation of different decision models in the domain of context-aware composition, and (3) an assessment of non-functional properties of these decision model in a specific problem.

Chapter 8 concludes this thesis, and Chapter 9 discusses future work that came out from the application of our Decision Algebra to different application domains of Computer Science. Additionally, Appendix A presents a theoretical application of our Decision Algebra in the program analysis domain.

Chapter 2

Decision Information: Motivation and Example

Observations made in this chapter are used to justify our choice of research goals presented in Chapter 1 and to motivate the theoretical approach we present in Chapter 3. Our structure is the following: we start in Section 2.1 by introducing a general idea to decision information that can be considered as an essential component in different application domains. We also discuss general characteristics of decision information and decision models as a means for representing this information. Basically, this section presents a set of formal definitions that will be used throughout this thesis.

In the following three sections we present a number of application domains where the processing of decision information is an essential part of an application. In each application domain we describe a decision model used to represent corresponding decision information and a set of functional and non-functional requirements for each domain.

In Section 2.2, a classification problem is described and a short overview of Decision Trees as a traditional decision model for representing decision information is given. Section 2.3 gives a general introduction to points-to analysis. We then introduce χ -terms as a means to handle context-sensitivity in data-flow analysis and show how it can be interpreted as a decision model capturing path-dependent reference information. Section 2.4 describes context-aware composition as a technique that is used for automatic selection of optimal variants of program components at runtime. As a decision model that captures optimal variants in context-aware composition we present dynamic Dispatch Tables. Finally, Section 2.5 discusses related work for each application domain.

We conclude this chapter in Section 2.6, where we outline the problems of decision information processing and make several observations in which we motivate the chosen research topic and guide the reader towards the next chapter that presents the theoretical foundation of this thesis.

2.1 Decision Information

The purpose of this section is to introduce a common vocabulary and a set of basic notions that will be used throughout this thesis. These notions characterize decision information used by different kinds of applications to adjust and alter their behavior.

Definition 1. A **decision tuple** $(\vec{a}, c)$ is a tuple that relates an actual context $\vec{a} \in \vec{A}$ with an actual decision $c \in C$, where $\vec{A}$ is a formal context and C is a formal decision. The decision tuple can also be referred to as a decision fact or a training instance.

Notice that we distinguish between an actual context and a formal context. An **actual context** $\vec{a} = (a_1, \dots, a_n)$ is a vector of attribute values $a_i \in A_i$, where A_i is an attribute that corresponds to a property in a certain problem domain.

A **formal context** $\vec{A}$ is the set of all actual contexts $\vec{a} = (a_1, \dots, a_n)$ for all possible $a_i \in A_i$ formed as a result of the Cartesian product $A = A_1 \times A_2 \dots \times A_n$ over sets of values corresponding to attributes $A_1, \dots, A_n$.

Finally, an **actual decision** $c \in C$ is a result of the choice made between alternative decisions in a situation of uncertainty. Hence, a **formal decision** C is a set of all available alternative decisions.

As an example of an actual context we can consider an explicit description of a particular processing point, situation or data instance. As an example of a decision we can consider algorithms, classes, computed values, etc.

Definition 2. A set of decision tuples $DR = \{(\vec{a}_1, c_1), \dots, (\vec{a}_n, c_n)\}$ is a **decision relation**. A decision relation can also be referred to as **decision information**, **dataset**, **training instances** or **training sample**.

Several properties have to be discussed when we present decision relations:

- A decision relation is complete if and only if:

$$\forall \vec{a} \in \vec{A} : (\vec{a}, c) \in DR$$

- A decision relation is non-contradictive if and only if:

$$\forall (\vec{a}_i, c_i), (\vec{a}_j, c_j) \in DR : \vec{a}_i = \vec{a}_j \Rightarrow c_i = c_j$$

More generally, a complete decision relation contains decisions for all possible actual contexts within a given problem domain; a non-contradictive decision relation contains only non-contradictive decision tuples, i.e., there are no two tuples having the same actual context $\vec{a}$ but leading to different decisions in the decision relation. A complete and non-contradictive decision relation represents a *decision function*.

Figure 2.1: A general process of using decision relation in a problem domain

Definition 3. A *decision function* df is a complete and non-contradictive decision relation. A decision function maps a formal context $\vec{A}$ (domain of df) to a formal decision C (co-domain of df):

$$df: A_1 \times \dots \times A_n \rightarrow C$$

In order to use a decision relation in a particular application we need to process it in such a way that we would be able to predict $c \in C$ given a new context $\vec{a}$. That is, we have to construct a specific representation of the decision function that, based on a given context $\vec{a}$, be able to determine a new decision tuple $(\vec{a}, c)$. This representation of a decision function that is capable of handling a new context $\vec{a}$ is denoted as *decision model*.

Definition 4. A *decision model* is a representation of a decision function that provides a set of rules determining relations between actual contexts and a set of decisions.

Figure 2.1 depicts a general model of how a decision relation can be used in a problem domain. A decision model can be of different types. The most obvious decision model for representing a decision relation is a complete set of all decision tuples captured in a table. However, there exist more complicated representations. For instance, the decision model can correspond to a collection of rules that match against the actual context, a mathematical formula that computes a decision based on input parameters, etc. Moreover, the decision model can be a tree $T(N, E)$, where attributes are internal nodes N with attribute values corresponding to outgoing edges E , and decisions are leaves (Figure 2.2). The decision procedure for such a decision model corresponds to a search of a path from a root node to a leaf representing the actual context $\vec{a}$ leading to a certain decision $c \in C$.

Figure 2.2: A tree representation of a decision function

Definition 5. *Learning* is the process of constructing a decision function from a given decision relation.

The learning process always involves *decision capturing*: we capture a *complete* and *non-contradictive* decision relation in a predefined decision function. Additionally, on top of that, we can perform *decision interpolation*: we derive a complete and non-contradictive decision relation when the initial relation is *incomplete* or *contradictive* and we can perform *decision approximation* that approximates a derived decision function.

In order to learn a decision function for a given decision relation, we must first select a proper decision model. In general, the choice of decision model might depend on (1) the problem domain where this decision relation is going to be used, (2) what learning algorithm is going to be used, and (3) what decision algorithm is going to be used for this decision model.

Ultimately, processing a decision relation in order to use it in a concrete problem domain involves two major phases: the first phase is called an inference phase and is responsible for selecting and integrating original data to a set of decision tuples defining decision relation followed by a learning decision function; the second phase is called decision phase and is responsible for interpretation of a learned decision function in order to derive a final decision. More formally:

Definition 6. *Deciding* is a process of applying a decision function to a given actual context $\vec{a} \in \vec{A}$ in order to determine a concrete decision $c \in C$.

These two phases are a subject of Decision Theory [6] that generally describes a set of quantitative methods for reaching an optimal decision and allows us making optimal decisions in situations involving uncertainty,

Figure 2.3: Inference and Decision step in Decision Theory

e.g., pattern recognition, classification, etc. These phases are depicted in Figure 2.3.

In this thesis we concentrate on the first phase of this process, namely on the decision function as a general form for the decision relation.

In the following sections we explore three different Computer Science application domains. Each of them involves the processing of a specific type of decision relation using a particular type of decision. Each application employs different learning approaches to construct a final decision function. However, these domains use different decision models that depend on non-functional requirements which are specific for each domain.

2.2 Classification

The purpose of this section is to introduce a common vocabulary and a few concepts that characterize the classification domain, the decision relation, and the decision model used.

Classification is the task of learning a decision function represented by a *classification model*: $A_1 \times \dots \times A_n \rightarrow C$ that maps a vector of *attribute* values $(a_1, \dots, a_n) \in A_1 \times \dots \times A_n$ to one of the predefined *classes* $c \in C$. The input data for the classification task is a collection of instances $(a_1, \dots, a_n, c)$ that is usually called the *training dataset*. In terms of general decision information representation discussed in the previous section, an instance corresponds to a decision tuple, a vector of attribute values corresponds to an actual context, a classification model corresponds to a decision model, and a training dataset corresponds to a decision relation.

Figure 2.4 shows a simple dataset that can be used as input to the clas-

Figure 2.4: Dataset and a corresponding decision tree for classifying physical activity

sification model recognizing the activity of playing a game. This dataset uses three attributes **Outlook**, **Humidity**, **Windy**, and a class label **Play**. The attribute **Outlook** can have three possible values (**overcast**, **sunny**, **rain**), the attribute **Humidity** is a continuous numeric attribute, and two others are binary (**yes**, **no**).

Decision Models in Classification

There exist several different classification techniques [55, 84], i.e., systematic approaches to construct and represent a decision model given a dataset. Here we will consider Decision Trees as a common used decision model in a classification domain. A *Decision Tree* is a special tree structure consisting of two types of nodes:

- *Leaf nodes* represent class labels c , the result of classifying an attribute set $(a_1, \dots, a_n)$.
- *Internal nodes* represent selections for an attribute A_i referring to different sub-trees for different attribute values a_i .

Each path from the root to a leaf in the Decision Tree represents the attribute values leading to a certain classification result. In Figure 2.4, we present the Decision Tree for our dataset.

To use a Decision Tree for the classification of a given input $(a_1, \dots, a_n)$ is straightforward: in each internal node for attribute A_i , starting at root, the corresponding attribute value a_i is used to decide which child to visit next. We refer to this operation as *decide*.

Leaves may contain classes directly but, in general, we capture distributions over classes representing the number of training instances with the same attribute values supporting a specific class.

Figure 2.5: Example of replication problem in decision trees

Usually, the learning of a Decision Tree from a given dataset is done by a decision interpolation approach, since the dataset is often incomplete and contradictory, or even contains noise, i.e., not valid instances. This operation in the classification domain referred to as *learn*. Efficient learning algorithms using the interpolation approach have been developed to compute Decision Trees for a given dataset R . Most of these algorithms employ a recursive top-down approach that greedily partitions the dataset by choosing a special order of the attributes for each path. Assume an initial dataset R and a root node n :

- *Base Case*: If all training instances in R belong to the same class label c , then n is a leaf with label c .
- *Recursive*: Otherwise, select one attribute A_i and create a child node n for each possible attribute value a_i . Partition R into smaller subsets R^j based on their particular value for the attribute A_i . Apply the algorithm recursively on each child node n using subsets R^j .

The order in which attributes are selected does not influence the information captured in the tree. However, it might have a crucial impact on the size of the resulting Decision Tree. We refer to the operation which reorders the attributes inside the decision tree as *evrt*. Several heuristics for selecting the attribute order are known, most popular is to select the attribute which reduces the entropy metric [55].

The size of the tree can be reduced by replacing subtrees reached by insignificant datasets (e.g., $|R^j| < 5$) with a single leaf node having the most frequent class label, or with a subtree rooted by its child. This type of tree *pruning* can occur during the learning process (*pre-pruning*) or afterwards

(*post-pruning*). Pre-pruning stops the growing of the tree early, before it reaches the point where it correctly classifies the training instances, while post-pruning allows the tree to overfit the dataset and then prunes the tree. Although the first approach seems more direct, the second approach has been found to be more practical. The reason is the difficulty to estimate the stop criterion of growing the tree [49]. Notice that the pruning process in Decision Trees corresponds to the decision approximation approach presented in the previous section. We refer to the process when a subtree is replaced with a single node as *merge*, and the whole pruning process as *approximate*.

Additionally, the dataset in the classification domain can be captured in other models often called *classifiers*, such as probability-based models, e.g. Naive Bayes classifier, maximum margin-based models, e.g., Support Vector Machines, graph-based models, e.g., Bayesian networks, and Decision Graphs, etc. [6, 25]. A probability-based model is learned by computing the probabilities for a particular decision over a dataset, while a maximum-margin model usually builds a hyperplane in n -dimensional space that separates training decision tuples into two classes. A specific decision model is usually chosen based on the non-functional requirements of the given application domain.

Functional Requirements

The required operations to process a decision relation captured in a Decision Tree are: *learn* - to construct a Decision Tree; *decide* - to make a decision based on a constructed Decision Tree; *evert* - to reorder attribute sequences in a Decision Tree; *merge* - to replace a subtree with a single node; *prune* - to reduce the size of a Decision Tree by using merge.

Non-functional Requirements

Classification process requirements are usually application-dependent. That is, the process can be performed under different requirements specified by a particular application's purpose. Therefore, classification is flexible in terms of learning and decision time. Often the learning time is not so crucial, while the decision should be done in seconds. An example is determining a solvency of a person in a bank while processing a person's request for credit. However, the learning time becomes quite important in the situations of online learning of the system. As an example, consider the learning of a spam filter without knowing in advance the best performing model, where the spam filter has to be learned online based on the incoming instances. Hence, based on the task we can bias a decision model and a learning algorithm constructing this decision model.

Memory requirements vary as well since they depend on the application settings which perform a classification process. However, if the application

requires low memory consumption, Decision Trees may not be the first choice because of the replication problem: a portion of a subtree might be constructed multiple times [41]. This makes the Decision Tree more complex than required and, perhaps, requires much more memory to store it. This situation can arise due to "divide-and-conquer" methods applied by most learning algorithm, where the same attribute can be used for testing several times to different parts of the Decision Tree, cf. Figure 2.5.

Another well-known problem of Decision Trees when applying recursive learning algorithms is *data fragmentation* (or *model overfitting*): When reaching the leaves, the number of instances in the remaining dataset can be too small to make a statistically significant decision about the class label. We can say that a decision model overfits the training instances if some other decision model that fits training instances less well actually performs better over the decision tuples that are beyond the given training dataset [49]. This might later lead to a less accurate decision model. Classification usually requires the decision to be as precise as possible.

2.3 Points-to Analysis

Points-to analysis is a static program analysis that tries to predict the dynamic behavior of programs without running them. It computes reference information by approximating for each pointer in the program, e.g., variables, a set of possible objects to which it could point at runtime [26]. Points-to analysis results can be used as input to a number of client-applications, such as static garbage collection which automatically deallocates method objects once a method execution has been completed, metric analysis which computes coupling and cohesion between program objects, etc.

Each reference variable v in a program is associated with a set of objects $\{o_1, o_2, \dots, o_n\}$, denoted as $pt(v)$, that contains all the objects that variable v might point to. The set $pt(v)$ is called the *points-to set* of variable v , and the objects, called *abstract objects*, are an abstraction of the different objects that can be created at a runtime.

Let us consider the program example presented in Figure 2.6. Here, o^i means an object created at allocation site s^i . *Allocation site* s^i is a syntactic point in a program in which a particular object is created. The goal of points-to analysis for this example is to compute what possible abstract objects variables a , b , c , and d can point to during any execution. The points-to results usually depend on the execution paths a particular program can take. However, this program example has two possible execution paths, and, to be safe, most analyses take an over-approximated (conservative) approach, i.e., they take into account both paths. Therefore, the points-to set of variable c contains both abstract objects $\{o^1, o^2\}$ at line 6. Ideally, points-

```

1: s1: a = new A(); // pt(a) = {o1}
2: s2: b = new B(); // pt(b) = {o2}
3: if(...)
4:   c = a;          // pt(c) = {o1}
5: else{
6:   c = b;          // pt(c) = {o1, o2}
7: }
8: d = c;            // pt(d) = {o1, o2}

```

Figure 2.6: A source code example

to results should be precise, i.e., as small as possible, and complete, i.e., not missing anything. However, exact points-to analysis is intractable and, hence, approximations must be accepted.

Precision and Costs: A program may create countable many runtime objects and may have unknown input, so exact points-to information cannot be computed in general. In real life, where the program to be analyzed may contain hundreds or even thousands of classes, there is a trade-of between the analysis costs and precision. The costs of the analysis generally include time and memory consumptions.

Recent advances in points-to analysis research have concentrated on the improvement of the analysis precision without the increase of memory costs [22, 4]. In what follows we point out the main dimensions proposed by Ryder [69] that determine the relative precision and cost of analysis and that are relevant to our points-to analysis presentation:

- **Object representation.** A points-to analysis distinguishes objects, denoted as *abstract objects*, by their static abstractions. The mapping between abstract objects and run-time objects is called a *name schema*. The choice of a name schema effects analysis precision and memory requirements. Coarse-grained schemata, where one abstract object can represent all instantiations of one class, will lead to low memory consumption but less precise analysis. The opposite holds to more fine-grained schemata, where each abstract object corresponds to a much smaller set of runtime objects. For our example we have chosen a name schema that distinguishes objects by their syntactic creation points s^i .
- **Context-Sensitivity.** In a context-insensitive points-to analysis, analysis values of different calls are propagated to the same method and get mixed there. The analysis value is then a merger of all calls targeting that method. A context-sensitive analysis addresses this source

of impression by distinguishing between different calling contexts of a method. It analyzes a method separately for different calling contexts of a method.

- **Flow-Sensitivity.** An analyses is flow-sensitive if it takes into account the order in which the statements of a program are executed. This leads to a different analysis that respects all dataflow and control flow dependencies among the program statements. Our example in Figure 2.6 is flow-sensitive. In a flow-insensitive scenario we would simply have a set of assignments and all four variables a, b, c, d would point to the same set of objects $\{o^1, o^2\}$.
- **Path-Sensitivity.**¹ An analysis is path-sensitive if it takes the feasibility of different execution paths into account. Feasibility can be determined by evaluating the expressions in control flow statements or by computing points-to values with respect to the program paths where they occur. Path-sensitive analysis is more accurate because it is able to reason about branch correlations, which is needed to compute points-to set information for different paths. Our example is path-insensitive, since variable c in the else part of if-statement points to two abstract objects $\{o^1, o^2\}$, where o^1 was computed in the if-part of if-statement. In a path-sensitive analysis, c would, at line 6, only point to the abstract object o^2 .

Both context- and flow-sensitivity will therefore, in general, give a more precise analysis. Additionally, a higher level of precision can be reached by introducing path-sensitivity to points-to analysis. However, the costs of path-sensitivity do not scale for large programs. The reason for the high cost, is that an accurate tracking of every path in the control flow of a program, where points-to sets for different abstract objects differ along the branches, may result in an exponential search space [16].

Moreover, path-sensitive points-to analysis requires avoiding mixing of memory in parallel branches of the program. A program usually has a number of splitting points where the execution is taking *only one* of a number of different possible branches. This happens in control flow statements (e.g., *if-then-else*, loops, etc.) but also in polymorphic calls. Therefore, path-sensitivity is a source of a decision relation in points-to analysis. The decision relation corresponds to points-to information that might depend on what particular branch the program can take in a control flow option. Thus, a compact representation of control flow options in a suitable decision model

¹Path-sensitivity was not included as a dimension by Ryder, but we included it as a dimension because it can affect analysis precision and it generates the contexts for the point-to analysis decision information.

is necessary. In the following we describe a decision model that is suited for capturing decision relations in points-to analysis.

Decision Model in Points-to Analysis

In this section we introduce χ -terms, a means for capturing context-sensitive information in data-flow analysis, as a decision model that captures decision information in points-to analysis. Generally χ -terms correspond to a compact representation of how different control flow options can affect a given value. The introduction of χ -terms is based on the formalism presented by Martin Trapp in his dissertation [74].

In what follows we will give an introduction to χ -terms as a decision model for capturing points-to information. This introduction is based on a simple code example depicted in Figure 2.7 that shows how χ -terms can be applied to points-to analysis. Our code example shows a simple piece of code that contains four if-else statements, a corresponding control flow graph (in the middle) and a Static Single Assignment form (SSA) (on the right). A control flow graph is a graph representation of a program showing all the paths that might be traversed during execution of the program. SSA is an intermediate representation that is tailored for certain program analyses and compiler optimizations. The traditional approach to SSA presented in [15, 53] tries to present every variable as the target of only one assignment statement that makes def-use relations explicit since a use of variable may use the value produced by particular definition if and only if the definition and use have the same variable name in SSA of the procedure. For instance, the assignment $a = 3$ is the second assignment to variable a , thus, the unique number 2 in $a = 3$ identifies it as a new value.

Join points in a program (ϕ -nodes in control flow graph) occur when several control flows from different predecessors are joined. This kind of multiple assignments to a variable is handled by pseudo computations, so called ϕ -functions. They make sure that each use of variable refers to exactly one definition, e.g., for the assignment $y = a$ the value of a is unique and corresponds to a_5 .

Each ϕ -function has as many arguments as there are possible predecessors coming together at that point. Each join point (ϕ) in SSA representation has ϕ -functions which symbolize the control flow uncertainties at these points. For example, the value a depends on which branches of the first and the second if-statement are chosen. The assignments at these points (e.g., $a_5 = \phi(a_1, a_4)$) indicate that we have a new value that is either a_1 or a_4 . Even more, by relying on the position of the value (argument) in the ϕ -function, we can determine from which path this value came from.

The control flow graph representing our code example contains basic blocks that correspond to assignment statements. It also contains splitting

Figure 2.7: A source code example with corresponding control flow graph and SSA form

points (γ) that represent if-else statements and corresponding join points that are represented by ϕ -functions.

Notice that after the data-flow analysis in the path-insensitive graph interpretation, the multi-valued variable values resulting from a join point in the control flow graph contain no information about the different control flow options that generated all these values. It simply states what possible values they are (e.g., values of b can be 3 or 2). By representing these values with χ -functions, a more precise description of the values can be achieved. For example, we can write down the value of variable b in Figure 2.7 using χ -function as $b = \chi^6(a_2, a_3) = \chi^6(3, 2)$, where 6 is a χ -function index referring to the basic block number where the ϕ -function occurred. Interpretation: variable b has the value 3 if it was reached from the first predecessor of block 6 (*if* - condition) in the control flow graph, and the value 2 if it was reached from the second predecessor block (*else* - condition). That is, the value expressed using χ - function does not only contain all possible values, but it also contains control flow options that generated each of these values. Using this approach we can write down the χ -term values for the variables at the end of the program execution in Figure 2.7 as:

$$\begin{aligned} a &= \chi^7(4, \chi^6(3, 2)), \\ b &= \chi^6(3, 2), \\ y &= \chi^{10}(\chi^7(4, \chi^6(3, 2)), 6), \\ c &= \chi^{13}(\chi^7(4, \chi^6(3, 2)), \chi^{10}(\chi^7(4, \chi^6(3, 2)), 6)) \end{aligned}$$

Notice that χ -term values (e.g., for variables y and c) that depend on several control flow options are constructed as a composition of the χ -functions representing these different options.

χ -functions The set of χ -functions $X(p)$ in a concrete program p depends on the set of ϕ -functions $\Phi(p)$ described above. Furthermore, the exact number of χ -functions depends on what type of loop handling is used. Since the loops can be handled in different ways, e.g., by ignoring the loops completely or by introducing approximations, we will not take them into account when describing the notations and operations on χ -terms in this section. Even though it is not relevant for the current presentation, the interested reader can find information about it in [47].

Each χ -function χ^i is identified by a unique identifier i , where: $i = \text{block}(\chi^i)$ is a basic block number. This number indicates the basic block of the ϕ -node.

χ -terms Two χ -terms $\chi^b(x_1, \dots, x_n)$ and $\chi^b(y_1, \dots, y_n)$ represented by the same χ -function χ^b have the same *switching behavior*. It means that for every execution of the program, it holds that iff branch x_k is selected for the first term then branch y_k is selected for the second term.

to values. The tree representation of a χ -term is easy to understand and important from a theoretical point of view: many of the operations and notations to be presented are based on a tree representation. However, a χ -term t can also be represented as a directed acyclic graph (DAG) shown on the right side of Figure 2.8. This DAG is a redundancy-free information representation of the χ -term.

In order to get a clear understanding of the operations and properties of the χ -terms, we have to introduce several notations based on a tree-representation of a χ -term $t \in X_V$. Tree representation is denoted as T_t :

- The *leaves* of t are the set of all leaf values in T_t ,
- The *sub-terms* of t are the set of all subtrees of t in T_t . For example, in Figure 2.8, χ^7 has four subtrees χ^6 , $\chi^0 = 4$, $\chi^0 = 3$ and $\chi^0 = 3$.
- The *children* of t are the set of children of a node corresponding to t in the tree representation,
- The *depth* of t is the depth of a tree representation of t in T_t . For example, in Figure 2.8, the depths is four.

In this section we have shown how *if-else* control flow options influence the intermediate and final value of variables in a simple program and how we can represent these options by introducing χ -terms. In what follows we will map this example to the points-to analysis domain and show what difference in precision can be gained by using χ -terms as a representation of points-to values.

Path-sensitive Points-to Analysis Let us consider the following example on Figure 2.9. Each abstract object $o \in O$ has a unique set of object fields $f \in F$. Each object field f is represented by a points-to set value $pt(f)$. In this type of analysis we never overwrite a previous value of object fields in store operations, i.e., we never execute *strong* updates. Instead we perform *weak* updates.

In our example f is an object field. Every *new* statement on a line i generates a new abstract object $o_i \in O$ according to the chosen name schema. When a *store* operation occurs, i.e., $a = f$, a points-to set value $pt(a)$ is merged with the points-to set value $pt(f)$ already stored in an object field f , i.e., $pt(a) = pt(a) \sqcup pt(f)$. Obviously, there are two operations applied to object fields: *store* and *load*, where *store* writes a new value to the object field and *load* reads from it.

Since our points-to analysis is conservative and flow-sensitive, but not path-sensitive, we traverse each branch of the control flow separately one by one following the top-down order of statements. The control flow graph is presented on the right-hand side of Figure 2.9. At point P_1 , after the first

Figure 2.9: Code example and corresponding control flow graph

branch of the first splitting point γ_1 is traversed and the store operation is executed (line 4), we have $pt(f) = \{o^2, o^4\}$. At point P_2 , the points-to set of f is $pt(f) = \{o^2, o^4, o^7, o^9\}$. Hence, $pt(b)$ after the assignment statement $b = f$ is equal to $pt(f)$. However, in a real execution, the points-to set of b , at this point, will be $pt(b) = \{o^9\}$. The current approach does not compute precise points-to result. The problem is that the analysis does not keep branches apart. That means that the values obtained from traversing the previous branch influence values obtained from traversing other branches. Therefore, the points-to values assigned to the same variable in different branches get mixed.

Now we are able to increase the analysis precision by introducing path-sensitivity to the analysis. This improvement will separate the branches of control flow options by generating contexts for each splitting point and the corresponding branches. In what follows we will show how χ -terms can be used to represent points-to values and to enable path-sensitivity. Moreover, we will show how to perform *field writing* and *field reading* using a χ -term points-to set representation.

Although we add path-sensitivity to the points-to analysis by separating program contexts, the program has to be traversed in the same top-down way as before, i.e., each branch is traversed completely from a splitting point γ_i until the corresponding join point ϕ_i . The difference is in the abstract memory representation: instead of referencing the abstract objects, we will reference the χ -terms which do not only state what objects were stored, but

also in which context they were stored.

Let us informally define a context and a context stack that will be used to keep track on a current context of the point where the program is analyzed. A program has a number of splitting points γ_i , where one execution is taken out of a number of different possible branches B_i . Later on, these branches are joined in the corresponding join points ϕ_i . Every splitting point γ_i together with a particular branch b_j correspond to a *context* cx which is represented by a tuple (i, j) , where i is a basic block number of splitting point γ_i and j is a branch of this splitting point. For instance, in our example, a context cx at program point P_2 will be $cx = (1, 2)(2, 2)$, where $(1, 2)$ corresponds to the first splitting point γ_1 and its second branch, and $(2, 2)$ corresponds to the second splitting point γ_2 and its second branch. Thus, in order to keep track of the current context at a particular program point, we use a *context stack*. When a points-to analysis reaches a splitting point γ_i and starts to analyze a particular branch of this point j , a context cx is formed and pushed to the stack. Each join point ϕ_i pops a context from the context stack identifying the return to the enclosing context. Thus, in every program point P_j we are aware of the context the analysis has been performed in.

A new abstract object created in a context cx is represented by a χ -term t with $children(t)$ equal to the number of branches for the current splitting point. All the children except for the current branch are initialized with $\perp$ value. For example, the current value for $f = new\ A(2)$ on line 4 will be $pt(f) = \chi^1(o^4, \perp)$. In order to get a new points-to value of f we *merge* the current value with a previous points-to value, denoted as $prev(f)$, that was computed for f before. That is, $pt(f) = prev(f) \sqcup \chi^1(o^4, \perp) = o^2 \sqcup \chi^1(o^4, \perp) = \chi^1(\{o^2, o^4\}, o^2)$.

Now, let us consider the changes in the points-to analysis results for our particular example shown in Figure 2.9 due to χ -terms representation of points-to values.

Field Writing Field writing is a part of a χ -term construction based on a given points-to information. We refer to this construction as to *learning*. Learning of χ -terms is done by a decision capturing approach, since in this case the decision relation corresponding to points-to set information is complete and non-contradictive.

As it was mentioned before, every splitting point γ_i in a control flow graph corresponds to several possible definitions of a single object field f . Each store operation in the current context constructs a new χ -term $\chi^x(\perp, \dots, o^k, \dots, \perp)$, where o^k is a value for f in context cx . In order to update the points-to value of variable f we *merge* χ -term t_{prev} representing its previous value $prev(f)$ with χ -term t_x representing its new value $pt(f)$: $t_{prev} \sqcup t_x$.

Let us return to the example from Figure 2.9 and assume that, initially, $pt(f) = \perp$, $pt(b) = \perp$.

- $f = \text{new } A(1)$ (line 2) : Store in a non-branching context, i.e. $cx = (0, 0)$

$$pt(f) = \perp \sqcup \{o^2\} = \{o^2\};$$

- $f = \text{new } A(2)$ (line 4): A new context $cx = (1, 1)$ is pushed on the context stack.

$$pt(f) = prev(f) \sqcup \chi^1(o^4, \perp) = \chi^1(\{o^2, o^4\}, o^2);$$

Context $cx = (1, 1)$ is popped from the context stack.

- $f = \text{new } C(1)$ (line 7): A new context $cx = (1, 2), (2, 1)$ is pushed on the context stack.

$$pt(f) = prev(f) \sqcup \chi^1(\perp, \chi^2(o^7, \perp)) = \chi^1(\{o^2, o^4\}, \chi^2(\{o^2, o^7\}, o^2));$$

Context $cx = (2, 1)$ is popped from the context stack.

- $f = \text{new } B(3)$ (line 9): A new context $cx = (2, 2)$ is pushed to the context stack. Current context stack status is $[(1, 2), (2, 2)]$.

$$pt(f) = prev(f) \sqcup \chi^1(\perp, \chi^2(\perp, o^9)) = \chi^1(\{o^2, o^4\}, \chi^2(\{o^2, o^7\}, \{o^2, o^9\})).$$

The representation of χ -terms presented in points-to analysis is slightly different from the χ -terms representation presented in the decision model introduction. The difference is in the information that χ -terms keep. In the previous representation, χ -terms kept track of the creation point of values, i.e., by looking at a χ -term we could determine in which branch of which control flow option a valued was created. In the current representation, χ -terms keep track of field-write operations, i.e., from each χ -term we can determine where a particular points-to value was assigned to the variable. For instance, let us consider the example from Figure 2.9. Assume that f is a local variable on line 4. Thus, the points-to value of f at this point is $pt(f) = \{o^4\}$ instead of $pt(f) = \chi^1(\{o^2, o^4\}, o^2)$.

Field Reading Field reading corresponds to the extraction of points-to analysis information after the analysis was completed. Therefore, this process can be considered as *decide*, since we read the points-to reference information based on the current actual context of the analysis.

On line 10 in Figure 2.9 we assign variable f to variable b . At this point the program reads the points-to value from f and writes it to points-to value of b . The current value of f at point P_2 is

$$pt(f) = \chi^1(\{o^2, o^4\}, \chi^2(\{o^2, o^7\}, \{o^2, o^9\})),$$

and the current context stack status is $[(1, 2), (2, 2)]$. The value of an object field is read by *restricting* the current memory value to the current context

value cx . The restriction operator is defined as $\chi|_{(i,j)}$. The restriction is made for each pair (i, j) in the stack. Hence, the points-to set of b is:

$$pt(b) = \chi^1|_{(1,2)} = \chi^2(\{o^2, o^7\}, \{o^2, o^9\})|_{2,2} = \{o^2, o^9\},$$

Therefore, by comparing the points-to value of b with the path-insensitive value, we can easily see the precision gain.

Approximation However, the size of the χ -term representing points-to value of f grows larger as the analysis proceeds and more control flow options influence the value of f . The current presented approach to points-to analysis represents a fully context-sensitive analysis where the effect of every control flow option for every variable is kept in a χ -term. The problem of this approach is that it does not give any upper bound for the size of χ -terms. In order to limit the size of the ever-growing χ -terms we can *approximate* the fully context-sensitive analysis to only keep track on the last k control-flow options that might influence the value of a variable. This type of approximation can be applied during the χ -term construction, and therefore, limit already the resulting χ -terms during the construction process. This approach corresponds to the one presented in Section 2.1 where decision relation can be approximated during the learning process.

The k -approximation of χ -terms is easy to understand using the tree representation. We replace all χ -terms $t_{sub} = \chi_i(t_1, \dots, t_n)$ in subterms(t) that have $depth(t, t_{sub}) \geq k$ with $\cup(t_1, \dots, t_n)$, where $depth(t, t_{sub}) = depth(t) - depth(t_{sub})$ is the depth of a term t_{sub} in the tree representation of t . The process starts in the leaves and proceeds towards the root node. The result is a new χ -term $t^{(k)}$ with $depth(t^{(k)}) \leq k$. The approximation process is recursive, where each iteration decreases k . We define this process as k -approximate or k -approx. For example, let $k = 1$:

$$k - approx(\chi^1(\{o^2, o^4\}, \chi^2(\{o^2, o^7\}, \{o^2, o^9\}))) = \chi^1(\{o^2, o^4\}, \{o^2, o^7, o^9\}).$$

The approach to points-to analysis presented in this section is based on a simple code example in order to give the reader an idea about how χ -terms can be applied to represent points-to values and how this representation can improve points-to analysis precision by applying certain operations during field reading and field writing. In general, we can use this type of points-to set representation in flow-sensitive points-to analysis.

As mentioned during points-to analysis description, the points-to information can correspond to decision relation, where control flow options are actual contexts represented by a tuple (i, j) and different sets of abstract objects are decisions. That is, points-to analysis in some extent process decision information.

Functional Requirements

The required operation to process decision information from the points-to analysis domain captured in χ -terms are: *learn* - the process of constructing χ -terms based on points-to information; *decide* - to read points-to analysis values based on a constructed χ -term; *merge* - to construct a new χ -term by merging a previously constructed χ -term with a new one; *k-approximate* - to limit the size of a χ -term to depth k .

Non-functional Requirements

As mentioned before, different control flow options in a program lead to different analysis values of a given program point. An accurate capturing of this type of decision relations is essential in order to increase analysis precision. High memory and construction speed costs are usually unacceptable since the analysis has to be performed with a response time in seconds rather than minutes. However, the decision time is not that critical and usually depends on the client application requirements which takes the points-to information as input.

The decision model must be usually precise in terms of its construction and representation, i.e., each context in an analyzed program has to be captured. Additionally, it should compactly capture influence of different control flow options on points-to values for program variables. Hence, χ -terms as a decision model has a number of properties that are essential for its construction and representation:

- The value that a χ -term $t \in X_V$ represents *does not depend on the order of nodes* in its tree representation. Therefore, two χ -terms are called equivalent if they represent the same context-sensitive value.
- χ -term $t \in X_V$ is *not redundant* due to its possible graph representation: $\forall t_i, t_j \in \text{subterms}(t), t_i \neq t_j$. A χ -term is redundant if all its sub-terms represent the same χ -term value and can therefore be replaced with this value. That is, $\chi^b(t, \dots, t) = t$.

2.4 Context-Aware Composition

Context-dependent computation became an essential part of a wide range of application domains, where an application should behave according to the conditions arising during execution (e.g., mobile applications, web services, etc.). A technique that enables this type of computation is called *Context-Oriented Programming*, which is a programming approach that treats contexts explicitly and makes them accessible and manipulatable by software [27, 14]. In order to get the desired behavior of an application, a specific composition,

(a) Decision Table			(b) Dispatch Table		
$\vec{A}$		C	PS	PA	
PA	PS			N	Y
N	4	SS	4	SS	SS
Y	4	SS	16	QS	QS
N	16	QS	64	QS	QS
Y	16	QS	256	QS	QS
N	64	QS	1024	QS	MS
Y	64	QS	4096	QS	MS
N	256	QS			
Y	256	QS			
N	1024	QS			
Y	1024	MS			
N	4096	QS			
Y	4096	MS			

Table 2.1: Example of (a) a Decision Table and (b) a Dispatch Table

called *Context-Aware Composition*, has to be made. This composition takes into account the current context of the execution.

Context-aware composition separates the concerns of defining component variants and the decision in favor of any of these variants in an actual composition context. The former is done by component designers who develop variants that might be advantageous in certain actual composition contexts, and can be possibly supported by variant generators that generate, for instance, different schedules. The latter is fully automated. Additionally, context-aware composition can improve software systems considerably as it dynamically composes with the prospected best variant of alternative algorithms, data structures, resource allocations, etc., for each actual composition context. Consequently, a set of composition contexts and component variants can be considered as a decision relation.

Decision Model in Context-Aware Composition

In a learning phase, variants are tested in different actual contexts, and the champion variant of each context is captured in a generalized Dispatch Table. During the actual composition phase, referred to as *deciding*, the Dispatch Table is used to get the champion variant in the actual composition context. The learning and composition phases can be separated (offline learning) or interleaved (online learning) leading to self-adaptive systems.

A context can cover a wide range of concepts varying from domain-specific to technology-dependent attributes, and including properties that may be spatial or temporal, or even based on hardware or software (e.g., number of available processors, problem size, etc.), thus, the number of attributes in the Dispatch Table can be very large.

The Dispatch Table is a decision model that compactly represents a complete and non-contradictive decision relation in context-aware composition. The Dispatch Table can be interpreted as a Decision Table containing four blocks: attribute, value of the attribute, decision interpretation, and decision value. The advantages of a Decision Table consist of the compact and easy problem description and of the independence of the resulting decision from the order of attributes in the table. However, it can be time-consuming to create and maintain large Decision Tables, especially when the table itself can keep redundant information referred to as the duplication problem [72].

Let us consider the example of the Decision Table for context-aware composition problem presented in Table 2.1(a). In this example, the dispatch technique attempts to speed up sorting by selecting the best algorithm for the current context. The context $\vec{A}$ includes the problem size PS and the processor availability PA, where PS is a continuous integer and PA is a boolean encoding whether or not processors are available (No or Yes). The algorithms are: SS - selection sort, QS - Quick sort and MS - parallel merge sort. In an actual application, this Decision Table will be encoded as a Dispatch Table presented in Table 2.1(b), that is, the entries of the table will correspond to the decisions, and attributes will correspond to table dimensions containing entries corresponding to the values of the context attributes.

The learning of the decision function represented by a Dispatch Table is trivial. In each construction step, a new decision tuple is taken from a complete and non-contradictive decision relation and added as a table row to the existing rows of the Dispatch Table. This add operation is referred to as *merge*.

Functional Requirements

The required operation to process decision relations captured in Dispatch Tables from the context-aware composition domain are: *learn* - to construct a Dispatch Table; *decide* - to look-up in the table for a corresponding decision; *merge* - to merge table rows with new rows in a composition phase.

Non-functional Requirements

The use of the context-aware composition approach in applications usually implies that a decision based on the actual context has to be made at runtime. Therefore, the speed for decision making becomes critical. However, it can

be compensated by the right choice of algorithm, method, etc., that are the most appropriate for the current context and, thus, speed up the whole application.

Scalability of context-aware composition generally depends on memory and speed scalability of used decision models that, in turn, depend on the actual number of contexts and decision model they employ. Therefore, a decision model used for capturing decision information in context-aware composition has to be scalable. However, capturing the decision relation in a Dispatch Table mapping context attributes to variants can be memory consumptive compared to Decision Trees and χ -terms: the tables grow exponentially with the number of context attributes. Hence, capturing these tables for every composition point, i.e., all calls to a function f of a component with several implementation variants implies that the amount of required memory for execution of a certain application might be increased dramatically. To remedy this problem, other implementations of decision models can be used.

The speed for learning depends on what type of system is chosen. In case of offline learning, the speed is not critical and therefore can vary over time. On the opposite hand, when online learning for self-adaptive system is chosen, the learning time becomes a crucial requirement. The learning has to be performed fast at program runtime.

2.5 Related Work

Classification

There is a great variety of algorithms and data structures (most commonly modifications of Decision Trees and Decision Tables) for learning and capturing classification information. Generally, a modification of these data structures comes with a modification of a corresponding learning algorithm.

Several variants approach the so-called fragmentation problem, a result of replications as discussed in [55]. One suggested approach uses Decision Tree nodes switching on *combinations* of attributes. For instance, Lam and Lee [41] present a method for building classification models by using correlation analysis of attributes (identifying so-called functionally dependent attributes). Similar ideas are presented in [59, 33]. Vilalta et al. [75] investigate top-down Decision Tree construction and prove theoretically and empirically the significance of the fragmentation problem in the learning process. To overcome this problem, they choose the best out of a number of possible attribute orderings by assessing their results against all training examples, and thereby avoid a misclassification of examples for which only little support is found.

Friedman et al. [21] present lazy learning, an algorithm which tries to construct the best Decision Tree for a given decision domain by basically keeping the information of each training instance. However, this algorithm requires a lot of memory when using the classical Decision Tree structure.

Quinlan presents the idea of merging different Decision Trees [62] and proposes an extraction of proposition rules from an already generated decision model. One of the benefits of a such approach is the elimination of unused conditions replicated in different paths of the tree. Sets of decision rules for the same data domain are merged in order to increase accuracy of a classifier. However, the rules have to be extracted from Decision Trees and their merger has to be implemented somehow, e.g., in a Decision Tree again.

In addition, there are many approaches suggesting different modifications of Decision Trees and tables data structures; several of them can be found in [35, 24, 72, 30, 54].

Points-to Analysis

Here we present current research related to a points-to analysis. We mostly focus on work related to path-sensitive points-to analysis approaches and work that employs special data structures for capturing context-sensitive information in points-to analysis.

Lundberg and Löwe [46] present a flow-sensitive approach to points-to analysis. Their approach to flow-sensitivity is an SSA-based points-to analysis that simulates the actual execution of a program. As a result, an analysis is locally and globally flow-sensitive. In Section 2.3 we extend this flow-sensitive points-to analysis by introducing path-sensitivity which leads to a more precise analysis.

Gutzmann *et. al* [23] present an approach that retains control-flow information by inserting path-specific filter operations in the program dataflow. The filter operations correspond to different types of branching conditions in the control flow. During analysis, they are treated as ordinary program entities of a program representation: each operation is assigned a transfer function, which is updated iteratively until a fixed point is reached. The experiments showed an analysis precision improvement, which was assessed by client-analyses that use this points-to information as an input.

Many approaches deal with the meet over all paths (MOP) dataflow problem. For example, Bodík *et. al* [7] develops a dataflow analysis framework based on a program representation called Value Name Graph. This representation makes explicit value-flow patterns that are of interest to a given analysis task, i.e., it uses path-sensitive value numbering on the employed name space. The presented approach overcomes the problem of naming the values that flow between equivalent computations with different lexical names.

Since the number of paths is, in general, unbounded, approaches narrow down the set of paths, e.g., by finding correlations between branch conditions. The approach presented in [16] accurately models only those branches in a program for which the property-related behavior differs along the paths of the branch. Additionally, the authors have designed an algorithm that is accurate enough to verify the program with respect to the given property, without paying the potentially exponential cost of full path-sensitive analysis. Xie *et al.* [85] use path-sensitivity for array access checker. Their approach to path-sensitivity selects a set of execution paths and eliminates infeasible paths based on branching conditions.

Binary Decision Diagrams (BDDs) have become widely used as a memory-efficient method of representing relations in points-to analysis [44, 43, 70, 88]. Whaley and Lam [81] present a context-sensitive points-to analysis where context-sensitivity is reached by creating a clone of a method for every context of interest. The relations between the methods are stored and computed using BDDs. By numbering similar contexts contiguously, the BDDs handle the exponential blowup of contexts by exploiting their commonalities. That is, BDDs only store unique contexts. The authors also developed a system called *bddbldb* [80] that represents relations using BDDs.

A similar approach is presented by Lhoták’s and Hendren’s Paddle framework [4] that integrates BDD-based analysis into the Java optimization framework Soot. Paddle is implemented based on the Jedd language [44], an extension of Java for expressing program-analysis in terms of relations using BDDs to store and manipulate these relations. In Paddle, all of the interdependent analyses are implemented using BDDs. The authors showed that BDD-based points-to analysis scales in terms of space, and can be implemented using standard BDD packages.

Context-Aware Composition

Context-aware composition, i.e., the runtime context-dependent binding of a call to a matching callee defined by another component, is gaining importance as a knob for performance optimization, in particular since the stagnation of CPU clock rates puts an urgent need to exploit new sources for performance improvements. Andersson *et al.* compose and optimize special implementations of data structures and algorithms, considering matrix multiplication as a case study [2]. This work may be considered as a generalization of the dispatch mechanism in object-oriented languages, for instance in the sense of Context-Oriented Programming (COP) [76, 27, 14].

The optimization of libraries for specific functionality such as linear algebra or signal processing is a natural target for optimized composition. Because the domain and the code base are limited and statically known, computations can often be described in a restricted domain-specific language

from which variants can be generated and tuned automatically. Here, even high off-line tuning times are acceptable due to the high reuse of libraries. Well-known examples include the library generators ATLAS for basic linear algebra computations and FFTW and SPIRAL [51, 52] for transformations in signal processing.

More recently, optimized composition has been proposed as an optimization technique also in the more general context of component based systems, where the programmer is responsible for annotating components so that their composition can be optimized for performance. However, only few approaches consider recursive components with *deep composition*, and only few consider the co-optimization of the selection of implementation variants with other variation possibilities, such as the layout and data structure of operands or scheduling.

Li *et al.* [45] implement a library generator for sorting that uses dynamic tuning to adapt the library to the target machine at installation time. They use a number of machine parameters (such as cache size, the size of the input and the distribution of the input) as input to a machine learning algorithm. The machine learning algorithm, a problem-specific combination of two different classifiers, is trained to pick the best algorithm for any considered scenario.

STAPL [73] non-recursively applies dynamic algorithm selection for sorting and matrix computations. Three different learning algorithms are used: a Decision Tree learner based on Quinlan’s ID3 [64] algorithm with different pruning strategies, found to perform best in the experimental evaluation, a standard feed-forward neural network with back-propagation learning, and a Naive Bayes classifier found to be inferior in its classification accuracy.

Yu and Rauchwerger [86] take a similar approach for reductions. From measurements of the individual implementation variants they construct predictor functions in two steps. First, they select a small number of polynomial terms in the context and machine model parameters from a term pool to build a generic prediction function. Second, they calibrate the coefficients from the training data using general linear regression and Decision Tree learning. For each call, the calibrated runtime prediction functions are then evaluated and the decision is memorized so it can be reused if the same parameter configuration should occur again. This way, the overhead of the dispatch at each call is reduced.

Kessler and Löwe [37, 38] consider optimized composition at the level of annotated user-defined components (i.e., not limited to closed libraries) together with scheduling, resource allocation and other optimizations, which allows for simultaneous optimization. For the off-line search and optimization phase, the approach uses an interleaved dynamic programming algorithm to construct a component variant Dispatch Table (V-table) for each component-provided functionality f and a resource allocation and schedule Dispatch Table (S-table) for independent calls. The tables are constructed

simultaneously bottom-up for increasing problem sizes and resource assignments.

Olszewski and Voss [57] proposed a dynamic adaptive algorithm selection framework for divide-and-conquer sorting algorithms in a fork-join parallel setup. Their approach is divided into two phases. First, they use a dynamic programming algorithm to select the best sequential algorithm for different problem sizes. Then, they determine the threshold problem sizes for when to submit the subproblems to a shared work queue and to execute in parallel rather than to execute sequentially. For the construction of a classifier, they use the C4.5 algorithm to generate a Decision Tree.

PetaBricks [3] applies a similar approach where, in the offline search phase, the variant choice functions for recursive components are not computed by dynamic programming but by a genetic algorithm, essentially applying heuristic cuts to the optimization space. There appears to be no further compression of the variant selection function. Schedules and resource allocation are not co-optimized with variant selection but delegated to a run-time system with a work-stealing dynamic scheduler. Numerical accuracy is considered as an additional algorithmic property in composition.

The recent paper by Wernsing and Stitt [79] presents an approach that builds upon PetaBricks but targets multicore CPU systems with FPGA-based accelerators. Like PetaBricks, this approach relies on dynamic scheduling and the learning is done offline, and like the approach by Kessler and Löwe [37, 38], the learned execution plan is stored in table form.

2.6 Conclusions

In this chapter we have introduced the notion of *decision relation* composed by a set of actual contexts, a set of arbitrary decisions and dependencies between them. We have also introduced the notions of the *decision tuple* and the *decision model*. As an example of application domains that process decision relations we have presented classification, points-to analysis and context-aware composition. It was shown that the data processed by these domains are actually the decision relation, where each domain has a corresponding actual context and a set of final decisions.

The classification domain from the field of Data Mining naturally operates decision relations and a formal representation of decision relation is in some extend referred to as classification information. Therefore, discussed application domains, to some extend, perform the *classification* task. Moreover, the field of Software Engineering (e.g., including points-to analysis and context-aware composition domain) turns out to be a field where many tasks could be formulated as decision problems and approached in terms of capturing and manipulating decision relations.

Due to the differences in non-functional requirements for application domains and the differences in the type of decision to be made based on the given decision relation, it becomes quite hard to present one general implementation of a decision model suitable for each application domain. Different domains still use different decision models, e.g., Decision Trees in the classification domain, χ -terms in points-to analysis or Dispatch Tables in context-aware composition.

Evaluation of the decision model depends on many considerations: (i) a type of a final decision to be made, (ii) an accuracy of the decision, (iii) interpretability, and (iv) a tradeoff between decision model complexity and degree to fit data [87]. The consideration (iv) is tightly connected to consideration (i): the type of decision to be made may influence the tradeoff between complexity and data match. For instance, in case of points-to analysis, the required decision should be precise in terms of final result interpretation, and, therefore, the decision model should to a high degree fit the decision relation. Thus, in order to learn this type of decision model, a decision capturing along with decision approximation approaches are usually employed. On the opposite side, the classification domain often requires a certain level of interpolation over the decision information due to the incompleteness and inconsistency of decision relations, and the decision is mostly a class prediction which has some degree of imprecision. Therefore, in this case a decision interpolation is the most preferred choice. Context-aware composition, in its turn, is somewhere in between these two requirements: it depends on the task it solves and, thus, the precision of the required decision can vary. So the corresponding decision model can be learned by several approaches separately. Hence, the representation of the decision relation is often problem-specific and formalism-dependent.

The problems connected with differences in decision models are usually approached by developing a great variety of learning algorithms and data structures based on a given decision relation. All learning algorithms, cf. Section 2.2, must bias the following factors: (1) the size or complexity of the decision model, (2) the amount of decision tuples, and (3) the decision accuracy of the decision model. Generally, a modification of the decision model comes with a modification of a corresponding learning algorithm.

Because of the variety of solutions and modifications to decision models and because of the variety of application domains manipulating decision relations (each coming with different notations and tailored implementations), we consider it worthwhile to introduce a generalized function, denoted as *decision function*, that captures decision relations and can be represented by different types of decision models. We base our theoretical generalization on the general decision relation representation from Section 2.1 combined with a number of operations required in application domains: *learn* and *decide* used by all domains; *evert* used by classification domain to reorder

attributes in Decision Trees; *merge* used by classification domain in pruning processes, by the points-to analysis domain for constructing new points-to values, and by the context-aware composition domain for constructing Dispatch Tables; *approximate* used by classification referred to as *pruning* process, and by points-to analysis referred to as *k-approximate*.

Thus, due to the theoretical formalization, decision functions can be formally described by a framework that unifies the theory on decision relation as well as the main operations required in the decision process. This theoretical framework is denoted as *Decision Algebra*. This formalization should allow one to uniformly describe and evaluate existing decision models capturing decision tuples derived from an amount of application domains. Furthermore, it should provide a simplified approach of choosing the right decision model for particular problems without adding new operations which, in general, may have a negative impact on application memory requirements and performance.

Chapter 3

Decision Algebra based on Higher Order Functions

This chapter presents a unifying theoretical framework that describes both decision relations and the main operations required in the decision (classification) process. We define a *Decision Algebra* which represents complete and non-contradictive decision relations as higher order decision functions, thus formalizing the decision relations introduced in Chapter 2. The notion of decision functions and their term representation is described in Section 3.1, where we first define decision functions and then introduce their notation and alternative representations. Additionally, the equivalence and redundancy of decision functions are discussed.

Section 3.2 formally describes the algorithm for construction decision functions, denoted as *learning*. Section 3.3 explains how to determine a correct decision based on the captured decision relations, i.e., how to *decide* on an item with a certain actual context.

The Decision Algebra defines the operations on the decision relation with regard to operations required in the application domains presented in the previous chapter. Therefore, in Section 3.4, we define *merge* that can serve as the basis for pruning operations and simple learning approaches, *approximate* that corresponds to different pruning strategies, and *evert* that can serve as the basis for different learning algorithms in which selecting an appropriate attribute order is essential. Moreover, we generalize *merge* to an operation called *apply* that allows us to symbolically compute with decision relations in general.

In Section 3.5 we introduce a partial ordering of decision functions which formalizes the "more accurate" relation on the set of decision functions. Moreover, the partial ordering and the set of decision functions together define a lattice referred to as the *Decision Lattice* over the set of decisions. This leads to an axiom which is valid for the *approximate* operation and states that this operation defines decision functions which give less accurate but not wrong decisions. In Section 3.6 we define *apply* and *evert* as core operations of our Decision Algebra; other operations can be derived from them. Finally, Section 3.7 gives a summary of this chapter.

3.1 Decision Functions

In this section we first define decision functions. Then we introduce alternative representations of them. Finally, we discuss the equivalence of decision functions and redundancy in the decision relations they capture.

Decision Function Definition

Definition 7. A decision function df is a mapping of contexts (attributes) $A_1, \dots, A_n$ to a classification decision C . We denote with DF the set of all decision functions with the same signature $A_1 \times \dots \times A_n \rightarrow C$.

For now, we assume that $A_1, \dots, A_n$ and C are discrete (or categorical) domains; a (possibly necessary) discretization of continuous domains will be discussed in the next two chapters with regard to different decision models. The *arity* of a decision function $df : A_1 \times \dots \times A_n \rightarrow C$, denoted as $\text{arity}(df)$, is the number of attributes n . Where required, we annotate the arity n as an index of a decision function df^n .

Decision Function Notation A decision function $df : A_1 \times \dots \times A_n \rightarrow C$ over finite domains can be defined extensionally by all its tuples $(\vec{a}, c)$ with $\vec{a} \in A_1 \times \dots \times A_n$ and $c \in C$ and alternatively represented in a Decision Table, cf. Section 2.2. The tuples can be captured in a *table* with n columns, one for each attribute. The tuple $(\vec{a} = [a_1, \dots, a_n], c)$ corresponds to the table entry $[a_1, \dots, a_n, c]$. Such a Decision Table has $|A_1| \cdot \dots \cdot |A_n| \cdot |C|$ cells.

Alternatively, we can define decision functions as higher order functions where a 0-ary (constant) decision function $df^0 : \rightarrow C$ is the result of a 1-ary decision function $df^1 : A_1 \rightarrow (\rightarrow C)$ and so forth. This leads to a *decision term* representation.

A decision function

$$df^n : A_1 \rightarrow \dots \rightarrow A_n \rightarrow C$$

can be represented by a term

$$df^n = x^1(df_1^{n-1}, \dots, df_{|A_1|}^{n-1})$$

where the $|A_1|$ -ary selection operator x^1 is applied to the arguments of A_1 . There are $|A_1|$ result functions $df^n(a)$, one for each attribute value $a \in A_1$, which are $(n - 1)$ -ary decision functions:

$$df_{idx_1(a)}^{n-1} : A_2 \rightarrow \dots \rightarrow A_n \rightarrow C$$

with $idx_i(a)$ being a bijective index mapping of each attribute value $a \in A_i$ to a unique *Natural* number.

Figure 3.1: Tree and graph representations of $x^1(x^3(1, 2), x^2(x^3(1, 2), 2))$

If necessary for distinction, we index a selection operator x^i with the index of the attribute A_i it switches on. For example, a decision function df^3 with three attributes can be represented as a term:

$$df^3 = x^1(x^2(x^3(1, 2), x^3(1, 2)), x^2(x^3(1, 2), x^3(2, 2)))$$

Redundancy and Equivalence

Because of its term representation and non-functional requirements in different application domains, a decision function df ought to inherit properties from the χ -term representation discussed in Section 2.3.

Definition 8. An n -ary decision function df^n is *redundant* if all its sub-functions are equivalent, i.e., represent the same decision, and can therefore be replaced with this decision.

That is:

$$df^n = x(df^{n-1}, \dots, df^{n-1}) \equiv df^{n-1}$$

Obviously, a decision function df containing a redundant sub-function $df_r = x^r(df_j, \dots, df_j)$ can be rewritten without any loss of information:

$$df = x(\dots, x^r(df_j, \dots, df_j), \dots) \equiv df' = x(\dots, df_j, \dots)$$

Definition 9. The process of removing redundancy is called *redundancy elimination*. In a decision term representation, this corresponds to replacing a subtree rooted with df_r by any of its (equivalent) sub-terms.

Figure 3.2: Equivalent decision functions: $df \equiv df'$

Because our example decision function from above contains redundant sub-functions $df^1 = x^3(2, 2) \equiv 2$ and $df^2 = x^2(x^3(1, 2), x^3(1, 2)) \equiv x^3(1, 2)$, it holds:

$$\begin{aligned} df^3 &= x^1(x^2(x^3(1, 2), x^3(1, 2)), x^2(x^3(1, 2), x^3(2, 2))) \\ &\equiv x^1(x^3(1, 2), x^2(x^3(1, 2), 2)) \end{aligned}$$

A redundancy elimination process overcomes the problem of *replication* inside the decision trees, as discussed in Section 2.2.

A non-redundant decision function df^3 can be represented both by a tree and a graph (Figure 3.1).

Definition 10. We consider two decision functions df and df' be equivalent, denoted $df \equiv df'$, if they capture the same decisions for the same actual context $\vec{a}$ (disregarding permutations of attribute values).

Figure 3.2 depicts an example of a decision function df given in previous paragraph and an equivalent decision function df' . Although both of them capture the same decision relation, the order of attributes is different. The reordering is done by the *evert* operation that will be discussed in Section 3.4. However, they are still equivalent since the dependencies between attributes and decisions remain unchanged. Notice that the representation of decision function df is non-redundant.

3.2 Learning Decision Functions

In Section 2.1 we presented different approaches to construction of decision functions from a given decision relation, denoted as *learning*. In this section

we discuss a simple learning algorithm corresponding to capturing decision relations and a learning algorithm corresponding to decision interpolation. Notice that learning is not a part of Decision Algebra operations; it is a separate domain-specific algorithm that is used in order to construct decision functions from a given decision relation.

The example of the first approach is points-to analysis, cf. Section 2.3, where a complete and precise representation of decision relation is required. Classification can serve as an example of the second approach where the decision relation can be incomplete and/or contradictory and, therefore, should be interpolated in order to decide on tuples not seen before. Context-aware composition, in turn, can apply both approaches depending on the goal of the application it is used in. Usually, the decision in favor of any of the two approaches is ought to be made based on application requirements. The approximation of decision functions will be discussed later in this chapter.

Decision Capturing

A contradictory decision relation (contains different decisions for the same actual context $\vec{a}$) requires a generalization of the decision terms as introduced before. Let C be the co-domain of a decision function (a finite set of discrete decisions) and define $d(C) = \{(c, n) | c \in C, n \in \mathbb{N}\}$ (with $\mathbb{N}$ the Natural numbers) a discrete *distribution* over C , i.e., a total mapping of the elements of C to their frequencies (or some weights). We denote the set of all possible distributions over C with $D(C)$. While capturing the decision tuples, which can be seen as a simple learning process, we replace the 0-ary (constant) decision functions $df^0 : \rightarrow C$ with 0-ary (constant) decision distribution functions $df^0 : \rightarrow D(C)$. For each decision tuple $(\vec{a}, c)$ in the decision relation DR , we update $d(C)$ of the corresponding leaf(s) by incrementing the frequency of c in $d(C)$ by one. Therefore this approach can be formalized as:

$$\begin{aligned}
 + &= D(C) \times n \rightarrow D(C), \quad n \in \mathbb{N} \\
 d(C) + 1 &= \{(c, n + 1) | (c, n) \in d(C)\} \\
 df &: A_1 \times \dots \times A_n \rightarrow D(C) \\
 learn(df, (\vec{a}, c) \in DR) &: \begin{cases} df(\vec{a}) = d(C) + 1, & \text{if } df(\vec{a}) = d(C) \\ df(\vec{a}) = (c, 1), & \text{otherwise} \end{cases}
 \end{aligned}$$

Now, decision functions can even capture contradictory information.

Decision Interpolation

The decision relation DR may be *incomplete*, i.e., it does not contain a decision for all possible combinations of actual contexts $\vec{a}$. In this case, the construction of a decision function requires interpolation of the decision relation by

a special algorithm. As discussed in Section 2.1, the decision function can be constructed in accordance with different types of decision models: set of rules, mathematical formulas, probabilities, etc. Hence, a decision function, in general, can be described by different representations. The learning algorithm takes a set of decision tuples DR as input and produces a decision function $df \in DF$ that estimates the defined decision model as an output:

$$learn : DR \rightarrow DF$$

Basically, each algorithm generalizes over the specific decision tuples, hypothesizing a general decision function that covers these tuples and other cases beyond the given decision relation [49].

The next chapter presents algorithms for both learning strategies that construct decision functions from a given decision relation.

3.3 Deciding Using Decision Functions

The decision relation supports several types of decisions, cf. Section 2.1. Points-to analysis serves as a good example of decision represented by a set of *final values* computed by an analysis, while classification and context-aware composition represent *class or algorithm* decisions. For this Decision Algebra presentation we will stick to the second interpretation of decisions captured by decision functions.

Deciding means to come to a unique decision c for a given actual context $\vec{a}$ using a decision function df . To decide based on the decision function we usually select the *mode* element of the decision distribution, i.e., the most frequently occurring element in the decision distribution: let $df(\vec{a}) = d(C)$, then $decide(df(\vec{a})) = mode(d(C))$. This means we apply the *mode*-function to a decision function $df : A_1 \times \dots \times A_n \rightarrow D(C)$:

$$\begin{aligned} df & : A_1 \times \dots \times A_n \rightarrow D(C) \\ mode & : D(C) \rightarrow C \\ decide & : (A_1 \times \dots \times A_n \rightarrow D(C)) \times A_1 \times \dots \times A_n \rightarrow C \\ decide(df) & = mode \circ df \end{aligned}$$

This variant of deciding is valid for most inductive-based classifiers, e.g., decision trees, decision tables, rule-based classifiers, etc. Obviously, for these cases, we can pre-compute *decide* if the learning phase precedes and is not interleaved with the decision phase. Therefore, we apply the *mode*-function to each of the distributions learned for a *df*-function and specific actual contexts. This saves space and decision time but loses information as stored in the decision distributions.

3.4 Decision Function Operations

We now motivate and define the operations on decision functions: *approximate*, *merge*, *apply*, and *evert*.

Approximating and Merging Decision Functions

Pre-computing *decide* after learning saves representation space without sacrificing decision accuracy. Alternatively, space can be traded off against information accuracy if we *approximate* a decision by ignoring one attribute and *merge* the different decision functions for alternative values of that attribute.

For defining approximations and mergers of decision functions, we first define the *restriction* of an n -ary decision function df to the k -th value of the i -th attribute, denoted $df^n|_{i:k}$, as a new $(n - 1)$ -ary decision function where the i -th attribute is bound to the value $a \in A_i$ with index $k = idx_i(a)$. Let, for example, $df^3 = x^1(x^3(1, 2), x^2(x^3(1, 2), 2))$. Then

$$\begin{aligned} df^3|_{1:2} &= x^2(x^3(1, 2), 2) \\ df^3|_{3:1} &= x^1(1, x^2(1, 2)) \end{aligned}$$

Let us define the merger $\sqcup$ of two distributions $d(C)$ and $d'(C)$ as:

$$\begin{aligned} \sqcup &: D(C) \times D(C) \rightarrow D(C) \\ d(C) \sqcup d'(C) &= \{(c, \max(n, n')) | (c, n) \in d(C), (c, n') \in d'(C)\} \end{aligned}$$

Note that we treat distributions as fuzzy sets and we define the merger of distributions as (a standard definition of) the union of fuzzy sets. This will come in handy when proving correct approximations of decision functions in Theorem 3 in Section 3.5.

Now we can recursively define the merger $\sqcup$ of two decision functions:

$$\begin{aligned} df, df' &: A_1 \times \dots \times A_n \rightarrow D(C) \\ df \sqcup df' &= x^1(df|_{1:1} \sqcup df'|_{1:1}, \dots, df|_{1:k} \sqcup df'|_{1:k}) \end{aligned}$$

where $k = |A_1|$. Note that $df|_{1:i}$ and $df'|_{1:i}$ are $(n - 1)$ -ary functions. Hence, we reduce the $\sqcup$ -definition of decision functions eventually to $\sqcup$ on distributions (0-ary decision functions).

Now we are ready to approximate a decision function df by ignoring an attribute A_i :

$$\begin{aligned} approx(i) &: (A_1 \times \dots \times A_n \rightarrow D(C)) \rightarrow \\ &\quad (A_1 \times \dots \times A_{i-1} \times A_{i+1} \times \dots \times A_n \rightarrow D(C)) \\ approx(i, df) &= \bigsqcup_{a \in A_i} df|_{i:a} \end{aligned}$$

Figure 3.3: Approximation and k-approximation of $x^1(x^3(1,2), x^2(x^3(1,4), 2))$

This operation is a basic operation for the *k-approximate* operation on χ -terms (Section 2.3), which can be interpreted as an approximation of decision function df to a certain attribute index k . Obviously, *k-approximate* is a repeated *approximate* starting from the leaves and proceeding to a certain k -index of decision function df . This operation can be formalized as follows:

$$\begin{aligned}
 k\text{-approx}(k) : (A_1 \times \dots \times A_k \times \dots \times A_n \rightarrow D(C)) \rightarrow \\
 (A_1 \times \dots \times A_k \rightarrow D(C)) \\
 k\text{-approx}(k, df) = \text{approx}(i, df_i), \forall i > k
 \end{aligned}$$

Figure 3.3 shows an example of approximation and k-approximation of decision function $df^3 = x^1(x^3(1,2), x^2(x^3(1,4), 2))$.

Applying Functions to Decision Functions

Recall that we applied *mode* to the leaf distributions of a decision function and that we applied the $\sqcup$ of two distributions to the corresponding leaves of two decision functions. These examples are just useful special cases of applying general functions to decision functions. By introducing such a general operation we determine a convenient way for manipulating decision functions, and, thus, manipulating decision relations. We define such a

general *apply* of arbitrary k -ary functions f to k -tuples of decision functions:

$$\begin{aligned}
 f & : C_1 \times \dots \times C_k \rightarrow C \\
 \text{apply}(f, c_1 \dots c_k) & = f(c_1 \dots c_k) \\
 df_1, \dots, df_k & : A_1 \times \dots \times A_n \rightarrow C_1 \\
 \text{apply}(f, df_1 \dots df_k) & = x^1(\text{apply}(f, df_{1|1:1} \dots df_{k|1:1}), \dots, \\
 & \quad \text{apply}(f, df_{1|1:k} \dots df_{k|1:k}))
 \end{aligned}$$

where $k = |A_1|$. The operation $\text{apply}(f, df_1 \dots df_k)$ recursively applies f to the respective subtrees of the arguments and eventually evaluates it on the leaves; the result is a decision function over C . In order to not mix this type of function f with a decision function we refer to it as to operation or as to *applying an operator op*. The special cases discussed earlier can be redefined as:

$$\begin{aligned}
 \text{decide}(df) & = \text{apply}(\text{mode}, df) \\
 df_1 \sqcup df_2 & = \text{apply}(\sqcup, df_1, df_2)
 \end{aligned}$$

Also, we can formalize the construction of a new decision function from previous knowledge represented by existing decision functions df, df' as:

$$\begin{aligned}
 \boxplus & : D(C) \times D(C) \rightarrow D(C) \\
 d(C) \boxplus d'(C) & = \{(c, n + n') \mid (c, n) \in d(C), (c, n') \in d'(C)\} \\
 \text{add}(df, df') & = \text{apply}(\boxplus, df, df')
 \end{aligned}$$

Note that $\boxplus$ can be also used as a merger of two different distributions in *approximation*. This type of approximation will be later on used in our experiments.

This way we can even formalize the simple learning procedure introduced in Section 3.2: a neutral element $\perp \in C$ is a default class representing “don’t know”, and learning starts without any knowledge, i.e., with the initial decision function $df \equiv \perp$. Each tuple $(\vec{a}, c)$ in the dataset corresponds to a decision function:

$$\begin{aligned}
 d(C) \boxplus \perp & = d(C) \\
 df'(\vec{b}) & = \begin{cases} c & \text{if } \vec{b} = \vec{a} \\ \perp & \text{otherwise} \end{cases}
 \end{aligned}$$

For each df' corresponding to a tuple of the dataset, learning incrementally sets $df := \text{add}(df, df')$. A similar learning procedure was discussed in Section 2.4 with respect to χ -term construction from decision relations. However, instead of the *add* operation on the leaves, the merge $\sqcup$ operation was used.

Everting Decision Functions

Heuristics for saving space during learning also alter the order in which attributes occur in the decision functions. Such an *evert* operation is naturally defined as a generalization of the so-called *Shannon expansion* of OBDD [8] over a decision function df with the i th attribute. It produces a new decision function. We define:

$$\begin{aligned} \text{evert}(i) : (A_1 \times \dots \times A_n \rightarrow C) &\rightarrow \\ &(A_i \times A_1 \times \dots \times A_{i-1} \times A_{i+1} \times \dots \times A_n \rightarrow C) \\ \text{evert}(i, df) &= x^i(df|_{i:1}, df|_{i:2}, \dots, df|_{i:k}) \\ df &\equiv \text{evert}(i, df) \end{aligned}$$

where $k = |A_1|$. Notice, the *evert* operation changes the order of decision terms in a given decision function but does not change the captured decision relation. It is just a rewrite rule that can be used to reorder the attributes of a decision function, e.g., to make the representation more compact. For example, let $df^3 = x^1(x^3(1, 2), x^2(x^3(1, 2), 2))$. Then

$$\begin{aligned} df_1^3 = \text{evert}(3, df^3) &= x^3(x^1(1, x^2(1, 2)), x^1(2, x^2(2, 2))) = x^3(x^1(1, x^2(1, 2)), 2) \\ df_2^3 = \text{evert}(2, df_1^3) &= x^2(x^3(1, 2), x^3(x^1(1, 2), 2)) \\ df_3^3 = \text{evert}(1, df_2^3) &= df^3 \end{aligned}$$

The everted decision function df_1^3 was previously depicted in Figure 3.2 in Section 3.1.

3.5 Decision Lattice

In the previous section we discussed, among other operations, the *approximation* of decision functions. This section presents axioms which are valid for this operation and which show that *approximate* operation defines decision functions which give less accurate but not wrong decisions (based on the decision relation it has learned from).

Therefore, we will introduce a partial ordering $\sqsubseteq$ of decision functions, which formalizes the “more accurate” relation of a set of decision functions DF over the same attributes (domain) and classes (co-domain). Note that our notion of accuracy can only approximate semantical correctness of decisions. Semantical correctness and accuracy are the same only in the ideal (but usually unrealistic) case of complete and non-contradictive datasets for learning. The pairs $(DF, \sqsubseteq)$ happen to define a lattice, which we refer to as *Decision Lattice*.

The “More Accurate” Relations

Assume C is a set of classes and $\mathcal{P}^C$ is the co-domain of a decision function, i.e., the set of all subsets of C . Our decision function suggests a set of possible decisions.

Intuitively, if two decision functions df, df' suggest the same decision for all attribute values $\vec{a}$, i.e., if $df \equiv df'$, they are equally accurate; they are non-comparable in accuracy if there exist attribute values $\vec{a}$, such that $df(\vec{a}) \neq df'(\vec{a})$.

Definition 11. *Decision function df is more accurate than decision function df' denoted by $df \sqsubseteq df'$ iff, for all attribute values $\vec{a}$, df' always suggests a few more (at least the same) alternative classes for some attribute values than df . The least accurate decision function suggests C for all attributes.*

In general, the merged decision suggesting $C_1 \cup C_2$ with $C_1, C_2 \in \mathcal{P}^C$ is less accurate than any of the two decisions C_1, C_2 and the “more accurate” relation on the decisions is the subset relation on $\sqsubseteq$ on the co-domain of the decision functions. We considered C_1 to be a more accurate classifications than $C_1 \cup C_2$ ($C_1, C_2 \in \mathcal{P}^C$), because there is no support to decide in favor of any of the classes $c \in C_1$ and $c \in C_1 \cup C_2$, respectively, and the probability of classifying correctly is $1/|C_1|$ and $1/(|C_1 \cup C_2|)$, respectively (provided that the correct class is contained in C_1 at all). Since $|C_1| \leq |C_1 \cup C_2|$, the probability of a correct classification is larger when based on C_1 than on $C_1 \cup C_2$.

The “more accurate” relation over sets of decisions C allows formalizing our intuitive “more accurate” relation over sets of decision functions with same attributes and decisions C . Therefore, we interpret $\sqsubseteq$ as a function $\sqsubseteq: \mathcal{P}^C \times \mathcal{P}^C \rightarrow \{\text{true}, \text{false}\}$ and define:

$$\begin{aligned} df_1, df_2 & : A_1 \times \dots \times A_n \rightarrow \mathcal{P}^C \\ df_1 \sqsubseteq df_2 & \Leftrightarrow \text{apply}(\sqsubseteq, df_1, df_2) \equiv \text{true} \end{aligned}$$

Note that $\text{apply}(\sqsubseteq, df_1, df_2)$ is a decision function with classes $\{\text{true}, \text{false}\}$ in the leaves: it is true iff for the sets of classes $C_1, C_2 \in \mathcal{P}^C$ of the corresponding leaves in df_1, df_2 , it holds that $C_1 \subseteq C_2$. Then $df_1 \sqsubseteq df_2$ holds iff, after redundancy elimination, $\text{apply}(\sqsubseteq, df_1, df_2) \equiv \text{true}$, i.e., it decides true for all arguments.

For instance:

$$\begin{aligned}
 df_1 &= x^1(x^3(1, 2), x^2(x^3(1, 2), 2)) \\
 df_2 &= x^1(x^3(1, 2), x^2(\{1, 2\}, 2)) \\
 apply(\subseteq, df_1, df_2) &\equiv x^1(\subseteq(x^3(1, 2), x^3(1, 2)), \subseteq(x^2(x^3(1, 2), 2), x^2(\{1, 2\}, 2))) \\
 &\equiv x^1(true, x^2(\subseteq(x^3(1, 2), \{1, 2\}), \subseteq(2, 2))) \\
 &\equiv x^1(true, x^2(x^3(\subseteq(1, \{1, 2\}), \subseteq(2, \{1, 2\})), true)) \\
 &\equiv x^1(true, x^2(x^3(true, true), true)) = true \\
 df_1 \sqsubseteq df_2 &\equiv true
 \end{aligned}$$

We have introduced a partial ordering $\sqsubseteq$ on a set of decision functions DF over the same attributes (domain) and classes (co-domain). In Section 3.4 we showed that arbitrary function f can be *applied* to a set of decision functions DF . This also holds for the *merge* operator $\sqcup$ on the decisions. Partial ordering $\sqsubseteq$, the set of decision functions DF and the *merge* operator $\sqcup$ can be used to define a decision-induced lattice DL , referred to as *Decision Lattice* (Figure 3.4), over the elements in DF .

Theorem 1. Let $DL = \{DF, \sqcup, \top, \perp\}$ be a lattice for some set of decisions C , and let $\sqsubseteq: DF \times DF \rightarrow \{true, false\}$ be an operator defined as:

$$df_1 \sqsubseteq df_2 \Leftrightarrow df_1 \sqcup df_2 = df_2, \forall df_1, df_2 \in DF.$$

The top element $\top \in DF$ of the Decision Lattice is a 0-ary decision function over the complete domain $\vec{A}$ and the complete co-domain C , and the bottom element $\perp \in DF$ is a 0-ary decision function over the empty co-domain C . The introduced lattice DL has an infinite height if the number of decision functions is unbounded (which is sometimes the case if no approximations are introduced, e.g., in points-to analysis).

Approximating Decision Functions

The aim of this section is to show that we can always replace any decision function with the merger of its sub-functions. This will, in general, lead to less accurate decision functions which, in general, require less space to capture. We refer to this type of decision term manipulation as $\sqcup$ -approximation.

For purpose of argumentation, in Theorem 2, we define yet another version of $\sqcup$ on sets.

Theorem 2. A decision function df is more accurate than its $\sqcup$ -approximation $\sqcup df$, i.e., for any decision function

$$\begin{aligned}
 df &: A_1 \times \dots \times A_n \rightarrow \mathcal{P}^C \\
 df &= x(df_1, \dots, df_n),
 \end{aligned}$$

Figure 3.4: Decision Lattice over a set of decision functions

it holds that

$$df \sqsubseteq \sqcup df,$$

where

$$\begin{aligned} \sqcup df &= (df_1 \sqcup \dots \sqcup df_n) \\ df_1 \sqcup df_2 &= \text{apply}(\cup, df_1, df_2) \end{aligned}$$

Proof. Relying on the formalization of the "more accurate" relation, it is sufficient to verify that $df \sqcup (\sqcup df) \equiv \sqcup df$, since we have to prove that df is a subset of $\sqcup df$. This can be done in a few steps starting with the definition of $\sqcup df$, continuing with applying $\sqcup$ over x , and ending with redundancy elimination:

$$\begin{aligned} df \sqcup (\sqcup df) &\equiv df \sqcup (df_1 \sqcup \dots \sqcup df_n) \\ &\equiv x(df_1, \dots, df_n) \sqcup (df_1 \sqcup \dots \sqcup df_n) \\ &\equiv x(df_1 \sqcup (df_1 \sqcup \dots \sqcup df_n), \dots, \\ &\quad df_n \sqcup (df_1 \sqcup \dots \sqcup df_n)) \\ &\equiv x((df_1 \sqcup \dots \sqcup df_n), \dots, (df_1 \sqcup \dots \sqcup df_n)) \\ &\equiv df_1 \sqcup \dots \sqcup df_n \\ &\equiv \sqcup df \end{aligned}$$

□

Theorem 2 is restricted to decision functions which, for each input attribute vector, provide sets of classes as output. It states that we can approximate these kinds of decision functions by ignoring an attribute and merging the respective sub-functions.

This approach would be even more relevant during learning. However, the decision functions constructed during learning may provide distributions over classes instead of sets of classes as output. In what follows, we generalize Theorem 2 such that it captures even decision functions having class distributions as the decisions.

Therefore, we generalize $\mathcal{P}^C$, the set of subsets of C , to $D(C)$, the set of all distributions over classes C . We interpret a distribution $d(C) \in D(C)$ as a fuzzy set where $(c, w) \in d(C)$ denotes that c belongs with a degree of membership w to fuzzy set $d(C)$, corresponding to the frequency w of c in $d(C)$. The standard union operation on fuzzy sets is actually our merge operation $\sqcup$ on distributions, and the standard generalization $\sqsubseteq$ of the set inclusion $\subseteq$ to fuzzy sets $C_1, C_2 \in D(C)$ is

$$C_1 \sqsubseteq C_2 \Leftrightarrow \forall (c, w_1) \in C_1, (c, w_2) \in C_2 : w_1 \leq w_2.$$

We can generalize the “more accurate” relation of decision functions with distributions over classes C as co-domains and define:

$$\begin{aligned} \sqsubseteq & : D(C) \times D(C) \rightarrow \{\text{true}, \text{false}\} \\ df_1, df_2 & : A_1 \times \dots \times A_n \rightarrow D(C) \\ df_1 \sqsubseteq df_2 & \Leftrightarrow \text{apply}(\sqsubseteq, df_1, df_2) \equiv \text{true} \end{aligned}$$

Now we are ready to generalize Theorem 2 to decision functions as occurring during learning.

Theorem 3. *A decision function df is more accurate than its $\sqcup$ -approximation $\sqcup df$, i.e., for any decision function*

$$\begin{aligned} df & : A_1 \times \dots \times A_n \rightarrow D(C) \\ df & = x(df_1, \dots, df_n) \end{aligned}$$

it holds that

$$\begin{aligned} df & \sqsubseteq \sqcup df \text{ where} \\ \sqcup df & = (df_1 \sqcup \dots \sqcup df_n) \\ df_1 \sqcup df_2 & = \text{apply}(\sqcup, df_1, df_2) \end{aligned}$$

Proof. Since we have defined a “more accurate” relation on distributions $D(C)$, the proof is structurally identical to the one of Theorem 2 and is therefore omitted. $\square$

Theorems 3 allows us to approximate decision functions by trading accuracy for space.

3.6 Core Operations of the Decision Algebra

The operations of the Decision Algebra can be separated in *core* operations and *non-core* operations. The core operations are the operations which all other operations (non-core) can be derived from. They include *apply* and *evert*. Therefore, derived operations are: *merge*, *decide* and *approximate*. These three operations are based on *apply*: *merge* corresponds to applying the $\sqcup$ operation on decision functions; *decide* is equal to applying a *mode* operation on a decision function in order to get a final decision; *approximate* can be interpreted as applying *merge* on decision function sub-terms.

3.7 Summary

In this chapter we have defined a Decision Algebra, a theoretical framework for capturing and manipulating decision relations. The Decision Algebra defines a notion of decision functions as a formal representation of complete and non-contradictive decision relations that can be used in different types of decision models.

The Decision Algebra defines a set of operations for manipulating decision relations: *approximate*, *merge*, *evert*, and *decide* representing the basic operations in different application domains, as discussed in Chapter 2. *Approximation* of a decision function, obtained by merging sub-terms of an approximated term, results into a decision function capturing "less accurate" but not wrong decision. Additionally, by applying the *merge* operation on sub-terms during the *approximation* process, we avoid a certain issue of how to interpret the decision function if the root term is approximated while its sub-terms keep the decision. The reordering of attributes within a decision function uses the *evert* operation. The result of *everting* a decision function is an equivalent decision function which keeps the same decision relation but has a different order of terms. Thus, the *evert* operation removes the distinction between decision terms that occur near the root of the decision function and those that occur near the leaves. The *decide* operation selects the *mode* element of the decision distribution of the decision function for a given actual context in order to come to a unique decision.

Notice that *learn* is not part of the operations on decision functions. Each decision model representing decision functions is constructed by a separate learning algorithm. This algorithm, in addition to several different computations, usually requires a decision function constructor that, by default, has to be implemented by each decision model. *Learn* is not the only domain-specific algorithm. In the next chapter we will present algorithms for decision function manipulation, such as *prune* and *k-approximate*, that were briefly introduced in Chapter 2. We will show how these operations

can be implemented based on the Decision Algebra core operations, and, hence, can be shared by different decision models.

The uniform theory presented in this chapter can be applied to formalize classical approaches in classification addressing the typical classification problems of fragmentation, replication and model overfitting discussed in Section 2.2. In fact, many classical decision models based on distribution representation of the decisions used in classification (e.g., Decision Trees and tables) and variants thereof as exploited in special learning and classification algorithms can be understood as Decision Algebra instances by varying the implementations of the Decision Algebra core operations (*apply* and *evert*). Non-core operations (*decide*, *merge* and *approximate*) have default implementations in terms of the core operations of the Decision Algebra, however, a dedicated implementation of these non-core operations can be found directly in specific decision models.

Several existing decision models (Decision Trees, Dispatch Tables) with corresponding core operations, along with an explicit Decision Algebra implementation (Decision Graphs), will be discussed in Chapter 5. Chapter 6 will give an overview of another two decision models: Naive Bayes classifier and Support Vector Machines, first mentioned in Section 2.2, that have slightly different representations in comparison to the one presented in this chapter (not distribution-based models). Thereafter, in Chapter 7, we present the experimental results that validate the implementation of decision models as instances of the Decision Algebra.

Chapter 4

Algorithms using Decision Algebra

In this chapter we present four algorithms from the three application domains discussed in Chapter 2. Section 4.1 describes a simple learning algorithm based on the decision capturing approach. This algorithm is a basic algorithm for constructing a χ -term based decision model in points-to analysis and a Dispatch Table decision model in context-aware composition. Section 4.2 presents the widely used C4.5 algorithm [65] as an example for learning algorithm interpolating decision relations. This learning algorithm is often used in the classification domain to construct tree-based decision models, e.g., Decision Trees or Decision Graphs. Sections 4.3 and 4.4 present two alternative approaches for approximating decision functions captured in decision models: one originates from points-to analysis (*k-approximation*), and the second is taken from classification (*pruning*).

The main goal of this chapter is to show that the discussed algorithms used to manipulate decision relations in different application domains can be represented based on Decision Algebra operations which allows implementing them without reference to a specific decision model implementation.

4.1 Decision Capturing

Learning a decision function represented by a Dispatch Table in context-aware composition and χ -terms in points-to analysis is a process of constructing a decision function by decision capturing as discussed in Section 3.2. In order to generalize a decision function over a set of decision tuples, a simple *learning* strategy as described in Section 3.4 ought to be applied. Using this learning approach, we expect decision relations to be complete and non-contradictive, cf. Section 2.1. The algorithm for constructing a decision function $df : A_1 \times \dots \times A_n \rightarrow D(C)$ from a given decision relation $DR : ((\vec{a}_1, c_1), \dots, (\vec{a}_n, c_n))$, where each vector $\vec{a}_i \in \vec{A}$ corresponds to an actual context leading to a decision $c_i \in C$, can be briefly sketched in Algorithm 1: For each decision tuple $(\vec{a}_i, c_i) \in DR$, we start by constructing an intermediate

Algorithm 1 Capturing decision relation

```

1:  $df = \perp$ 
2: for each  $(\vec{a}_i, c_i) \in DR$  do
3:    $df'(\vec{b}) = \begin{cases} c_i & \text{if } \vec{b} = \vec{a} \\ \perp & \text{otherwise} \end{cases}$ 
4:    $df = \text{apply}(\sqcup, df, df')$ 
5: end for
6: return  $df$ 

```

decision function $df'(\vec{b})$, cf. line 3. The intermediate decision function has a concrete decision $c_i \in C$ only for the actual context $\vec{b}$ it represents, and, therefore, it uses a neutral element $\perp \in C$ as a default class representation for other unknown contexts. In order to construct a decision function df , an intermediate decision function df' is merged $\sqcup$ with a decision function df , cf. line 4. The merge operation $\sqcup$ depends on application domain a decision function has to be used, e.g., it corresponds to the union operation in the points-to analysis domain or the $\boxplus$ in the classification domain.

4.2 Decision Interpolation

In the best case, the decision function could capture a decision relation DR without any loss of accuracy, as was shown in Section 4.1. However, in classification this is not the case since the decision relation is rarely complete and non-contradictive. In classification, *learning* algorithms compute decision interpolation in order to avoid that the decision function overfits the decision relation and is not able to classify new incoming tuples. Most of these algorithms employ a recursive top-down approach that partitions the input relation DR , along with the corresponding decision function structure. Each step divides the input relation into several regions by attribute value $v \in A_i$, depending on whether $v < \theta_i$ or $v > \theta_i$, where θ_i is a calculated parameter. This creates several sub-sets, each of which can be then subdivided independently. Depending on the type of attribute A_i there exist two most common scenarios of expanding the decision function [25]:

- A_i is a discrete attribute: the outcomes of the intermediate decision function df_v correspond directly to the known values of A_i . That is, a branch $df_{idx(v)}$ is created for each known value $v \in A_i$, and, therefore, the number of children of df_v is equal to the number of attribute values.
- A_i is a continuous attribute: df_v has two possible outcomes, corresponding to conditions $v \leq \theta_i$ and $v > \theta_i$, respectively, where θ_i is a

splitting point returned by a best attribute selection method as a part of the splitting criterion.

In order to select the most relevant partitioning, the learning algorithm changes the order of the attributes for each input decision relation so that the most relevant attributes are selected first, and possibly the less relevant [50] ignored. As an example of such a recursive algorithm that learns a decision function df from a decision relation DR , we take commonly used C4.5 algorithm [65], which is outlined in the Algorithm 2. The algorithm works by

Algorithm 2 Learning decision function

```

1: compute distribution of classes  $D(C)$  in  $DR$ ;
2: compute error of just selecting the mode of  $D(C)$  as the decision;
3: if error acceptable then
4:   return  $df: \rightarrow D(C)$ 
5: end if
6: for each  $A_i \in \vec{A}$  do
7:   compute  $gain(A_i)$ 
8: end for
9: choose the attribute  $A_i$  with  $max\ gain(A_i)$ 
10: if  $gain$  not acceptable then
11:   return  $df: \rightarrow D(C)$ 
12: end if
13: for each  $v \in A_i$  do
14:   let  $DR_v \leftarrow \{(\vec{a}, c) \in DR | A_i = v\}$ 
15:   recursively construct  $df_v$  for a decision relation  $DR_v$ 
16: end for
17: return  $df$ , where  $\forall v \in A_i, df(v) = df_v$ 

```

recursively selecting the best attribute split A_i to the relation (lines 6-9) and expanding the terms of the decision function (lines 13-16) until the stopping criteria are met (lines 3, 10). Every decision function is created by a decision function constructor (lines 4, 11, 17) that has to be implemented by decision models. Each decision function is created when each corresponding DR_v is processed (line 15).

4.3 k-approximation

In this section we present an algorithm for *k-approximation* that was formally introduced in Section 3.4. This algorithm is used to approximate points-to values while the points-to analysis is performed, cf. Section 2.3. This method is based on the merge operation $\sqcup$ corresponding to the union operation in

the points-to analysis domain or additive merge operation $\boxplus$ in the classification domain. The k -approximation method is presented by Algorithm 3, where DF is a set of decision functions. This method uses the *collapse* method presented in Algorithm 4.

Algorithm 3 $k\text{-approx}(k \in \mathbb{N}, x \in DF) \rightarrow DF$

```

1: if  $k = 0$  then
2:   return new collapse( $x$ )
3: end if
4: for each  $x^i \in \text{children}(x)$  do
5:   recursively construct  $x^*$ , where  $x_i^* = k\text{-approx}(k - 1, x^i)$ 
6: end for
7: return  $df$ , where  $df = \text{new } x(x_1^*, \dots, x_{|\text{children}(df)|}^*)$ 

```

Algorithm 4 $\text{collapse}(x \in DF) \rightarrow C$

```

1: if  $x \in C$  then
2:   return  $x$ 
3: end if
4:  $v \leftarrow \perp$ 
5: for each  $x' \in \text{children}(x)$  do
6:    $v \leftarrow \text{apply}(\sqcup, v, \text{collapse}(x'))$ 
7: end for
8: return new  $v$ 

```

The k -approximation of decision functions can be understood based on the tree representation: all subtrees of depth $\geq k$ from the root are replaced with the merger $\sqcup$ of their leaves, cf. line 6 in Algorithm 4. The result is a new decision function with arity $\leq k$, cf. line 7 in Algorithm 3. Algorithm 3 finds the subtrees at depth k , Algorithm 4 then merges their respective leaves.

4.4 Pruning

To save space or to avoid overfitting in Decision Trees, pruning is applied. Different pruning variants can easily be implemented using the *approximation* operation defined in Sections 3.4, 3.5, allowing to ignore actually any attribute or level of a decision function.

Learning algorithms for decision functions represented by Decision Trees might apply two different basic strategies: pre-pruning and post-pruning as discussed in Section 2.2. A pre-pruning strategy stops growing the tree when it reaches a node that yields the accuracy above a pre-defined threshold θ .

On the other hand, post-pruning in the first step creates a complete tree, and then prunes unnecessary subtrees by keeping the number of attributes that influence the decision the most and ignoring other attributes by merging the decisions. A *post-pruning* operation on Decision Trees can be considered as an alternative implementation of the *approximate* operation on decision functions. Therefore, in this section we present only *post-pruning* strategies.

Two rather different approaches are used for post-pruning: subtree replacement and subtree raising [83]. The subtree replacement operation corresponds to the *k-approximation* operation on decision functions and is sketched in the Algorithm 5. The operation proceeds from the leaves to-

Algorithm 5 $\text{tree-replacement}(n \in DF) \rightarrow DF$

```

1: if  $\text{error}(n, DR) \geq \theta$  then
2:    $n \leftarrow k\text{-approx}(0, n)$ ;
3: else
4:   for each  $r \in \text{children}(n)$  do
5:      $\text{tree-replacement}(r)$ 
6:   end for
7: end if
8: return  $n$ 

```

wards the root (lines 5–7) choosing the subtree to be removed which will lead to increasing the decision accuracy for the verification decision relation DR , or, in other words, which subtree has an estimated accuracy error higher than the expected defined value θ , cf. line 1. The algorithm selects particular subtrees and replaces them with single leaves assigned the most common classification corresponding to the highest class distribution on the leaves if these subtrees, cf. line 2. This operation is performed by *k-approx* on a node. Notice that, *k-approximation* operation used in this algorithm is slightly different from the one presented in Section 3.4. The difference is that the operation presented here chops a certain depth of the subtree to the $k = 0$ level. The subtrees are removed only if the resulting pruned tree performs not worse than the original one over the validation decision relation DR . The second pruning operation, the subtree raising, corresponds to the *approximate* operation, and is more complex. The idea of this strategy, sketched in Algorithm 6, is to remove a certain node n from the tree by raising one of its subtrees. This strategy is used by the C4.5 learning algorithm, presented in Section 4.2, after a complete learning. The actual C4.5 implementation is restricted to raising the subtree of the most popular branch. That is, it replaces a node n by the branch rooted by a node r outgoing from n that has more training examples than other branches of this node, cf. lines 2, 3.

Algorithm 6 raising($n \in DF$) $\rightarrow DF$

```

1: if error( $n, DR$ )  $\geq \theta$  then
2:   find  $r \in children(n)$  with maximum  $d(C)$  on the leaves
3:    $n \leftarrow r$ 
4: else
5:   for each  $n' \in children(n)$  do
6:     tree-replacement( $n'$ )
7:   end for
8: end if
9: return  $n$ 

```

Additionally, this pruning algorithm can be adjusted by raising a determined merge $\sqcup$ of the subtrees of a pruned node n , instead of selecting the most popular branch, cf. line 2 in the Algorithm 7. This is a slightly dif-

Algorithm 7 raising($n \in DF$) $\rightarrow DF$

```

1: if error( $n, DR$ )  $\geq \theta$  then
2:    $n \leftarrow apply(\sqcup, children(n))$ 
3: else
4:   for each  $n' \in children(n)$  do
5:     tree-replacement( $n'$ )
6:   end for
7: end if
8: return  $n$ 

```

ferent approach to pruning since the pruned decision tree does not lose any decision information kept in the primary tree.

4.5 Conclusions

In this chapter we have shown that algorithms for manipulation or construction of decision functions can be implemented using the core operations of Decision Algebra. The particular algorithms presented here are based on a constructor for a decision function and the *apply* operation.

The Decision Algebra core operations can be implemented differently by decision models. For instance, depending on the constructor implementation, the decision functions produced by different algorithms can have different representations, e.g., they can be represented by Decision Trees, χ -terms or Dispatch Tables. Moreover, the result of applying $\sqcup$ on a decision function also depends on the implementation of the $\sqcup$ operation in different decision models. Non-core operations, such as *merge*, *approx* and *decide*, can

be implemented in Decision Algebra platform and can be reused by different Decision Algebra implementations.

The implementation of the algorithms using Decision Algebra operations instead of binding them to the specific data structures implementing decision models gives the advantage of exchanging these algorithms between different decision models, and, therefore, between different application domains. For instance, *k-approximate* can be reused by Decision Trees or Dispatch Tables, the learning strategies can be applied to construct different decision models, etc. The only requirement to enable the algorithm exchange is to provide an implementation of the core operations in each decision model.

In the next chapter we present three decision models: Decision Trees, Decision Graphs - an implementation variant of χ -terms and the implicit implementation of the Decision Algebra, and Dispatch Tables, and show how core operations can be implemented in each of these decision models.

Chapter 5

Decision Models: Possible Implementations

In Chapter 3 we presented a theoretical framework, denoted as *Decision Algebra*, for capturing and manipulating decision relations. We formally described the core operations (*apply*, *evert*) over decision functions from which all other required operations (*merge*, *decide*, *approximate*) can be derived. In the previous chapter, we showed how existing algorithms originating from different application domains can be implemented using the Decision Algebra's core operations.

Depending on the decision procedure and problem to solve, decision models differ in speed of learning and deciding, memory consumption, and decision accuracy. The decision model used to represent a decision function is usually chosen based on the problem domain, sometimes even on the sample data [58, 55]. In this chapter we introduce a few existing decision models from the field of Data Mining and Machine Learning as possible implementations of the Decision Algebra. These decision models are: Decision Trees, used in classification; Decision Graphs, an implicit implementation of the Decision Algebra and an implementation variant of χ -terms used in points-to analysis; and Dispatch Tables, used in context-aware composition. Each decision model is an instantiation of Decision Algebra and, therefore, has to implement the Decision Algebra core operations. Additionally, bounds on non-functional properties, such as memory and decision overheads, can be estimated theoretically for every decision model regardless of the specific problem domain it might be used in. However, model accuracy is a subject to experiments, since it is based on the particular decision relation.

Therefore, this chapter presents each decision model along with its implementations of the core operations and non-functional properties estimation. Accordingly, the structure of this chapter is as follows: Section 5.1 presents Decision Trees that encode context attributes in inner nodes. Section 5.2 introduces Decision Graphs which avoid redundancy within the representation of decision functions. Section 5.3 presents Dispatch Tables as a baseline decision function implementation. Finally, Section 5.4 concludes this chapter.

5.1 Decision Trees

Decision Trees were introduced as a decision model in classification the domain in Section 2.2. Decision Trees are one of the widely used and practical decision models for inductive inference [49]. The corresponding data structure implementing this model represents trees that encode context attributes in inner nodes. Each outgoing edge of such a node refers to a value (or value range) of the context attribute, and each path from the root node to a leaf represents actual context values leading to a classification result.

Let $df : A_1 \times \dots \times A_n \rightarrow C$ be an n -ary decision function. Every decision term representation of df can be seen as a tree $G = (N, E, r)$. The root node $r \in N$ corresponds to the selection operator x^1 of attribute A_1 , returning the child $idx_1(a)$ for a given argument $a \in A_1$. The child is a sub-tree representing the $(n-1)$ -ary decision function $df(a)$. 0-ary decision functions df^0 are leaves labeled with elements of C .

Accordingly, the standard operations on the Decision Tree data structure, which are required to implement the set of core-operations, are: *children(x)* returns sub-trees of a tree node x ; *getChildAt(x, i)* returns the i -th child of a node x ; $|children(x)|$ returns the number of sub-trees of a node x .

The construction of a Decision Tree is done by the learning algorithm presented in Section 4.2 that interpolates decision relation. The only difference is that every recursively constructed df_v is immediately added to a node as a child. This difference occurs due to the specifics of the implementation of a tree node constructor $x^j(x)$ that constructs a new tree node corresponding to selection operator x^j . As an argument x , it takes either a decision distribution $d(C)$ and generates a leaf node, or an integer number n and generates an inner node with empty slots for n children.

Apply

As discussed in Section 3.4, the higher order function *apply*, cf. Algorithm 8 below, pushes an operator op to the leaves and then applies it on them. Hence, the operations must be defined on the leaf values C . The result of *apply* on two Decision Tree nodes n_1 and n_2 , with root nodes corresponding to attributes A_k and A_j , respectively, is a Decision Tree recursively constructed from the results of the operation applied to the leaves. To push the operator op to the leaves of the tree, an order of all nodes ought to be maintained: if attribute A_i is chosen for splitting by a learning algorithm before attribute A_j , then $i < j$ and $x^i < x^j$, and selection operator x^i ought to occur before operator x^j on all paths from the root to the leaves in the corresponding Decision Tree. Lines 1–3 handle the base case where tree nodes are leaves and the operation is applicable directly. Lines 4–9 and, similarly, lines 10–15 handle the case when the selection operator of the first term is smaller than

Algorithm 8 $\text{apply}(op \in C \times C \rightarrow C, n_1 = x^k, n_2 = x^j) \rightarrow x^z$

```

1: if  $x^k, x^j \in C$  then
2:   return  $op(x^k, x^j)$ 
3: end if
4: if  $k < j$  then
5:   for each  $x' \in \text{children}(x^k)$  do
6:      $x^* = \text{apply}(op, x^j, x')$ 
7:   end for
8:   return  $x^k(x^*_1, \dots, x^*_{|\text{children}(x^k)|})$ 
9: end if
10: if  $j < k$  then
11:   for each  $x' \in \text{children}(x^j)$  do
12:      $x^* = \text{apply}(op, x^k, x')$ 
13:   end for
14:   return  $x^j(x^*_1, \dots, x^*_{|\text{children}(x^j)|})$ 
15: end if
16: for each  $i \in [1 \dots |\text{children}(x^k)|]$  do
17:    $x^* = \text{apply}(op, \text{getChildAt}(x^k, i), \text{getChildAt}(x^j, i))$ 
18: end for
19: return  $x^j(x^*_1, \dots, x^*_{|\text{children}(x^k)|})$ 

```

the second or if it is larger than the second. Lines 16–19 handle the case where the selection operators are identical. In all three cases a tree node representing a new decision function is created, cf. lines 8, 14, 19.

Evert

An implementation of the *evert* operation on Decision Trees is similar to the implementation of Shannon expansion on Ordered Binary Decision Diagrams [8]. This operation was introduced as Decision Algebra's core operation in Section 3.4. Given an attribute A_j and a root node r as an input, the operation reorders nodes in the tree rooted by node r corresponding to attribute r in such a way that attribute A_j represented by a node j becomes a new root, cf. Algorithm 9. Notice that the everted Decision Tree has the same dependencies over attributes represented by nodes and keeps the same decision relation as before.

The *evert* operation recursively *restricts* a current root node n to the i -th branch ranging from one to m , where m corresponds to a number of values the attribute A_j has, cf. lines 4–6. A restriction operation has its counterpart in the OBDD description of a Boolean formula $f(x_1, \dots, x_n)$ that generates a new formula where the i -th variable x_i has been assigned the constant

value *true* [9]. A version of the restriction operation for Decision Trees, cf. Algorithm 10, generates a new decision node $n_{(z):i}$ by only considering the value from the i -th successor of node z corresponding to the attribute A_j that is in a subtree rooted by n . The algorithm recursively traverses subtrees of node n , cf. lines 4–10, until it finds a node j that has to be restricted to the i -th child, cf. lines 5–7.

Algorithm 9 $\text{evert}(r = x^r, A_j) \rightarrow x^j$

```

1: if  $x^r \in C \parallel r = j$  then
2:   return  $x^r$ 
3: end if
4: for each  $a_i \in A_j$  do
5:    $x_i^* = \text{restrict}(x^r, i)$ 
6: end for
7: return  $x^j(x_1^*, \dots, x_{|A_j|}^*)$ 

```

Algorithm 10 $\text{restrict}(x^n, i \in N) \rightarrow x^y$

```

1: if  $x^n \in C$  then
2:   return  $x^n$ 
3: end if
4: for each  $x^z \in \text{children}(x^n)$  do
5:   if  $z = j$  then
6:      $x_m^* = \text{restrict}(\text{getChildAt}(x^z, i), i)$ 
7:   else
8:      $x_m^* = \text{restrict}(x^z, i)$ 
9:   end if
10: end for
11: return  $x^n(x_1^*, \dots, x_{|\text{children}(x^n)|}^*)$ 

```

In what follows we will present non-functional properties that can be estimated for the Decision Tree data structure implementation.

Non-Functional Properties

In the worst case scenario, when the decision relation captured in the Decision Tree is complete and non-contradictive, the memory required for capturing the Decision Tree is quite large and grows exponentially with the number of used attributes n . The decision relation in this case has $k = |A_1| \times \dots \times |A_n|$ decision tuples, and the corresponding Decision Tree has $L = k$ leaves and (almost) k inner nodes. Thus, the number of *edges* is $O(L) = O(|A_1| \times \dots \times |A_n|)$. This size is reduced when the learning strategy interpolates the decisions

and therefore the number of edges becomes smaller. It can also be reduced if all paths from an inner node lead to the same decision (making this whole subtree redundant). Hence, the memory consumption is

$$M = \text{size} \times \text{edges},$$

where *edges* is the number of edges in the tree and *size* Bytes are necessary and sufficient to encode all different nodes of the tree.

Using a Decision Tree as a dispatch repository for a given context $\vec{a} = (a_1, \dots, a_n)$ is straightforward: in each inner node corresponding to attribute A_i , starting at the root node, the attribute value $a_i \in A_i$ is used to decide which child to visit next. The number of tests necessary to reach a leaf is equal to the *depth* of the Decision Tree. This *depth* varies around the number n of context attributes: for *discrete* context attributes it is at most n ; *continuous* attributes can even occur several times on the path due to data partitioning [66]. So, generally, the prediction time is $\text{depth} \times T_{aa}$ and we approximate

$$T(n) \approx n \times T_{aa} + c.$$

where T_{aa} is the array access time, and c is a constant time used for small operations.

5.2 Decision Graphs

The main drawback of Decision Trees is that they grow exponentially with the number of context attributes, and even very small decision problems require relatively large Decision Trees [31]. The main concept of Decision Graphs is to represent Decision Trees in a compact way by eliminating redundant subtrees.

Our Decision Graph implementation is an implicit Decision Algebra implementation and is partly based on Ordered Binary Decision Diagrams (OBDDs) [9, 8] and partly on χ -terms data structures used in the points-to analysis (Section 2.3). In contrast to OBDDs, we do not have binary but general decisions, since a context attribute could have any domain. Additionally, the decision values are not binary either, since there are, in general, more than two variants to choose from. These more general Decision Graphs are known as *Multi-Terminal Multi-Valued Decision Diagrams* (MMDDs) [34]. They correspond to rooted directed acyclic graphs with inner nodes representing context attributes and leaves representing decisions.

The learning algorithm captures a decision relation to Decision Graphs similarly to Decision Trees by applying the same *learning* algorithm C4.5 presented in the previous section. The advantage of Decision Graphs over Decision Trees is a redundancy elimination that can lead to a decreased size

and, therefore, lower memory consumption. The construction of the nodes of the graph is done by a decision function constructor implemented in Decision Graphs. It requires the same input as a Decision Tree constructor. However, unlike to tree nodes, every graph node is created when each corresponding DS_v is processed.

The implementation of Decision Graphs uses a *repository* capturing graph nodes. However, the repository guarantees that each node corresponding to a unique decision (sub-)function, reduced by redundancy elimination, cf. Section 3.1, is captured only once and gets a unique identifier. The children in the decision nodes only refer to the unique identifier of the corresponding sub-graphs. As a consequence, two *equivalent*, cf. Section 3.1, graph nodes are never stored in the repository, and there are never two equivalent sub-graphs in the same decision node.

The remaining parts of the implementation of Decision Graphs follow the same strategy as the implementation of Decision Trees and, therefore, a set of standard operations over Decision Graphs is equal to a set of standard operations over Decision Trees. Moreover, the implementation of Decision Algebra core operations *invert* and *apply* is the same as for Decision Trees, and, therefore, their representation in this section is omitted. The only difference is the constructor implementation discussed above.

Decision Graphs: Related Work

The idea of Decision Graphs is not new. Several studies present Decision Graphs as an alternative decision model that overcomes several issues of Decision Trees. For instance, Oliver [56] presents Decision Graphs similar to χ -terms as a modification of Decision Trees. However, Oliver had to invent a completely new learning algorithm that builds a decision graph based on the so-called *Minimum Description Length Principle*, which minimized the number of encoding bits used for graph implementation.

Another type of Decision Graphs was presented by Jensen *et.al* [31]. These Decision Graphs are built on the concept of Bayesian Networks, which form the basis for so-called Influence Diagram Framework. This framework denotes a number of properties for Decision Graphs, where the core conception for all properties is based on the approach of collapsing identical subtrees in Decision Trees. The identity of subtrees in Influence Diagrams is defined similarly to the identity of decision functions: if both subtrees give the same decision in the end, they are pronounced to be the same regardless of the order of attributes. However, the purpose of introducing Decision Graphs is building normative systems, namely Decision Support Systems. Hence, the Influence Diagram Framework is concentrated on solving problems related to normative systems and is rather heavy to apply to different applications dealing with simple classification tasks.

Non-Functional Properties

Generally, decision time and worst case memory size of graphs are the same as for Decision Trees. Thus, the worst case memory and lookup overhead for Decision Trees and Decision Graphs are equal, but, due to the elimination of redundancies, the size of Decision Graphs is expected to be considerably smaller in practice.

5.3 Dispatch Tables

Dispatch Tables were introduced as a standard decision model used in context-aware composition in Section 2.4. Basically, Dispatch Tables are a table-based representation of a given complete and non-contradictive decision relation which is based on a set of tuples $(\vec{a}, c)$, where $\vec{a}$ is an actual context, i.e., a vector of attribute values $(a_1, \dots, a_n)$ with each attribute value $a_i \in A_i$, that leads to a certain decision c .

A Dispatch Table is usually sorted in terms of attributes and attribute values. This means that the rows in the table and the values of attributes in each row are sorted in a predefined order. To construct a Dispatch Table based on an arbitrary decision relation, a *learning* algorithm using a decision capturing or a decision interpolation approach, cf. Section 4.1, 4.2, ought to be applied. The chosen algorithm makes sure that the decision relation is complete and non-contradictive, even though it can contain duplicate decision tuples. Each attribute A_i of this decision relation corresponds to a Dispatch Table dimension, and, thus, each attribute value $a_i \in A_i$ is an entry in this dimension. Therefore, a constructor of a table takes each decision tuple $(\vec{a}_i, c)$, sorts its values of attributes according to the order of the attributes in the Dispatch Table, and inserts a decision c to a table cell identified by $\vec{a}_i$. The distributions corresponding to each decision can be taken into account in case when the Dispatch Table is constructed using a decision interpolation approach. However, the distributions in Dispatch Table implementations are often neglected. In a scenario where an attribute is continuous, the Dispatch Table captures variants at sample points.

The standard operations on a Dispatch Table data structure are: *insert* $(\vec{a}, c)$ inserts a decision $c \in C$ to a cell in a Dispatch Table with respect to attribute values of actual context $\vec{a}$; *lookup* $(\vec{a})$ returns a decision $c \in C$ contained in a cell corresponding to attribute values of the actual context $\vec{a}$. Before making a *lookup*, the closest sample point for each actual context attribute value needs to be computed. Assume we sample discrete attributes completely and continuous attributes with a logarithmic distance. Then we find the index corresponding to a discrete attribute value by a mapping table lookup, and the index of a continuous attribute value by computing the logarithm of the

actual attribute value. Finding the sample point closest to an actual context continuous attribute value must be done prior to each table lookup. In contrast to that, Decision Trees and Decision Graphs can decide on continuous attributes directly.

The Dispatch Table has a different representation than Decision Trees and Decision Graphs. Therefore, in addition to non-core operations of the Decision Algebra such as *merge* and *approximate* this data structure has alternative interpretations of these operations. Speaking the language of Object-Oriented Programming, the implementation of these operations can be overwritten in this decision model.

Apply

In order to *apply* function f on Dispatch Table $d \in DT$, the table d has to be traversed cell by cell and operator op has to be applied to each cell.

In case when the *apply* operation has to be performed on two Dispatch Tables $d_1, d_2 \in DT$, a new table $d \in DT$ is constructed. The operation traverses these tables and looks for decision tuples which are equivalent in terms of attribute values. Assume that two given Dispatch Tables d_1, d_2 represent the same formal context $\vec{A}$ and have the same type of sorting. Then, this *apply* operation can be formalized in Algorithm 11. The algorithm applies operator op to the cells of Dispatch Tables having the same set of attribute values $\vec{a}$, cf. line 2, and puts the result to a new table d , cf. line 3. This approach is

Algorithm 11 $apply(op \in C \times C \rightarrow C, d_1, d_2 \in DT) \rightarrow d \in DT$

```

1: for each  $(\vec{a}, c) \in d_1$  do
2:    $c' \leftarrow d_2.lookup(\vec{a})$ 
3:    $d.put(\vec{a}, op(c, c'))$ 
4: end for
5: return  $d$ 

```

similar to the one presented for Decision Trees and Decision Graphs, where the operator op is pushed towards the leaves and applied to them. In case Dispatch Tables represent a different formal context $\vec{A}$, those tuples that do not have equivalent representatives in tables are automatically added to a new constructed table.

Evert

Because the Dispatch Table is a table-representation of a complete and non-contradictive decision relation, the operations *merge* and *evert* are rather simple. To *evert* a Dispatch Table according to the chosen attribute A_i , we change the order of attributes in the Dispatch Table by considering attribute

A_i as a first attribute used in lookup operation, and, thus, attribute A_i becomes a first dimension of the Dispatch Table. In case that a Dispatch Table is represented by a Decision Table, as an example, cf. Table 2.1(a), we can say that we change the column order of the table by moving the column corresponding to attribute A_i to the left hand side.

Merge

Additionally to the *merge* operation, which can be implemented by the *apply* operation, there exists a more simple approach to merge two Dispatch Tables: we simply merge their decision tuples $(\vec{a}, c)$ and add them to a new Dispatch Table. If two tuples having equivalent actual contexts lead to different decisions, the decisions are merged in a resulting cell or, alternatively, a class distribution is computed.

Approximate

Algorithm 12 $\text{approx}(d' \in DT, A_i \in \vec{A}) \rightarrow d \in DT$

```

1: for each  $\vec{a} = (a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n) \in A_1 \times \dots \times A_{i-1} \times A_{i+1} \times \dots \times A_n$  do
2:    $c \leftarrow \perp$ 
3:   for each  $a_i \in A_i$  do
4:     let  $\vec{b} = (a_1, \dots, a_i, \dots, a_n)$ 
5:      $c \leftarrow c \sqcup d'.\text{lookup}(\vec{b})$ 
6:   end for
7:    $d.\text{put}(\vec{a}, c)$ 
8: end for
9: return  $d$ 

```

Approximation of a Dispatch Table refers to removing a certain attribute A_i from a table representation. As one possible approach, the corresponding attribute can be simply removed from the table and, thus, does no longer influence the decision. An alternative approach is the one presented for Decision Algebra, cf. Section 3.4, which merges decisions in table cells. We present it in a separate Algorithm 12.

In order to approximate the Dispatch Table d' by removing the certain attribute A_i , the table cells should be merged for all attribute values $a_i \in A_i$ provided that values of other attributes $(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n)$ are the same, cf. lines 3 – 6. A new decision is added to a new Dispatch Table d with respect to an actual context $\vec{a}$ that does not include attribute A_i , cf. line 7.

Non-Functional Properties

Dispatch Tables are implemented as n -dimensional arrays, n being the number of context attributes. Each dimension i contains entries corresponding to the sample values of the context attribute A_i . Thus, the memory consumption M of the Dispatch Table grows exponentially with the number of context attributes and can be approximated from below by

$$M = \text{size} \times m^n,$$

where size Bytes are necessary to encode all variants in C , and m is the minimum number of values of any of the context attributes.

Decide corresponds to a lookup in the table based on the indexes of input context attributes, where the index for a continuous attribute has to be computed with a logarithmic distance. Additionally, each access to an n -dimensional array is basically an access to a 1-dimensional array requiring some offset calculations:

$$\text{offset} = \text{base_address} + (((d_1 \times |A_2| + d_2) \times \dots \times |A_{n-1}| + d_{n-1}) \times |A_n| + d_n) \times \text{size},$$

where base_address is the address of the first element of the array, d_i is the index of the element in the dimension i and $|A_i|$ is the number of values of the context attribute A_i . Therefore, the decision time for an n -dimensional Dispatch Table can be estimated as

$$T(n) = (\log \times k + n) \times T_{flops} + (n - k + 1) \times T_{aa} + c,$$

where k is a number of continuous attributes, $\log$ is a number of floating point operations for calculating the logarithm¹, T_{flops} is the time for a flop, T_{aa} is the array access time, and c is a constant time used for small operations.

5.4 Conclusions

In this chapter we presented three existing decision models as Decision Algebra instantiations: Decision Trees, Decision Graphs, and Dispatch Tables, which serve as a mean to capture decision relations in three application domains discussed in Chapter 2.

For each decision model, its non-functional properties in terms of memory consumption and decision time were assessed. Due to this assessment, an appropriate decision model can be chosen for an application domain with certain non-functional requirements. For instance, Decision Trees fulfill the non-functional requirements of the classification domain: a decision relation

¹Many processors provide the integer $\log_2$ in a single instruction in hardware; in our Java implementation we need 21 flops.

is usually interpolated by learning algorithm and, thus, the decision model is scalable to the number of attributes; the fragmentation problem is solved by different pruning strategies, including approximation algorithms from the points-to analysis domain. However, Decision Trees still have the problem of replication which can increase required memory.

On the other hand, Decision Graphs, having the same characteristics and non-functional properties as Decision Trees, solve the replication problems due to the specifics in their construction which is based on a redundancy elimination process. Therefore, Decision Graphs can even serve as a better decision model in the classification domain and, for some tasks, in the context-aware composition domain. Moreover, Decision Graphs fulfill all non-functional requirements of the points-to analysis domain. By applying a simple learning algorithm, they can capture a complete and non-contradictive decision relation in a compact representation due to the decision function properties they implement: redundancy elimination and equivalence. Additionally, different implementations of pruning and approximation can reduce amount of the points-to information.

Decision Tables may be considered as a non-scalable but precise and fast data structure that is suitable in context-aware composition domain when a precise representation of a decision relation is required. They ought to perform well if they capture decision relation with a small number of non-continuous attributes.

Each decision model includes an implementation of the core operations of the Decision Algebra. Following the signatures provided by the Decision Algebra, implementations of these operations on the existing decision models are not more complex than if they were not implementing the Decision Algebra interface. Therefore, the implementation of existing decision models as Decision Algebra instantiations does not impose any complexity overhead compared to if these decision models were implemented as stand-alone models.

In the next chapter we give an overview of two additional decision models: Naive Bayes classifier based on a probability model, and Support Vector Machines based on a maximum-margin model; they have different construction strategies. We show that even these types of decision models can fit to our Decision Algebra.

Chapter 6

Probability- and Maximum Margin-based Decision Models Integrated into Decision Algebra

In Chapter 3 we introduced the Decision Algebra that unifies the theory behind the process of capturing and manipulating decision relations. The Decision Algebra provides a general representation of complete and non-contradictive decision relations, denoted as decision functions, along with a set of operations and axioms over these decision functions. In Chapter 4 we then presented three existing decision models as Decision Algebra instantiations, where the core operations were implemented in accordance to each decision model. As it was shown, the decision functions, and, thus, the corresponding decision models have a specific representation, where each decision function consists of a set of sub-functions that lead to the distribution over classes C , denoted as $d(C) \in D(C)$.

In Section 2.2 we have briefly mentioned several other decision models that, similarly to the discussed models, are used as classifiers in the field of Data Mining and Machine Learning. The representations of these models differ from our general decision function representation. Therefore, in this chapter we present a generalization over the Decision Algebra, denoted as *Abstracted Decision Algebra*, which includes a generalization over the decision function notion and, consequently, the operation signatures (Section 6.1). Moreover, we will present linear classifiers, such as Naive Bayes classifier and Support Vector Machines, along with a set of operations that follow the operation signatures of the Abstracted Decision Algebra, cf. Sections 6.2 and 6.3, respectively. Naive Bayes classifier is based on a probability model, and Support Vector Machines is based on a maximum margin model.

6.1 Decision Algebra: Generalization

Decision models such as Decision Trees, Decision Graphs and Dispatch Tables can be basically considered as a set of attribute values leading to a

set of distributions $d(C) \in D(C)$ over class values C . The general representation of this type of decision model is a decision function of the form $df = A_1 \times \dots \times A_n \rightarrow D(C)$.

However, there exist other types of classifiers where constructed decision models have different representations and, therefore, instead of a set of distributions, the results of decision functions can correspond to a set of class probabilities over attribute values (Naive Bayes classifier), or a set of computed coefficients (Support Vector Machines), etc.

Hence, the decision function df over finite domains has to be defined more generally in order to capture different types of computation results over decisions. From now on we represent a decision function df as a mapping of contexts (attributes) $A_1, \dots, A_n$ to a general decision representation $X(C)$ over a classification decision C :

$$\begin{aligned} C &\sqsubseteq X(C) \\ df &= A_1 \times \dots \times A_n \rightarrow X(C) \end{aligned}$$

Accordingly, the signatures of operations presented over decision functions in Chapter 3 have to be redefined, since the decision functions are not based on distributions any more. The redefinition of decision functions and a set of operations signatures leads to a definition of a new multi-sorted term Decision Algebra DA_A , which we will refer to as an *Abstracted Decision Algebra*. Let us define:

$$DA_A = \langle DF_c, \Omega, R \rangle$$

where DF_c is a collection of sets of decision function arguments $DF_c = \{DF_\lambda | \lambda \in S\}$, where decision functions have the same signature $A_1 \times \dots \times A_n \rightarrow X(C)$; DF_λ is a set of arguments of the same sort, called domain; S is a set of sorts of the domains (attributes, general decisions); Ω is a set of operations defined over DF_c , and R is an equivalence relation.

The equivalence relation partitions the terms of the Algebra DA_A into a number of equivalent classes. This relation enables us to say that certain decision functions which are syntactically distinct do, nevertheless, have the same representation. Therefore, the partitioning of all the terms of DA_A into the appropriate collection of equivalence classes, corresponding to different types of decisions that can be captured by decision function, leads to the Decision Algebra DA defined over distributions. The carrier sets of this algebra consist of decision functions with co-domain corresponding to distribution sets. This Decision Algebra is called a *multi-sorted quotient algebra* over algebra DA_A , where the operations are induced by operations described in DA_A . Thus, any Decision Algebra determined to define a concrete decision model can be considered as a quotient algebra over DA_A [17, 19]. In order to redefine algebra operations over a decision function corresponding to a

concrete class of decision models, we have only to determine which type of decision $X(C)$ a decision function has to capture.

In what follows we will redefine a set of operations signatures for the new Abstracted Decision Algebra. Notice that the selection of core and non-core operations is left unchanged.

Decide

In order to *decide* based on a generalized decision function df , i.e., to come to a unique decision $c \in C$ for a given actual context $\vec{a}$, we usually select the result of the decision function while applying a predefined operation on it. The reason is that the decision function can be learned by different strategies, and, thus, the interpretation of its result $X(C)$ can differ. Therefore, we apply the function $f : X(C) \rightarrow C$ to (the results of) a decision function $df : A_1 \times \dots \times A_n \rightarrow X(C)$:

$$\begin{aligned} df &: A_1 \times \dots \times A_n \rightarrow X(C) \\ f &: X(C) \rightarrow C \\ decide &: (A_1 \times \dots \times A_n \rightarrow X) \times A_1 \times \dots \times A_n \rightarrow C \\ decide(df) &= f \circ df \end{aligned}$$

The function f has to derive a concrete decision $c \in C$ based on a particular decision representation $X(C)$ that a decision function returns on a given actual context $\vec{a}$. Function f depends on a decision representation $X(C)$ and has to be determined along with the decision procedure, i.e., how a concrete decision has to be computed, for a specific decision model before its construction. For instance, in Decision Trees we often apply the *mode* operation to distributions in order to get a final decision; in Naive Bayes classifier a specific computation has to be made based on a final set of probabilities representing $X(C)$; in Dispatch Tables no such special function is required since usually a captured decision corresponds to $c \in C$, etc.

Apply

A general function f can be applied to decision functions, provided that f is determined on the argument $X(C)$ of the decision function df . We define such a general *apply* of arbitrary k -ary functions f to k -tuples of decision functions as:

$$\begin{aligned} f &: X_1(C) \times \dots \times X_k(C) \rightarrow X(C) \\ apply(f, x(c) \dots x_k(c)) &= f(x(c) \dots x_k(c)) \\ df_1, \dots, df_k &: A_1 \times \dots \times A_n \rightarrow X(C) \\ apply(f, df_1 \dots df_k) &: \rightarrow df \\ df &= A_1 \times \dots \times A_n \rightarrow f(x_1(C), \dots, x_k(C)) \end{aligned}$$

The operation $apply(f, df_1, \dots, df_k)$ recursively applies f to the respective arguments $A_1, \dots, A_n$ of a decision function and eventually evaluates it on $X(C)$; the result is a new decision function over $X(C)$.

Merge is a derivative operation of the *apply* operation. The aim of the *merge* operation is to combine decision functions $df_1, \dots, df_k$ to get a new decision function df that captures decision information from all $df_1, \dots, df_k$:

$$\begin{aligned} \sqcup & : X(C) \times X(C) \rightarrow X(C) \\ df_1, \dots, df_k & : A_1 \times \dots \times A_n \rightarrow X(C) \\ \sqcup(df_1, \dots, df_k) & = apply(\sqcup, df_1, \dots, df_k) \end{aligned}$$

The specific operation $\sqcup$ has to be determined in each concrete decision model.

Evert

The reordering of attributes inside a decision function still holds as a core operation in the Abstracted Decision Algebra. *Evert* produces a modification of a given decision function df , where it changes the order of attributes but not the captured decision information. This operation can be defined as:

$$\begin{aligned} evert(i) & : (A_1 \times \dots \times A_n \rightarrow X(C)) \rightarrow \\ & (A_i \times A_1 \times \dots \times A_{i-1} \times A_{i+1} \times \dots \times A_n \rightarrow X(C)) \\ df & \equiv evert(i, df) \end{aligned}$$

Approximate

The approximation of a decision function df is performed by ignoring an attribute A_i in this decision function and can be defined as:

$$\begin{aligned} approximate(i) & : (A_1 \times \dots \times A_n \rightarrow X(C)) \rightarrow \\ & (A_1 \times \dots \times A_{i-1} \times A_{i+1} \times \dots \times A_n \rightarrow X(C)) \end{aligned}$$

Notice that *approximate* operation defined in the Abstracted Decision Algebra DA_A is not based on a *merge* operation, unlike its definition in Section 3.4. In DA_A we can not theoretically guarantee that the result of an approximated decision function df will be less accurate but not wrong due to the difference in the decision representation $X(C)$ over classes C and in determination of the *merge* operation.

In what follows we will give an overview of Naive Bayes classifier and Support Vector Machines as instances of the Abstracted Decision Algebra. For each classifier we will define the decision representation $X(C)$ it captures, construction process, and a set of operations.

6.2 Naive Bayes Classifier

A Naive Bayes classifier is a simple probabilistic classifier based on Bayesian statistics. It naively assumes a conditional independence of the context attributes from each other. This means that the effect of an attribute value on a given class is independent of the values of other attributes. This assumption is made to simplify the computations that classify a tuple by determining a probability of its belonging to a certain class $c \in C$ [49, 36].

A Naive Bayes classifier computes a set of probabilities $P(C|\vec{A})$ for each attribute vector $A_i \in \vec{A}$ over all class values C . Thus, for a Naive Bayes classifier a decision representation $X(C)$ over C corresponds to a Cartesian product $\mathcal{P}(C)^n$ over all class values C and attribute values $A_i \in \vec{A}$ representing all the probabilities in $P(C|\vec{A})$:

$$\mathcal{P}(C)^n = P(C \times \vec{A}) = \{P(C|A_1), \dots, P(C|A_n)\} \quad (6.1)$$

where n is the number of attributes.

Construction

To compute a decision representation $\mathcal{P}(C)^n$ from a decision relation represented by $\vec{A}$ and C , let us assume that C is a binary class with values c_1 and c_2 and $\vec{A}$ is a formal context of n attributes. Each tuple is represented by an n -dimensional actual context $\vec{a} = (a_1, a_2, \dots, a_n)$, where $a_i \in A_i$. Applying Bayes' theorem, $P(C|\vec{A})$ can be computed as:

$$P(C|\vec{A}) = \frac{P(\vec{A}|C) \times P(C)}{P(\vec{A})}$$

That is, the Naive Bayes classifier predicts that tuple $\vec{a}_i$ belongs to class c_1 if and only if:

$$P(C = c_1|\vec{a}_i) > P(C = c_2|\vec{a}_i)$$

Thus, in order to determine a final decision, a maximization of $P(C = c_1|\vec{a}_i)$ has to be performed [25]. The class c_1 for which $P(C = c_1|\vec{a}_i)$ is maximized is called *maximum posteriori hypothesis*. By Bayes' theorem:

$$P(C = c_1|\vec{a}_i) = \frac{P(\vec{a}_i|C = c_1) \times P(C = c_1)}{P(\vec{a}_i|C = c_1) \times P(C = c_1) + P(\vec{a}_i|C = c_2) \times P(C = c_2)} \quad (6.2)$$

where $\vec{a}_i$ denotes the i -th possible vector of $\vec{A}$, and the summation in the denominator is done over all values of class C . However, only the numerator

$P(\vec{A}|C) \times P(C)$ has to be maximized, since the dominator is constant for each class $c_i \in C$. Accordingly,

$$P(\vec{a}_i|C = c_i) = \prod_{k=1}^n P(a_k|c_i) = P(a_1|c_i) \times P(a_2|c_i) \times \dots \times P(a_n|c_i) \quad (6.3)$$

$$P(C = c_i) = \frac{n_{c_i}}{|DR|} \quad (6.4)$$

where DR is a set of decision tuples and n_{c_i} is a number of tuples of class c_i in DR . The resulting set of probabilities $P(C)^n$ can be defined as:

$$\begin{aligned} \mathcal{P}(C)^n = P(\vec{A}|C) &= P(A_1|c_1) \times \dots \times P(A_1|c_m) \times \dots \times \\ &P(A_n|c_1) \times \dots \times P(A_n|c_m) \end{aligned}$$

where $P(A_i|c_i)$ is a set of probabilities $P(a_k|c_i)$ of occurrence attribute value $a_k \in A_i$ with a class value $c_i \in C$ in one decision tuple, and m is a number of possible classes. The probability $P(a_k|c_i)$ can be estimated from the given decision relation as follows:

$$P(a_k|c_i) = \frac{|DR(A_i = a_k, C = c_i)|}{n_{c_i}} \quad (6.5)$$

where $|DR(A_k = a_k, C = c_i)|$ is a number of decision tuples of class c_i where attribute A_i has value a_k .

Notice that, for each attribute, we check whether it is categorical or continuous. Assume we need to classify a new tuple $\vec{a} = \{a_1, a_2, \dots, a_n\}$, where a_k is a value of attribute A_i . To compute $P(\vec{a}|C = c_i)$, we consider for each attribute A_i the following:

- if A_i is categorical, then we take $P(a_k|C = c_i)$ as presented in Equation 6.5.
- if A_i is continuous:

$$P(a_k|C = c_i) = g(a_k, \mu_{c_i}, \sigma_{c_i}^2)$$

That is, we have to compute a *Gaussian distribution* g for class c_i . A continuous-valued attribute is typically assumed to have a Gaussian distribution with a *mean* μ and *variance* σ :

$$g(a_k, \mu_{c_i}, \sigma_{c_i}^2) = \frac{1}{\sqrt{2\pi\sigma_{c_i}^2}} e^{\frac{-(a_k - \mu_{c_i})^2}{2\sigma_{c_i}^2}} \quad (6.6)$$

with

$$\mu_{c_i} = \frac{1}{n_{c_i}} \sum_{i=1}^n a_i \quad (6.7)$$

$$\sigma_{c_i}^2 = \frac{\sum_{i=1}^n (a_i - \mu_{c_i})^2}{n_{c_i} - 1} \quad (6.8)$$

where n_{c_i} is a number of decision tuples of class c_i and a_i is a value of attribute A_i for each tuple of class c_i .

Decide

In order to predict a class value for an actual context $\vec{a}$, $P(\vec{a}|C = c_i)P(C = c_i)$ has to be evaluated for each class c_i . Thus we apply the function corresponding to Equation 6.3 to our decision representation $\mathcal{P}(C)^n$. The Naive Bayes classifier selects the most likely classification c as following:

$$c = \arg \max_{c_i \in C} P(\vec{a}|C = c_i)P(C = c_i)$$

Notice that, the case when a probability $P(a_k|C = c_i)$ is equal to zero leads to zero in the final probability $P(\vec{a}|C = c_i)$. The reason of this situation is an absence of instances of class c_i with attribute A_i having value a_k . In order to avoid this situation there exist two possible approaches: (1) to use a predefined value ϵ instead of the 0-probability, or (2) to use a Laplacian correction or Laplace estimator [25].

Apply

In order to apply a function f on a Naive Bayes classifier NB , f has to be defined over a decision representation $\mathcal{P}(C)^n$.

As an example of such an operation let us consider a merge operation, where we apply $\sqcup$ on two Naive Bayes classifiers NB_1 and NB_2 . The *merge* operation $\sqcup$ on two Naive Bayes classifiers NB_1 and NB_2 refers to the sum of probabilities with respect to each class value $c_i \in C$ and attribute value $a_i \in A_i$. Therefore, a new classifier $NB = NB_1 \sqcup NB_2$ is constructed as follows: The probability $P_{NB}(C = c_i)$ of a new classifier NB is

$$P_{NB}(C = c_i) = \frac{P_{NB_1}(C = c_i) + P_{NB_2}(C = c_i)}{2}$$

However, when it comes to computing a new probability $P(A_i = a_k|C = c_i)$ there is a difference in computation based on whether attribute A_i is categorical- or continuous-valued. For a categorical attribute A_i and class

$C = c_i$, a probability $P_{NB}(A_i = a_k|C = c_i)$ of a new classifier NB is computed similar to class probability:

$$P_{NB}(A_i = a_k|C = c_i) = \frac{P_{NB_1}(A_i = a_k|C = c_i) + P_{NB_2}(A_i = a_k|C = c_i)}{2}$$

For continuous attributes we need to sum the variance σ^2 and mean μ for each class value c_i . Therefore, the new mean value μ_{NB} is:

$$\mu_{NB} = \mu_{NB_1} + \mu_{NB_2} = \frac{\mu_{NB_1} \times n_{c_i}^1 + \mu_{NB_2} \times n_{c_i}^2}{n_{c_i}^1 + n_{c_i}^2}$$

and a new variance σ_{NB}^2 is:

$$\begin{aligned} \sigma_{NB}^2 &= \sigma_{NB_1}^2 + \sigma_{NB_2}^2 = \\ &= \frac{n_{c_i}^1(\sigma_{NB_1}^2 + (\mu_{NB_1} - \mu_{NB})^2) + n_{c_i}^2(\sigma_{NB_2}^2 + (\mu_{NB_2} - \mu_{NB})^2) - \sigma_{NB_1}^2 - \sigma_{NB_2}^2}{n_{c_i}^1 + n_{c_i}^2 - 1} \end{aligned}$$

where n_{c_i} is the number of tuples of class c_i , and μ_{NB_i} and $\sigma_{NB_i}^2$ are mean and variance of attribute A_i corresponding to tuples with class c_i .

Evert

The interpretation of *evert* is quite simple due to the main property of Naive Bayes classifier – the independence of attributes from each other. Therefore, the order of attributes (attribute probabilities) does not influence the final probability $P(\vec{a}|C = c_i)$, and it can be changed arbitrary.

Approximate

The same approach corresponds to the *approximate* operation: we can exclude any attribute probability replacing it with 1 or a predefined value ϵ . For instance, we need to classify vector $\vec{a} = \{a_1, a_2, a_3\}$ based on an approximated classifier where attribute A_3 is excluded, then:

$$P(\vec{a}|C = c_i) = P(a_1|C = c_i) \times P(a_2|C = c_i) \times \epsilon$$

Non-functional requirements

Naive Bayes classifiers use the decision relation and compute all possible probabilities for each attribute vector $\vec{a}_i$ over all class values C . As a result, a Naive Bayes classifier is just a set of probabilities that are accessed during classification for computing the most probable variant. For a discrete attribute A_i , the probability is stored in an array with $|C| \times |A_i|$ elements, where

C is a set of all possible class values; for a continuous attribute, a *mean* and a *variance* are computed and stored in two arrays of size $|C|$. So the memory consumption is

$$M = |C| \times size \times (2k + 1) + (n - k) \sum_{i=1}^{(n-k)} |A_i|,$$

with n being the number of all attributes and k being the number of continuous attributes.

A *decision* using a Naive Bayes classifier takes quite some time: it requires 4 flops for each discrete attribute and 88 flops for each continuous attribute (including mathematical operations computing Gaussian) for each possible class. Thus, the decision time is estimated as

$$T(n) = (4n + 84k) \times T_{flops} + (2n + k) \times T_{aa} + c$$

where T_{flops} is the time for a flop, T_{aa} is the array access time, and c is a constant time used for small operations.

6.3 Support Vector Machines

Support Vector Machines (SVMs) have been widely applied to pattern recognition, classification problems and non-linear regression [6, 77]. They represent a statistical learning algorithm first presented by Vapnik [13]. The basic idea of SVMs is to find an optimal linear separating hyperplane in a vector space that can maximize a margin between two groups of samples representing two different classes $c_1 = +1$ and $c_2 = -1$, $c_1, c_2 \in C$.

Support Vector Machines compute a set of support vectors S corresponding to a subset of decision tuples, and a set of corresponding coefficients $\mathcal{A}$. Each support vector $s_i \in S$ has a specific coefficient $\alpha_i \in \mathcal{A}$. Thus, for Support Vector Machines a general decision representation $X(C)$ corresponds to $C \times \mathcal{A}$, since every support vector is a decision tuple $(\vec{a}, c)$ where a set of attribute values leads to a decision $c \in C$.

In what follows we will give a short overview of the main principles of Support Vector Machines taken from [6, 25, 77, 83].

Construction

Let l input decision tuples $(\vec{a}_i, c) \in DR$ that belong to two classes $c_i \in \{-1, +1\}$, with n being the number of attributes be given. They can be linearly separated if there exists a hyperplane:

$$y(\vec{a}) = \vec{w}^T \vec{a} + b_0 = 0 \quad (6.9)$$

Figure 6.1: An optimal hyperplane for classification in two-dimensional space for a linearly separable case [25]

that separates two classes such that $y(\vec{a}_i) \geq 0$ for all decision tuples with $c = +1$ and $y(\vec{a}_i) < 0$ for all decision tuples where $c = -1$. $\vec{w}^T = \{w_1, w_2, \dots, w_n\}$ is a weight normal vector perpendicular to the hyperplane, $\vec{a} = \{a_1, a_2, \dots, a_n\}$ is a decision tuple corresponding to the input vector of attribute values graphically represented as a dot in n -dimensional space $\mathbb{R}^n$, and b is a bias parameter.

Linearly separable data The input decision tuples can be linearly separable. In this case, the construction of a separating hyperplane is discussed in the following. There can exist several separating hyperplanes which classify the decision tuples explicitly. Hence, the task for the algorithm is to find the one that will give the smallest generalization error [6], i.e., the hyperplane which has a maximum *margin*. The *maximum margin hyperplane MMH* is the one that gives the greatest separation between the classes and is a perpendicular to the shortest line connecting the decision tuples from different classes, represented by points in n -dimensional space. The shortest distance from the hyperplane to one side of its margin should be equal to the shortest distance from the hyperplane to the other side of its margin. An example is shown in Figure 6.1, where each tuple from the decision relation is represented by a dot in two-dimensional space. The parallel hyperplanes H_1 and H_2 that lie on the borders of a separating hyperplane are shown as dashed lines and define the size of margin as:

$$\begin{aligned} H_1 : \vec{w}\vec{a} + b_0 &\geq +1, \text{ for } c = +1 \\ H_2 : \vec{w}\vec{a} + b_0 &\leq -1, \text{ for } c = -1 \end{aligned}$$

Every tuple that falls below or on H_2 belongs to class $c = -1$, and every tuple that falls above or on H_1 belongs to class $c = +1$. Those tuples that fall on hyperplanes H_1 and H_2 are called *support vectors* S and are depicted on Figure 6.1 with a circle. All decision tuples in a given decision relation will satisfy the following constraint that is a combination of the two equations above:

$$c_i(\vec{w}\vec{a}_i + b_0) \geq 1, i = 1, \dots, l \quad (6.10)$$

The objective of SVMs construction is to maximize the size of the margin between two classes that is equal to $\frac{2}{\|\vec{w}\|}$, i.e., to maximize $\|\vec{w}\|^{-1}$ or minimize $\|\vec{w}\|^2$. Thus, the optimization problem can be defined as [6, 77]:

$$\left\{ \begin{array}{l} \arg \min_{w,b} \frac{1}{2} \|\vec{w}\|^2 \\ c_i(\vec{w}\vec{a}_i + b_0) \geq 1, i = 1, \dots, l \end{array} \right. \quad (6.11)$$

This optimization problem with constraints can be solved by Lagrangian multipliers $\alpha_i \in \mathcal{A}, \alpha_i \geq 0$, with one multiplier for each constraint in Equation 6.10. In the end, the problem of finding the maximum margin is reduced to a quadratic programming problem with dual coefficients, where the Lagrangian function has to be maximized with respect to several constraints:

$$\left\{ \begin{array}{l} L(\alpha) = \sum_{i=1}^l \alpha_i - \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l \alpha_i \alpha_j c_i c_j \vec{a}_j \cdot \vec{a}_i \rightarrow \max_{\alpha} \\ \alpha_i \geq 0, n = 1, \dots, l \\ \sum_{i=1}^l \alpha_i c_i = 0 \end{array} \right. \quad (6.12)$$

Solving a constrained quadratic problem (QP) usually requires $O(l^3)$, where l is the number of decision tuples [40]. The description of this problem and the approach to solve it can be found in [6, 5].

Based on the Lagrangian formulation, the maximum margin hyperplane can be rewritten as a decision boundary:

$$y(\vec{a}^T) = \sum_{i=1}^l c_i \alpha_i \vec{a}_i \cdot \vec{a}^T + b_0 \quad (6.13)$$

where $\vec{a}^T$ is a test tuple, $\vec{a}_i \cdot \vec{a}^T$ is the *dot product* of two vectors, that is:

$$\vec{a} \cdot \vec{a}^T = \sum_{j=1}^n a_j \times a_j^T,$$

with n being the number of attributes.

Obviously, any decision tuple $(\vec{a}_i, c)$ for which $\alpha_i = 0$ will not influence the result of the Equation 6.13 and, hence, it plays no role in making a final decision $y(\vec{a}^T)$. The remaining decision tuples are called *support vectors* and correspond to points that lie on hyperplanes of maximum margin. We will refer to support vectors as $s_i \in S$.

Knowing coefficient $\alpha_i \in \mathcal{A}$ for each support vector $\vec{s}_i$, we can identify a normal vector $\vec{w}$:

$$\vec{w} = \sum_{i=1}^{|S|} \alpha_i \vec{s}_i \quad (6.14)$$

Hence, the complexity of a classifier is determined by the number of support vectors $|S|$ which are a subset of training tuples rather than the number of attributes. Therefore, the data that are computed by the construction algorithm correspond to a set of support vectors S and a set of coefficients $\mathcal{A}$. Since a support vector is a decision tuple, the general decision representation $X(C)$ corresponds to a set of classes and a set of coefficients $C \times \mathcal{A}$.

Linearly non-separable data If the decision relation cannot be linearly separated by hyperplanes in the current dimension space, the algorithm uses a nonlinear mapping function $\Phi : \mathbb{R}^n \rightarrow H$ to transform the decision tuples from the original space $\mathbb{R}^n$ into a higher dimensional space H where the tuples can be linearly separated. The SVMs construction method argues that the decision tuples from two classes can always be linearly separated by a hyperplane with an appropriate nonlinear mapping to a higher dimension space.

Combining the transformation and the linear optimal hyperplane we can formulate the separating hyperplane of a non-linear SVMs as follows:

$$y(\vec{a}_i) = \vec{w}^T \Phi(\vec{a}_i) + b_0 = 0 \quad (6.15)$$

where $\Phi(\vec{a}_i)$ is the function mapping the i -th decision tuple represented by a vector to a potentially higher dimension feature space.

The construction of SVMs in this case is very similar to the construction of a linear optimal hyperplane presented above. The only difference is that the input decision tuple $\vec{a}_i$ has been replaced by $\Phi(\vec{a}_i)$. Hence, the dot product $\vec{a}_i \vec{a}_j$ is also replaced by the dot product $\Phi(\vec{a}_i) \Phi(\vec{a}_j)$, e.g., in Equations 6.12 and 6.13.

Since we transfer data to a new dimension space H by introducing new dimensions, the operation of computing dot products becomes computationally expensive. However, instead of computing dot products, SVMs compute a kernel function $K(\vec{a}_i, \vec{a}_j)$ that implicitly maps the data into a higher dimensional feature space.

A function K is called kernel function if it can be expressed as:

$$K(\vec{a}_i, \vec{a}_j) = \Phi(\vec{a}_i)\Phi(\vec{a}_j)$$

for some mapping $\Phi : \mathbb{R}^n \rightarrow H$, where H is a space with a dot product.

The kernel function, therefore, represents a dot product in a feature space having more dimensions than the original space, where the mapping from the original space to feature space is described by vector function $\Phi(\vec{a})$. Due to the calculation of a kernel function, all the computations can be made in the original input space. However, evaluating and choosing a kernel function is often done experimentally. There exist different types of kernel functions that can be used in SVMs [25, 6, 28].

Decide

In order to decide given a test decision tuple $\vec{a}^T$, we apply a function f corresponding to the following equation on a decision representation $C \times \mathcal{A}$:

$$y(\vec{a}^T) = \sum_{i=1}^{|\mathcal{S}|} c_i \alpha_i \vec{s}_i * \vec{a}^T + b_0 \quad (6.16)$$

and determine the class by checking the sign of the result: if the sign is positive, then $c = +1$, otherwise $c = -1$.

Apply

In order to apply a function f on Support Vector Machines, this function has to be defined on the set of classes C and a set of corresponding coefficients $\mathcal{A}$.

In what follows we will discuss a *merge* operation as an operation derived from *apply*. In Support Vector Machines the *merge* operation can be interpreted in two different ways. The first one merges two Support Vector Machines by applying a pre-defined operation $\sqcup$ on coefficients and classes of both sets of support vectors. Basically, this corresponds to finding the average hyperplane by applying $\sqcup$ on two normal vectors w_1 and w_2 ($\sqcup(w_1, w_2)$). For this approach, a special operation $\sqcup$ has to be defined. However, its evaluation requires additional mathematical implications, and, therefore, we leave it as a matter of future work.

The second approach is widely used in Support Vector Machine construction and is referred to as *incremental learning*. The main disadvantages of SVMs are their large memory requirement and computation time. Therefore, the possibility of incremental learning could be very helpful especially when the decision tuples are obtained at different intervals. There exist several approaches to incremental learning based on Support Vector Machines [71, 68, 32, 18, 78, 10].

The approach presented in [71] for each new input set of decision tuples trains SVMs on these tuples together with support vectors from previous steps. The idea behind this approach is: the final decision of SVMs depends only on the decision tuples that represent support vectors. Thus, learning only on these tuples will give the same result as learning on the complete decision relation. Therefore, we can expect to get the same result from an incrementally learned SVMs on a set of support vectors from a previously learned classifier and a new decision relation as from a non-incrementally learned SVMs on two decision relations. Relying on the fact that the complexity for solving the QP problem required during SVMs learning is $O(l^3)$, where l is the number of given decision tuples, the complexity of an incrementally learned SVMs will be $O((l + |S|)^3)$, where $|S|$ is the number of support vectors obtained from previous learning. The complexity of re-learning an SVMs on the whole batch of decision tuples is $O((l + l')^3)$, where l' is the number of decision tuples from the previous step.

However, Rüping [68] argues that the assumption that a new decision relation is appropriate is wrong, i.e., the influence of previous support vectors on the decision function in the next learning step may be very small if the new decision tuples are distributed differently, so the final hyperplane does not differ significantly from the hyperplane obtained from the new decision relation. He claims that a new SVM result largely ignores the old support vectors and almost always corresponds to the decision function that would have been learned on the second decision relation alone. Based on these assumptions he proposes an approach of adding the additional weight coefficients for a set of old support vectors so some of them are guaranteed to be a part of new final support vectors.

Zhang *et al.* [32] propose a division-incremental algorithm that considers the possible impact of new decision tuples to previous learning results. Decision tuples are separated into smaller sets, and the final set of support vectors is obtained in a cross-computation way. The algorithm considers not only obtained support vectors but also the tuples between the margins that are usually classified incorrectly in each step of the incremental algorithm. The computation of the proposed approach is focused on constructing hyperplanes with several sets of support vectors. The aim of this approach is to combine potential decision tuples as the support vectors with regard to their influences to parts of the decision relation.

Additionally, there can be a case when the decision relations on which different SVM classifiers were learned are not available. The only data that is disposed is the batch of support vectors obtained from two classifiers. The solution for this case is based on the heuristic that we can merge SVMs relying solely on these sets of support vectors. Hence, the complexity for such re-learning is $O((|S_1| + |S_2|)^3)$, where $|S_1|$ and $|S_2|$ are the number of support

Figure 6.2: An example of a hyperplane of an SVM gotten from the re-learning on support vectors of two previous SVMs

vectors obtained from two SVMs, respectively. However, for this case we cannot guarantee that the set of support vectors obtained from two classifiers will correspond to the actual support vectors obtained from the learning on two decision relations. The theoretical problem behind this is similar to the one discussed by Rüping, i.e., the support vectors are a sufficient description of the decision boundary between the decision tuples, but not the tuples themselves. Therefore, this method can propagate the errors in the classification as presented in Figure 6.2. The dashed lines represent the margin of the separating hyperplane from SVM_1 with support vectors x_1, x_2 , the dashed with dots represent the margin of separating hyperplane from SVM_2 with support vectors y_1, y_2 . The solid lines, in turn, represent a new hyperplane as a result of learning new $SVM = SVM_1 \sqcup SVM_2$ basing on support vectors $\vec{x}_1, \vec{x}_2, \vec{y}_1, \vec{y}_2 \in S$. As shown, the tuples corresponding to vectors $\vec{a}, \vec{b}$ will be classified wrongly based on the new hyperplane: $\vec{a}$ ends up on hyperplane with $y(\vec{a}) = 0$, and $\vec{b}$ will be assigned the wrong class. However, the purpose of the heuristics for decreasing learning time is to find a good enough solution to merge SVMs and, thus, the error in this case can be accepted.

Approximate

The *approximate* operation for SVMs can be interpreted as reducing the number of dimensions in the separating hyperplane, that is, projecting a hyperplane to a less dimensional space by equating this dimension to zero. However, approximating a *valuable* attribute can lead to a new set of support vectors. An attribute is *valuable* if it influences the position of the support vectors the most. Therefore, in order to correctly reduce the number of dimensions, a less valuable attribute has to be chosen for approximation, or the SVM classifier has to be re-learned on the previous set of decision tu-

ples lacking one attribute. For instance, for the hyperplane represented in Figure 6.2 with dot-dashed lines, a valuable dimension is x , while a good attribute for approximation is y , since projecting of support vectors and decision tuples captured in classifying decision relation DR on dimension x will unlikely change the set of computed support vectors.

Evert

The *evert* operation is not essential for this type of decision function since the solution to Equation 6.13 does not depend on the sequence of attributes in each vector and, hence, they can be reordered arbitrarily.

Non-functional properties

In order to assess non-functional properties we refer to LIBSVM [11], a library for concrete implementations of Support Vector Machines. It is as a black box with a radial basis function for the kernel function and follows the normalization and parameter finding suggestions detailed in [29].

SVMs can be trained to be very accurate classifiers. However, as it was discussed above the training time for SVMs can be very long, especially on large sets of decision tuples, since it takes $O(l^3)$ computations, where l is the number of decision tuples. The encoded SVM classifier in the LIBSVM library has a quite high memory consumption for capturing the support vectors, approximated with

$$M = 12s + 24|C| + 4|C|^2 + 100$$

double precision values, where s is the number of support vectors and C is a set of class values.

More importantly, deciding requires an evaluation of the kernel function that takes for the selected kernel approximately $20 + (n + |C|)$ flops for each support vector and a number of additional flops for finding the actual variant, where n is the number of attributes. The number s of support vectors is usually proportional to the number of decision tuples l , so we can approximate

$$T(l, n) \approx T(s, n) \approx (20 + n + 2|C| - 1) \times s \times T_{flops} + s \times (2|C| - 1) \times T_{aa} + c$$

6.4 Conclusions

The Decision Algebra presented in Chapter 3 describes a special class of distribution-based decision models. The set of operations defined in the algebra is, basically, oriented to the type of decision functions represented by these decision models. The best-known instances of such decision models are: Decision Trees, Decision Graphs and Dispatch Tables. However, there exist other types of decision models such as Naive Bayes classifier and Support Vector Machines that do not have similar decision models to those presented before, and, therefore, the operation signatures outlined in the Decision Algebra can no longer define operations over the decision functions corresponding to these decision models.

In this chapter we have generalized our Decision Algebra to an Abstracted Decision Algebra. We have redefined the notion of decision functions along with a set of operation signatures. The Abstracted Decision Algebra is able to define not only the same type of decision models, but decision models that have different representations. Due to such a generalization, the Decision Algebra previously defined for distribution-based decision functions becomes a quotient algebra over the Abstracted Decision Algebra. Thus, by defining a decision representation $X(C)$ different types of decision models can be formalized to a Decision Algebra based on a concrete decision model representation. For instance, decision models representing Naive Bayes classifier and Support Vector Machines are formalized to instances of the Abstracted Decision Algebra as a probability-based Decision Algebra, where the construction is based on the computing of probabilities over attribute values, and a maximum margin-based Decision Algebra, where the construction process computes a separating hyperplane with a maximum margin between decision tuples representing different classification classes, respectively. These two decision models were explained in a very detailed way compared to the other approaches presented before. This is done in order to explain these decision models for those who are not experts in the fields of Data Mining and Machine Learning.

Based on the theoretical estimations of non-functional properties of decision models presented in this and in the previous chapter, we still cannot decide which decision model to prefer. It depends on the bias between acceptable decision time, memory overhead and decision accuracy. Thus, it depends on the concrete problem, i.e., the number of context attributes, decision tuples, etc. However, once the number of attributes and the decision tuples are known, the above estimations can be used to (pre-)select a preferred decision model ignoring the accuracy parameter, which can be determined only in experiments. In the next chapter we will present two series of experiments where these decision models will be evaluated in the classification and the context-aware composition domain.

Chapter 7

Decision Algebra: Implementation and Validation

In Chapters 5 and 6, we presented several existing decision models that can serve as variants of Decision Algebras instances: Decision Trees, Decision Graphs, Dispatch Tables, Naive Bayes classifier, and Support Vector Machines, along with their implementation details.

This chapter presents the results obtained from two experiment series performed with respect to our second research goal. The first experiment in Section 7.1 compares two decision models (research goal 2.1) corresponding to distributed-based decision functions constructed using the same decision relation but having different implementations: Decision Graphs are compared with Decision Trees. The comparison is based on the evaluation of a set of metrics: memory consumption, computational costs, and accuracy. The experiment, therefore, consists of two parts: the first part compares the memory consumption and time used for learning and deciding based on Decision Graphs and Decision Trees; the second part evaluates the accuracy of k -approximated Decision Graphs and post-pruned Decision Trees and compares the time required for both approximation approaches.

The second experiment is discussed in Section 7.2. We use context-aware composition, presented in Section 2.4, as an appropriate application domain which can benefit from decision model scalability. In the learning phase of the application, variants are tested in different actual contexts, and the champion variant of each context is captured in a Dispatch Table. The scalability of context-aware composition generally depends on the scalability of memory and speed of the plugged-in decision model. However, as discussed before in Section 2.4, Dispatch Tables are not very scalable. Hence, this makes context-aware composition a good candidate to validate decision models in a concrete domain with concrete requirements. As alternatives to Dispatch Tables we take Decision Trees, Decision Graphs, Naive Bayes classifier, and Support Vector Machines decision models. This experiment is based on a sorting problem. Sorting is an extreme problem for the context-aware composition domain in the sense that the ratio of dynamic decisions and payload operations is rather high. However, the insight on this extreme case can de-

liver a guidance for other cases, since it provides a general overview and interpretation of each learning technology.

Finally, Section 7.3 concludes this chapter with a short summary of the results of the experiments.

7.1 *Decision Graphs vs. Decision Trees*

In this section, we present two sub-experiments where we compare a graph-based representation of decision functions - Decision Graphs - with a well-known tree-based representation - Decision Trees. In both cases the decision functions are generated by the C4.5 learning algorithm from the same decision relation, which in this section is referred to as *dataset*.

Experimental Setup

Our experiments are performed on 16 different benchmark datasets from the UCI Machine Learning Repository [20]. While selecting these specific datasets, we were only interested in a classification problem and, therefore, selected the 14 largest with both categorical and continuous attributes. We also added two large datasets with only continuous attributes to show the benefits of our approach even in these cases.

The used datasets are presented in Table 7.1 in ascending order of the number of training instances. In addition to the dataset names, the table also reports on the number of training and test instances, the number of nodes/depths of the generated Decision Trees. The final column (Attributes) describes what type of attributes each dataset uses. We will present our experimental results for these datasets in the same order as in this table.

Implementation Details

We used the Decision Trees generated by the FC4.5 learning algorithm [60, 67] as a baseline to which we compare our graph-based implementation. FC4.5 is a fast implementation of the C4.5 learning algorithm outlined in Section 4.2. We adopted the FC4.5 algorithm to directly learn *both* Decision Trees and Decision Graphs. As a result, both representations have exactly the same classification accuracy when no additional pruning is applied¹. In order to make a fair comparison between the two representations, we had to make a few minor adjustments, though:

- (1) Each internal node of the Decision Tree constructed by FC4.5 keeps a training weight, a distribution, and a possible classification, information that is later used for decision making. In our Decision Graph

¹This is confirmed experimentally as well.

Id	Dataset	Training ins.		Test ins.		Tree Size/Depth		Attributes
1	ionosphere	309	42		21/7			34-categorical
2	cancer-wisconsin	500	199		125/4			1-cont., 9-categorical
3	australian	552	138		143/9			6-cont., 8-categorical
4	crx	600	119		174/9			6-cont., 9-categorical
5	dibetes	688	100		27/7			8-cont.
6	anneal	798	133		151/14			6-cont., 32-categorical
7	german	800	200		416/11			7-cont., 13-categorical
8	hypothyroid	2527	636		19/8			7-cont., 18-categorical
9	ad	3057	420		153/55			3-cont., 1556-categorical
10	waveform	4000	1000		515/18			21-cont.
11	nursery	11664	1294		905/7			8-categorical
12	chess	28042	4886		10001/5			6-categorical
13	adult	32561	16281		8124/20			6-cont., 8-categorical
14	connect-4	67557	13994		15940/22			42-categorical
15	census-income	159617	39906		46363/26			8-cont., 33-categorical
16	covtype	409985	171027		28389/63			10-cont., 45-categorical

Table 7.1: Dataset Characteristics

implementation, the internal nodes do not contain any such information. They just contain information about the attribute they represent. It is only the leaves that keep a classification distribution $d(C)$. Both representations use the same type of distributions, the frequency-based distribution presented in Section 3.2. Additionally, we take into account unknown attribute values by using counts less than one in the distributions [67]. Moreover, in cases when there are no instances distributed to a leaf, we use a *bottom distribution* where all classes have a frequency equal to the classification probability $1/|C|$. Due to these simplification, the repository is able to identify (and reuse) equivalent sub-graphs without losing any information.

- (2) As a consequence of (1), the *decide* operation used for both Decision Graphs and Decision Trees is a simplified version of *decide* as implemented in the FC4.5 algorithm. For example, when a test attribute value is missing, the test data is passed to all the children of the current node without taking into account the partitioned weight of the children. Because of this simplification, we expected to lose classification accuracy when comparing the results with the pure, unmodified FC4.5 implementation. However, the accuracy remained the same in all experiments. This observation justifies the simplifications presented in (1) where we neglect additional information kept in the internal nodes.
- (3) A continuous attribute A_i can be partitioned into different intervals in different branches of the tree. In these cases, we consider each new partitioning as a new categorical attribute and also coming with a new selection operator. As we will see when we discuss our experimental results, this drastically reduces the chance for the repository to identify redundancy due to equivalent sub-graphs in datasets where many continuous attributes are used.

These implementation details will be of importance when we later on discuss our experimental results.

Memory Consumption

In this sub-section, we present the result of the first sub-experiment. Our Decision Graph implementation recognizes identical subtrees and makes sure that we only keep one such instance. Thus, we expect the Decision Graphs to contain fewer nodes than the corresponding decision trees.

Figure 7.1 displays two bars for each dataset for comparing the number of nodes in the Decision Graphs (right) with the number of nodes in the Decision Trees (left, always scaled to 100%). For example, Bars 1 (ionosphere) show that the Decision Graph has the same size as a corresponding Decision

Figure 7.1: The percentage of reduced internal nodes and leaves compared to the total tree size (100%)

Figure 7.2: Times of learning and deciding based on a Decision Graph as % of Decision Tree (100%)

Tree, whereas Bars 2 (cancer-wisconsin) show that the Decision Graph contains only 36% of the nodes of the corresponding Decision Tree, indicating a 64% node reduction. The overall result, an average node reduction of 44%, indicates that much memory can be saved by using Decision Graphs as representation.

Each bar in the chart is also divided into two parts, separating internal nodes from leaves. When comparing the number of internal nodes in the trees with the corresponding graphs, we see that, in most cases, the numbers are almost the same, indicating that a majority (98%) of the reduced nodes are leaves. This has two reasons:

- A large part of the removed leaf nodes are *bottom leaves* representing attribute value combinations not covered by any instances in the dataset. Although associated with different weights, a majority of these leaves could be removed. The remaining part of the removed leaves are due to non-empty, but identical, distributions.
- The internal node reduction is quite small (4%). The major reason is that we treat different interval partitions of continuous attributes as entirely different attributes. Hence, the possibility of identifying identical subtrees in these cases is very low. This explanation is supported by the observation that in datasets where we are using mainly categorical attributes (2,11,12,14), we have a larger reduction of internal nodes. For example, case 11 (nursery) has a reduction of 37%.

Additionally, we have measured the time for construction the Decision Trees (DT) and Decision Graphs (DG) and also the time used for classifying (*decide*) a fixed number of instances for each dataset. The time was measured in milliseconds, but we have used a relative measure ($DG.Time/DT.Time$) in order to simplify the comparison. The results presented in Figure 7.2 show that the Decision Graph implementation is faster in almost all cases. The only exception is the graph construction in case 13. The average construction and classification time for Decision Graphs is about 19 and 20% less than for Decision Trees, respectively. The reduced classification time for Decision Graphs is at first glance a bit surprising, given that the number of selections in both cases are the same. However, this is likely due to a reduced strain on disk caches and the hierarchy of memory caches due to the reduced memory usage in the smaller graphs.

Approximation

In a second sub-experiment, we compare the accuracy of k -approximated Decision Graphs with post-pruned Decision Trees. We also compare the time required for learning followed by pruning of Decision Trees with the time required to learn directly to k -approximated Decision Graphs.

The post-pruning in the FC4.5 implementation uses a so-called *reduced error pruning strategy* [63], a modification of the post-pruning algorithm presented in Section 4.4, where internal nodes of a fully grown tree are removed one at a time as long as the error is decreasing.

Our k -approximation² is, in contrast, a very simple process, where we merge the leaves of all subtrees below a certain depth k , cf. Section 4.3. Notice that, although the pruning takes place during the learning process, it is some kind of post-pruning since we merge the leaves of fully grown branches, unlike pre-pruning which can suffer from premature termination of a tree-growing process.

In our experiments, we used the depths of the pruned Decision Trees to decide on the k used in the k -approximations. Figure 7.3 shows the results of the accuracy comparison.

For each dataset (1–16) we have four bars. The first three show: 1) the accuracy (%) of the Decision Tree before pruning (denoted DT), 2) the accuracy of the pruned Decision Tree (denoted PDT), and 3) the accuracy of the k -approximated Decision Graph (denoted KDG). On top of each PDT bar, we show the depth k of the pruned Decision Tree; this is also the depth of the corresponding KDG. Finally, the fourth bar shows the results of an improved k -approximation referred to as KDG2: the depth k is decremented step-by-step as long as the error decreases further. The fourth bar shows the KDG2 accuracy along with the finally reached depth as the number on top of that bar.

First, the results for the two approaches (PDT and KDG) are quite similar. On average, the pruned Decision Trees outperform the k -approximated graphs by only 0.04%. This is a bit surprising given the difference in complexity of the two approaches.

Second, in a number of datasets (1,10,11,12) the depths of the pruned trees remain unchanged. Consequently, no k -approximation is applied and the results for DT and KDG are the same. At least for dataset 10 (waveform), the second approximation strategy leads to improvements: it reduces the size of the Decision Graph considerably and reaches the accuracy of the pruned Decision Tree, cf. PDT vs. KDG2.

Finally, there are cases where KDG and KDG2 are more accurate than PDT (3, 12, 14). For example, dataset 14 (connect-4) indicates that the accuracy of the Decision Tree before pruning was 92%, after pruning 88%, the k -approximated accuracy 92% with $k=19$, and the k -approximated accuracy 91% with $k = 16$. There are also other cases where pruning is not improving accuracy at all and cases where PDT is more accurate than KDG and KDG2.

Regarding the learning and pruning/approximation times, the results are,

²Notice that, it is slightly different approach to k -approximation from the one presented in Section 3.4 where k is attribute index.

Figure 7.3: The accuracy gained by pruning Decision Trees and using k-approximated Decision Graphs

Figure 7.4: Learning and approximation times of Decision Graphs as % of Decision Trees (100%)

again, non-ambiguous. In the time measurements, we have used the same relative metric as in Section 7.1. Figure 7.4 shows that the learning of k -approximated Decision Graphs clearly outperforms the joint procedure of learning and pruning the Decision Trees by up to 50%. On average, the k -approximated approach requires about 21% less time than the tree pruning approach.

7.2 Decision Models in Context-Aware Composition

The theoretical assessment of memory consumption and decision time for decision models was presented in Chapters 5 and 6. Thus, at first we practically assess memory and decision overhead in a number of experiments. Moreover, we compare the accuracy of different learning technologies with Dispatch Tables as the baseline. Finally, we show the speedups of context-aware composition using different decision function implementations. Decision Graphs and Dispatch Tables provide the lowest overheads compared to other decision models. Thus, we assess the overall performance of context-aware composition using Decision Graphs vs. Dispatch Tables.

Experimental Setup

Algorithm Variants: We implemented the well-known sorting algorithms Selection sort, Quicksort, and Merge sort along with two parallel versions of Quicksort and Merge sort, which spawn a new thread for one of the sub-

problems in each divide step. We used textbook implementations [12] and did not optimize the variants. In particular, we greedily create new threads regardless of the number of available cores. All algorithms are implemented in Java/JDK 1.6.

Platforms: All practical experiments are executed on three different multi-core machines:

M1 a 2 core Dell Latitude PC running Windox XP (2002, SP 3) on an Intel Dual Core T2300 at 1.66GHz and 2GB RAM,

M2 an 8 core IBM Blade Server running Linux (CentOS) on an Intel 8 Core 5450 at 3GHz and 16GB RAM, and

M3 a 2 core MacBook Pro running Mac OS X (10.6.5) on an Intel Core i5 at 2.4 Ghz and 8GB RAM.

All tests run on the respective native JVMs with `-Xms128m -Xmx1g` as the virtual machine parameters.

Memory Overhead

To compare different classification approaches, we constructed three Dispatch (decision) Tables for our sorting problem for different multi-core machines M1, M2, M3. The dispatch technique attempts to speed up sorting by selecting the best algorithm (class) Y for the current context (problem size N , processor availability P), where N is a continuous integer sampled at powers of two between $2^0 \dots 2^{16}$, P is boolean with 0 and 1 as possible values (encoding whether or not processors are available). Y is a discrete integer in the range $1 \dots 5$, each representing one of the algorithm variants. The memory needed for storing $34 (2 \times 17)$ entries is rather small, i.e., $M = 4 \times 34 = 136$ Bytes, cf. Table 7.2 for the size of the Dispatch Table and the alternative classifiers constructed.

To encode a Dispatch Table in a Decision Tree or a Decision Graph, we used the FC4.5 learning algorithm. Decision Trees have a moderate learning effort. The memory compression rate is quite high: $\approx 82\%$ for tables M1, M2 (the number of *edges* equals 6, requiring 24 Bytes) and $\approx 71\%$ for M3 (*edges* = 10, 40 Bytes).

The Decision Graphs are a redundancy-free encoding of the Decision Trees and, hence, the depth does not change. However, Dispatch Table compression does not improve compared to Decision Trees (the number of edges is the same). In our example, the graphs use up to 27% fewer nodes than the tree.

The Naive Bayes classifier was the fastest to construct and the encoded Dispatch Tables take only 80 Bytes giving a 41% of reduction immediately

	Table in Bytes	Tree in Bytes	Graph in Bytes	Bayes in Bytes	SVM in Bytes
Classifier size	136	24/40	24/40	80	544

Table 7.2: Memory overhead of different classification models.

	Table # operations	Tree # operations	Graph # operations	Bayes # operations	SVM # operations
Flops	23	0	0	46	644
Array access	1	2.5	2.5	5	140

Table 7.3: Decision overhead of different classification models.

without any additional optimizations, since the size of the classifier only depends on the context attributes, not their values.

In order to construct an accurate prediction model based on Support Vector Machines, a learning phase requires an optimization of kernel function parameters specific to each problem domain and input sample data. Although done automatically, finding parameters giving an acceptable decision accuracy requires a cross-validation pre-learning over the large range of parameter values. In our example, (pre-)learning time was still acceptable in range of a few seconds. The memory required to encode the SVM classifier is 544 Bytes (based on the LIBSVM library implementation). This is the highest memory overhead and even 75% larger than Dispatch Table memory consumption.

Decision Accuracy

In this section, we compare the accuracy of the different decision approaches. As we do not know the right decision for each actual context (processors available and problem size), but only for the sample points measured in the training phase, we can assess accuracy only approximatively: (1) by comparing the decisions of the different approaches at these sample points, and (2) by comparing their decisions with the decisions of the table approach as the baseline. We define a decision error as (1) the ratio of decisions not suggesting the best algorithm variant in the sample points over all decisions, and (2) the ratio of decisions diverging from the suggestion of the decision table over all decisions. The Dispatch Table captures the best implementation for the sample points. Hence, its error is 0 in the measure (1). However, we do not know if the Dispatch Table suggests the best variant between sampled problem sizes. Hence, (2) is an accuracy measure relative to the baseline implementation of Context-Aware Composition using Dispatch Tables.

For the different platforms, Table 7.5 shows the error (1) at the sample points and the error (2) for selected sizes of arrays to sort. All ratios are given in %.

Dispatch Tables (trivially), Decision Trees, Decision Graphs, and Support Vector Machines (cf. rows "Sample points") are to 100% accurate according to measure (1) and, generally, all decision approaches perform very accurate at the sample points with an error of at most 15%.

The error is somewhat higher for the accuracy measure (2). Independent of the problem size, Decision Trees and Graphs suggest in 21% to 25% of the cases different algorithm variants than the Dispatch Table (cf. columns "Tree" and "Graph"). For the Bayesian classifier and the classifier based on Support Vector Machines, the error (2) increases with the problem size. However, it is in most cases smaller than for Decision Trees and Graphs. Note that the error of 100% for the Bayesian classifier for the very large problems is due

Platform	Problem	Tree in %	Graph in %	Bayes in %	SVM in %
PC 2 cores	Sample points	0	0	15	0
	10,000	21	21	0	10
	100,000	21	21	4	10
	1,000,000	21	21	100	10
Server 8 cores	Sample points	0	0	9	0
	10,000	25	25	0	0
	100,000	25	25	1	0
	1,000,000	25	5	100	25
MAC 2 cores	Sample points	0	0	6	0
	10,000	21	21	0	9
	100,000	21	21	3	9
	1,000,000	21	21	100	9

Table 7.4: Errors of different decision approaches.

to its first (wrong) decision in favor of Selection sort. As Selection sort does not contain any recursive calls, there are no other decision points, and this wrong decision is the only one.

As a conclusion, none of the approaches ought to be dropped from further evaluations since their accuracy is comparable with the accuracy of decision tables.

Decision Overhead

In this sub-section, we compare the overhead of the different approaches for decision making. We compare them first based on our theoretical assumptions and then experimentally.

For the 2-dimensional Dispatch Table based on one continuous and one discrete attribute, the prediction time includes 23 floating point operations and one array access, that is $T_{flops} \times 23 + T_{aa}$. It is worth mentioning that in our cost model a look-up for any 2-dimensional Dispatch Table is constant and does not depend its number of entries, which is a simplification ignoring caching effects. Table 7.3 shows the floating point operations and the number of array accesses for the classifiers.

The decision time required by Decision Trees and Graphs depends on the trees and diagrams depth, resp. For each of the three Decision Trees (graphs) constructed, the decision requires at most three indirect array accesses, since the maximum depth is three for all trees (the average depth is 2.5). Thus, the expected overhead can be estimated as $T_{aa} \times 2.5$.

As discussed, a Naive Bayes classifier is quite effective in terms of memory

consumption, but on the downside, it has quite a high runtime overhead for making decisions. In our setting, it requires 46 floating point operations and 5 array accesses, $T_{flops} \times 46 + T_{aa} \times 5$. This is a factor of two when it comes to flops and a factor of five when it comes to array accesses, compared to Dispatch Tables, which makes it a slow classifier for our problem.

A decisions in SVM requires 21 flops that correspond to a kernel function computation calculated for each of the $l = 28$ support vectors and then additional 56 flops for classification. Altogether, the decision phase takes 644 flops and the whole decision time is $T_{flops} \times 644 + T_{aa} \times 140$. This is the highest look-up overhead among all classification models.

Experiments confirm these overhead predictions. For these experiments, we use the fastest homogeneous solution – Quicksort – as the baseline. The third column of Table 7.5 shows the time for Quicksort for the different platforms and three selected array sizes to sort.

On each recursive invocation of Quicksort (with a certain number of processors still available and a certain sub-problem size), we look up the best algorithm variant in the different decision repositories. However, in order to compare the decision overhead in a fair way, we always invoke Quicksort regardless of the best algorithm variant suggested. Hence, all compared solutions essentially implement Quicksort.

Table 7.5 shows the execution times of the Context-Aware Composition based on the different decision functions relative to the execution time of Quicksort on the same platform and architecture. For instance, the Dispatch Table (cf. column "Table") introduces an overhead of 43% on the PC with 2 cores and the problem size of 10,000 array elements, as it requires $1.43 \times 3.62\text{msec} = 5.18\text{msec}$ of the corresponding Quicksort execution times without table lookup (3.62msec).

There are $O(N^2)$ expected lookups for problems of size N in Quicksort; the expected work of Quicksort is $O(N \log N)$. The lookup time is $O(1)$; it only depends on the number of algorithm variants, the number of context attributes, and the decision repository variant, but it does not grow with the problem size. Hence, for all decision repository variants, the overhead gets smaller with increasing problem size. This is confirmed by our measurements. The Decision Graph (column "Graph") introduces the lowest overhead in almost all cases, with the Decision Tree (column "Tree") not far behind. These overheads are between 6% and 36% depending on problem size and platform. The Dispatch Table comes with an overhead between 15% and 55%. Algorithm variant selection using Naive Bayes classifiers and Support Vector Machines (columns "Bayes" and "SVM", respectively) slows down the execution by factors between 2.4 and almost 42.

Quicksort (and other recursive sorting algorithms) are extreme in the sense that the ratio between decision points (recursive calls) and workload is rather high. Therefore, we observed a rather high overhead for context-

Platform	Problem size	Quicksort in msec.	Table in %	Tree in %	Graph in %	Bayes in %	SVM in %
PC 2 cores (M1)	10,000	3.62	43	25	22	1161	4099
	100,000	51.91	28	14	10	816	2871
	1,000,000	998.36	15	8	6	423	1449
Server 8 cores (M2)	10,000	1.87	52	34	34	680	1595
	100,000	22.93	55	29	30	553	1321
	1,000,000	552.51	24	12	12	243	576
MAC 2 cores (M3)	10,000	1.99	37	36	33	328	1008
	100,000	28.08	29	19	18	224	707
	1,000,000	478.55	25	16	15	139	426

Table 7.5: Time overhead (in %) of different decision approaches.

7.2. Decision Models in Context-Aware Composition

(a) PC 2 cores.

(b) Server 8 cores.

(c) MAC 2 cores.

Figure 7.5: Homogeneous Quicksort and Context-Aware Sorting using Decision Graphs ("Opt Graph") and Dispatch Tables ("Opt Table"). The x -axis displays the array size, the y -axis the time in $msec$.

aware composition compared to the homogeneous Quicksort variant. In conclusion, Dispatch Tables, Trees, and Graphs introduce an overhead that is still acceptable as it can be expected that a clever algorithm variant selection compensates for the overhead. At least for recursive sorting problems, this is not the case for Naive Bayes classifier and Support Vector Machines.

As conclusion, we disregard Naive Bayes classifier and Support Vector Machines as decision repositories for improving sorting using context-aware composition due to their (too) high decision overhead.

As Decision Trees and Graphs by construction always suggest the same variant, and Decision Graphs are smaller in size and have a slightly smaller overhead, we disregard Decision Trees in the final overall assessment.

Overall Performance

Now we are ready to assess the overall performance of context-aware composition using Decision Graphs vs. Dispatch Tables. Figure 7.5 shows the experimental results on the different platforms. As a reference, it also shows how the fastest homogeneous implementation variant (sequential Quicksort) performs on the three platforms.

On the PC with two cores M1, the optimized version using Decision Graphs gains a speed-up of 1.47 over sequential Quicksort, on average over all array sizes from 10,000 – 1,000,000 (step 10,000), while the optimized version using Dispatch Tables gains a speed-up of 1.46. On the server with 8 cores M2, the difference between the two implementations is even more pronounced: an average speed-up of 1.92 for the graph-based solution vs. 1.79 for the table-based solution. On the MAC with 2 cores M3, the speed-up results are 1.39 vs. 1.37, again in favor of the graph-based solution.

Obviously, the decisions of the Dispatch Table are (slightly) more accurate, but this is more than compensated by the lower runtime overhead of the Decision Graphs. Altogether, the experiments showed that the Decision Graph is not only smaller by a factor of five than the Dispatch Table, but also (slightly) faster when used as a decision repository in Context-Aware Composition.

7.3 Summary

In the first part of this chapter we compared Decision Graphs and Decision Trees as alternative decision models. On the practical side, our experiments showed benefits of Decision Graphs compared to Decision Trees regarding memory resource and time utilization as detailed below.

First, the experiments show that, without losing accuracy, Decision Graphs reduce the memory consumption of Decision Trees by 44% (on average over a

number of standard datasets). The reduction is mostly caused by the elimination of redundant leaves, but replicated and redundant decision (sub-)trees also contribute to the memory reduction. Redundancy (and with it the potential saving of our redundancy elimination) increases as Decision Graphs avoid keeping additional information in internal nodes. Such internal information can be different for different (otherwise identical) subtrees. The absence of such information does not influence the classification accuracy, as proved by our experiments. Also worth mentioning is that the reduction appears to grow with the number of categorical attributes and with the size of (number of instances in) the dataset.

Second, simple k-approximated Decision Graphs and pruned Decision Trees have almost equivalent accuracy. As opposed to, e.g., error-based post-pruning, the k-approximated Decision Graphs do not apply any complex statistical calculations in the leaves and simply merge classes in a fully grown tree branch. This means that using approximated Decision Graphs allows avoiding additional costly post-pruning.

Third, the time for Decision Graphs construction shows a decrease by 19% compared to the time for Decision Tree construction. Furthermore, the time measured for classification using Decision Graphs was 20% less than in the corresponding Decision Trees. This is, likely, the result of less strain on caches due to the memory reduction in Decision Graphs. The time for learning followed by pruning the Decision Tree compared to learning directly to the k-approximated Decision Graphs decreased by around 21%. This result supports our statement about avoiding post-pruning operations mentioned above. Finally, the second part of this chapter contributes with:

- A platform for plugging classifiers into Context-Aware Composition. It was instantiated with 5 classifier variants: Dispatch Tables (baseline), Decision Trees, Decision Graphs, Naive Bayes classifier and SVMs.
- A practical assessment on three different hardware platforms using sorting as a running context-aware composition example.

Decision Graphs introduced the lowest overhead in almost all cases for our sorting problem (different problem sizes and hardware platforms). These overheads are between 6% and 36% depending on problem size and platform. On the other hand, SVM and Naive Bayes classifier slow down the execution by factors between 2.4 and almost 42 due to the high decision overhead. For the sorting problem in context-aware composition, Decision Graphs reduced the memory consumption by a factor of five and increased overall performance by a few percent. Therefore, this experiment also suggests Decision Graphs to be a preferred implementation of decision functions.

Chapter 8

Conclusions

In this chapter, we summarize the findings from this thesis and relate them to the goals and the goal criteria presented in Chapter 1.

This thesis explores the common theory behind capturing and manipulating decision information that can be used to deduce a relationship between a context and a decision. This type of information is usually represented by decision models using different learning strategies, having different memory requirements, and employing different operations. Chapter 2 introduces the set of basic notions that characterize decision information used in different application domains. Decision information is denoted as a decision relation which, in turn, consists of a set of decision tuples. We denote a decision function as a complete and non-contradictive decision relation and a decision model as a representation of a decision function.

Moreover, we present a number of application domains, such as classification, points-to analysis, and context-aware composition, as examples of the domains in Computer Science fields that process different types of decision relations. Every domain is presented with the type of the decision model used, the type of the operations enrolled for manipulating decision relations, and non-functional requirements. For instance, points-to analysis manipulates reference information where the representation of this information is required to be precise. This requirement is fulfilled by employing the χ -terms decision model. On the other hand, the classification domain can be less strict in terms of precision of the used decision model. Thus, it requires decision interpolation, which is achieved by using a special learning algorithm that constructs an appropriate decision model, e.g., Decision Trees. The type of the used decision model in the context-aware composition domain depends on application requirements, but, nevertheless, precision of a model is usually a substantial parameter. Therefore, Dispatch Table is a most often used decision model in the context-aware composition domain.

Even though all presented domains operate on decision relations, they use different decision models implemented by different data structures due to differences in non-functional requirements. However, because of the variety of solutions and modifications of decision models, the variety of application domains manipulating decision relations (each coming with dif-

ferent notations and tailored implementations), we consider it worthwhile to introduce a general decision function algebra corresponding to any type of decision relation which can be implemented by different decision models along with a set of defined operations. This leads to a definition of a theoretical framework that we denote as the Decision Algebra.

In Chapter 3 we present the Decision Algebra that is based on higher order decision functions based on distributions. We formally define the notion of a decision function along with a set of properties. Construction of decision functions usually employs a learning algorithm that interpolates a given decision relation. Another approach to construct decision functions corresponds to a simple capturing of decision relations using a specific decision model.

We separate the Decision Algebra operations over decision functions into core and non-core operations. The core operations are *apply* and *evert*, and non-core operations are those that can be derived from them. The *apply* operation is an operation over decision functions that applies a general function to the terms of decision functions. *Evert* changes the order of decision terms in a given decision function. Non-core operations are *decide*, *merge*, and *approximate*. *Decide* corresponds to applying the *mode* function, which computes a final class over the set of distributions, on the results of decision functions. *Merge* is an operation that combines several decision functions into a new one using the *apply* operation that applies the $\sqcup$ operation on initial decision functions. Finally, the *approximate* operation corresponds to the approximation of a given decision function by replacing this function with the merger of its sub-functions. Thus, all these operations are derived from the *apply* operation.

Chapter 4 shows that existing algorithms used in the application domains presented in Chapter 2 can be implemented based on the Decision Algebra operations. As examples of the algorithms we take two types of the learning algorithms: C4.5, which interpolates decision relations from the classification domain, and a simple algorithm based on a capturing of decision relations from the points-to analysis and context-aware composition domains. Additionally, we take two pruning algorithms known from the classification domain and the so-called *k-approximation* algorithm from the points-to analysis domain. Having the implementation of these algorithms separated from special data structures gives the advantage of exchanging these algorithms between different decision models and, therefore, between different domains. For instance, *k-approximation* can be reused by Decision Trees or Dispatch Tables, learning strategies can be applied to construct different decision models, etc.

Chapter 5 presents three existing decision models based on distributions: Decision Trees, Decision Graphs, and Dispatch Tables, with concrete implementations of the Decision Algebra core operations. Decision Trees are used

as the decision model in the classification domain, Decision Graphs are the explicit Decision Algebra implementation that corresponds to the χ -terms decision model in the points-to analysis domain, and Dispatch Tables are widely used in the context-aware composition domain. Each decision model has its own implementation of the *apply* and *evert* operations. Additionally, we analyze the non-functional properties for each decision model in terms of memory consumption and decision time. Based on these examples we show that the core operations can be implemented in each decision model under the common interface provided by the Decision Algebra. The overhead introduced by implementing these operations under the Decision Algebra interface instead of their direct implementation on the decision models has a constant time complexity.

There exist other types of decision models originated in the fields of Data Mining and Machine Learning, which we are not able to represent as distribution-based decision models. In Chapter 6 we show that their differences mainly consists of the decision representation that decision functions capture from a given decision relation during a construction process. That is, instead of distributions representation, decision functions can have a general decision representation $X(C)$ over decisions C . This leads to a generalization of the notion of decision function and, consequently, signatures of the operations on decision functions. As a result of such a generalization, we get a new multi-sorted Decision Algebra, denoted as Abstracted Decision Algebra, that even describes decision functions represented by decision models different from distribution-based decision models. Thus, the Decision Algebra presented in Chapter 3 becomes a quotient algebra over the Abstracted Decision Algebra. Moreover, in this chapter we discuss two existing decision models: Naive Bayes classifier and Support Vector Machines. For each decision model, we determine the concrete decision representation $X(C)$ along with the set of operations that follow the signatures of the new Abstracted Decision Algebra.

Since most Decision Algebra operations, namely all non-core operations, can be defined on the general decision function level (regardless of their implementations), comparing the decision models is rather fair showing advantages and disadvantages of the decision models instead of advantages and disadvantages of the different implementations thereof. This is shown in the practical part of this thesis, Chapter 7, where two experiment series show the benefits of using a specific Decision Algebra implementation, Decision Graphs, over four other decision models that were introduced in Chapters 5 and 6. Practical problems that we stumbled across when comparing different instances of the Decision Algebra as well as our solutions and workarounds to them are also discussed at this point. We divide the experimental part into two experiment series: (1) the first experiment series compares Decision Trees with Decision Graphs regarding the memory consumption, the learn-

ing and deciding speed, and the accuracy; (2) the second experiment series compares the decision and memory overheads of five decision models as alternative implementations of the Abstracted Decision Algebra capturing decision relations in the context-aware composition domain.

The first experiment series shows that Decision Graphs outperform Decision Trees in all measurements including speed for learning and deciding, as well as memory consumption: Decision Graphs gain learning and classification speedup up to 20% without accuracy loss, and reduce memory consumption by 44%. The result of the second experiment series shows that using Decision Graphs in context-aware composition leads to better speed and memory scalability of the context-aware composition domain compared to Dispatch Tables as the baseline decision model: Decision Graphs introduce the lowest decision overhead of 6% - 34% over all other decision function implementations and reduce the memory consumption by factor of five. This means that the context-aware composition approach can be applied at more program points and include more context attributes than before.

This thesis shows that the Decision Algebra as a generalized theoretical framework for decision information can be applied to formalize classical approaches in Data Mining and Software Engineering fields, cf. Chapter 2, addressing the problems of manipulating and capturing decision information as well as the typical classification problems of fragmentation, replication, and model overfitting. In fact, classical decision models used in Data Mining and Machine Learning (Decision Trees, Decision Graphs, Dispatch Tables, Naive Bayes classifier and Support Vector Machines) and variants thereof as exploited in special learning and classification algorithms can be understood as instances of the Decision Algebra by varying the implementation of the Decision Algebra operations as presented in Chapters 5 and 6. Having the general representation of decision information can lead to further insights in order to select an appropriate decision model, even though it is always a tradeoff between several performance metrics. Additionally, having a defined set of operations that are able to propagate to any type of decision models can lead to developing a new set of algorithms or improving a performance of decision models, i.e., increasing decision accuracy, reducing memory and decision overheads, etc.

8.1 *Review of the Goals and Goal Criteria*

We now review the results of this thesis with respect to the goals we have set in Chapter 1, which are:

1. Theory: Develop a unifying theoretical framework for capturing and manipulating decision information. We refer to it as Decision Algebra.

2. **Implementation:** Develop a platform that allows comparing different instantiations of the framework, i.e., decision models implementing the Decision Algebra, and employing these decision models as alternative variants in applications from different domains where processing of decision information is essential.

Goal 1

The goal criteria for our first goal are:

- 1.1 The Decision Algebra shall be complete: it shall provide a complete set of operations which is used by known decision models like Decision Trees, Decision Graphs, Decision Tables, Naive Bayes classifier, Support Vector Machines, and others for decision information processing. The complete set of operations shall be specified based on the observations made in concrete problem domains.
- 1.2 The Decision Algebra shall be efficient in terms of time complexity: the implementation of existing decision models as Decision Algebra instantiations shall not impose any complexity overhead compared to if they were implemented as stand-alone models. That is, no algorithm using Decision Algebra operations shall have a higher time complexity than if it was directly using specific decision model operations.
- 1.3 The Decision Algebra shall be efficient in implementation: it shall reduce the number of operations to a few core operations that are able to describe known decision models. This makes the Decision Algebra more reusable as it reduces the work required to implement a new decision model using this set of core operations.

Criterion 1 is fulfilled: the set of operations defined in the Decision Algebra represents the basic operations required by the decision models, discussed in Chapters 5 and 6, to process decision relations. The necessity of these particular operations is assessed in Chapter 2 for the classification, points-to analysis, and context-aware composition domains, and in Chapter 6 where the main principles of construction and application of Naive Bayes classifier and Support Vector Machines are discussed.

Criterion 2 is fulfilled: the discussed decision models, such as Decision Trees, Decision Graphs, Dispatch Tables, Naive Bayes classifier and Support Vector Machines are interpreted as instantiations of the Decision Algebra in Chapters 5 and 6. This implies that the operations of the Decision Algebra have to be inherited by each specific decision model. This does not impose any additional overhead in the time complexity of the existing operations of the decision models compared to if these operations were implemented independently of the Decision Algebra signature. Moreover, a set of algorithms

for learning, pruning, and approximating decision functions, discussed in Chapter 4, is presented based on the Decision Algebra core operations. These algorithm representations in terms of the core operations do not make algorithms more complex compared to if they were implemented directly using specific decision models operations. Therefore, the overhead in terms of the time complexity introduced by implementing the decision models as the Decision Algebra instantiations and by implementing the algorithm based on the Decision Algebra operations is constant.

Criterion 3 is fulfilled: Chapters 3 and 6 reduce the complete set of the Decision Algebra operations to the set of core operations including only the *evert* and *apply* operations. In Section 3.4 we show that the non-core operations such as *decide*, *merge*, and *approximate* are derived from the *apply* operation, and *evert* can serve as the basic operation for different learning strategies. Additionally, Chapter 5 and Chapter 6 show how existing decision models can be represented as instances of the Decision Algebra by interpreting or implementing this set of core operations. The non-core operations have default implementations in terms of the core operations. However, dedicated implementations of the non-core operations can be found directly in decision models, since operation interpretations can differ from the one presented by the Decision Algebra. These implementations can have the same performance characteristics but be more advantageous in terms of other properties, e.g., accuracy, required memory, etc.

Goal 2

The goal criteria for our second goal are as following:

- 2.1 It shall be possible to compare different decision models under a common Decision Algebra interface regardless of implementation details.
- 2.2 It shall be possible to improve performance in one domain by using a decision model introduced by another application domain.
- 2.3 It shall be possible to assess non-functional properties for each decision model based on platform validation and deduce different performance profiles.

Criterion 1 is fulfilled: In Section 7.1 we compare two decision models as the Decision Algebra instances representing the distribution-based decision functions - Decision Graphs (a variant of the χ -terms decision model) and Decision Trees as a widely used decision model in the classification domain. The construction of decision functions is based on the C4.5 learning algorithm that produced two different decision models, which gives the advantage of being able to compare them independently from their specific implementation details. The comparison is done based on the number of

well-known metrics used for evaluation of classification (prediction) models in the classification domain.

Criterion 2 is fulfilled: in Section 7.2 we demonstrate how four decision models, Decision Trees, Decision Graphs, Naive Bayes classifier and Support Vector Machines, can be plugged into the context-aware composition domain for capturing the best composition contexts, as alternatives to the previously used Dispatch Table model. Moreover, we compared the accuracy of the different learning algorithms with the Dispatch Tables capturing algorithm as a baseline and finally show that the performance is improved in terms of the memory and speed gained by the context-aware composition domain using the decision models introduced by the points-to analysis and classification domains.

Criterion 3 is fulfilled: Chapters 5 and 6 give a theoretical assessment of the memory consumption and decision time for the five alternative Decision Algebra instantiations corresponding to the discussed decision models. The practical assessment of their non-functional properties is presented and discussed in Section 7.2, where these properties are evaluated in the context-aware composition domain and different performance profiles are shown.

In conclusion, we see that the goals we set for this thesis are fulfilled.

Chapter 9

Future Work

In this chapter, we present future work towards improving our theoretical framework presented in Chapters 3 and 5, and improving our Decision Algebra implementation presented in Chapter 7. Additionally, we have to consider new challenges in applying Machine Learning and Data Mining approaches in Software Engineering tasks where programs must dynamically adapt to changing conditions.

9.1 Improving Theoretical and Practical Aspects of the Decision Algebra

The Decision Algebra is only the first attempt towards a unifying theory for capturing and manipulating decision information, and our explicit implementation of decision functions, Decision Graphs, is still a prototype. Quite a few theoretical and implementation aspects should be considered in the future. This includes theoretical improvements of the Abstracted Decision Algebra, where existing decision models used for classification purposes in Data Mining and Machine Learning have to be described in terms of quotient algebras. That is, in addition to distribution-based decision functions defined as multi-sorted quotient Decision Algebras, implemented by a limited number of decision models, we have to generalize other types of decision models (classifiers), such as probability classifiers, linear classifiers, kernel machines classifiers, etc., to sets of specific types of decision functions. Each such set along with the set of defined operations corresponding to the algebra signature will introduce a new quotient algebra constructed from the Abstracted Decision Algebra. In this way, in addition to the distribution-based Decision Algebra, we will obtain other types of algebras where the carrier sets will be based on different co-domains. Moreover, the set of Abstracted Decision Algebra instances has to be extended by new decision models, such as Bayesian Network which is a probability-based decision model, Neural Networks which is a kernel machines model, etc.

The *merge* operator presented in Section 3.4, which combines decision functions from same application domains and thereby performs an iterative

learning, seeks theoretical and implementation improvements in terms of decision accuracy: by combining two decision functions, the accuracy shall be increased or, at least, not drop dramatically. The function that has to be applied on decision functions during the *merging* operation has to be defined in each decision model. For instance, in addition to the *add* operation used for merging Decision Trees and Decision Graphs, we have to define a corresponding operation for Support Vector Machines.

Moreover, additional algorithms manipulating different decision models in the classification domain have to be explored, and their implementations have to be defined based on the Decision Algebra operations.

Theoretical and practical modifications of the learning process of decision functions have to be done in order to enable efficient online learning of decision functions with high accuracy. For now, we explored only Eager learners that, given a set of tuples, construct a generalization model before receiving new tuples to classify. However, in order to enable a different type of learning that is referred to as *incremental learning*, we have to consider Lazy learners, where a given training tuple is simply stored, and a model generalization is only performed when a new test tuple comes for classification. However, classifying on Lazy learners can be computationally expensive. Lazy learners are well-suited for implementation on a parallel hardware and usually require efficient storage techniques. Therefore, we can try to use our Decision Graphs as a data structure to represent Lazy learners and investigate how the Decision Algebra operations map to these types of decision models.

New instances of the Decision Algebra have to be implemented in our platform and compared in terms of their memory consumption, learning and decision time. In addition to the existing operations and used algorithms, our platform has to implement the *merge* and *approximate* operations, and the accuracy of computed decision functions has to be evaluated. We also have to validate the algorithms (prune, k-approximate) when applied to different decision models, e.g., k-approximate to Decision Trees or pruning to Decision Graphs.

9.2 Machine Learning in Software Engineering

Machine Learning algorithms are used in many different problem domains. One of the traditional domains is Data Mining, where large databases contain valuable implicit regularities that can be discovered automatically. Moreover, there is a number of poorly understood application domains with a lack of knowledge needed to develop effective algorithms [87].

Not surprisingly, the field of Software Engineering turns out to be a fertile ground where many software development and maintenance tasks

can be formulated as learning problems and approached in terms of different learning techniques. Examples of such Software Engineering domains in this thesis are: points-to analysis and context-aware composition.

In order to use Machine Learning methods as a tool for solving Software Engineering problems, we need to have a clear understanding of both problems and methods. It is imperative that we have to know (1) available Machine Learning methods at our disposal, (2) characteristics of those methods, (3) circumstances under which the methods can be most effectively applied, and (4) their theoretical underpinnings.

As our first step towards applying Machine Learning strategies to Software Engineering problems we have considered the context-aware composition domain, where we tried to improve the application performance by selecting the best algorithm variant for certain runtime conditions. We used the sorting problem as a problem to solve in this domain. Sorting is an extreme problem in the sense that the ratio of dynamic decisions and payload operations is rather high. Therefore, as a matter of future work, we need to extend our experimental basis to problems with a lower rate of decisions and payload, which possibly allows for a higher overhead in decision making. In these scenarios, speed-ups could be achieved using Naive Bayes classifier and SVM benefiting from their higher decision accuracy despite their high decision overhead.

Also, composition based on more context attributes (e.g., including data structure implementations of arguments and results) ought to be considered. This leads to larger Dispatch Tables and allows validating the memory reduction of Decision Graphs and their alternatives. Moreover, we need to reevaluate our findings in online learning scenarios as required in self-adaptive systems. Here, we would change the system environment, e.g., force energy saving mode or generate extra workload outside the program's control, which invalidates the learned best-fit variants. Execution of suboptimal variants detects such changed system environments. In these scenarios, learning time becomes crucial as it adds to the overhead of context-aware composition.

We used the different learning technology (almost) as black boxes, i.e., we did not adapt them to our specific application context. Doing so could improve the overall speed-up (and the performance of the different technologies relative to each other) and further reduce memory consumption. For instance, we could capture Decision Graphs in a very compact array containing indices (inner nodes) and variants (leaves) or split the SVM classifier in several classification steps, each possibly requiring only a linear kernel function making them both faster and more compact and, hence, competitive again, etc.

Finally, more Software Engineering problems benefiting from applying Machine Learning approaches have to be discovered in the future. Thus,

our platform has to be used to plug different decision models processing a certain type of decision information into more application domains.

Appendix A

Context-Sensitive Strong Updates in Points-to Analysis

In Section 2.3 we considered points-to analyses that always perform weak updates, i.e., that use incremental updates of all values. However, most of the precision benefits of flow-sensitivity comes from strong updates since the execution order of statements is known. A major theoretical obstacle in a flow-sensitive analysis is the question when it is safe to perform *strong updates*. A strong update occurs when an assignment supersedes (or kills) all previous assignments. Although strong updates obviously give a more precise analysis, they are much more difficult to apply correctly. The problem is that if a points-to set of a dereferenced variable contains several abstract objects, the analysis cannot guarantee that anyone of the target in the set is definitely overwritten. Hence, strong updates are only permitted if the dereferenced may-points-to set is a singleton and it is known that a new value completely overwrites the previous value of a given pointer[42].

For fully field-flow-sensitive analysis the strong updates have to be performed on heap array elements. However, a strong update can be applied at a store statement $v.f = v_1$ only under three conditions [61, 82]:

- v points to a single abstract heap object that represents *at most* one concrete object at a runtime. That is, the method in which the abstract object is allocated is not a part of any loop or recursive call chain, where an object can be allocated more than once.
- f represents a single location in that object. This condition is satisfied unless f is an array of elements.
- the method where the strong update can be performed is not part of parallelization, that is, we can guarantee that the points-to values of a object field will not be updated in parallel executions.

Until this point we saw how the employment of χ -terms for points-to values representation can reduce the number of available abstract objects for the current point. The question that we discuss in this appendix is whether or not χ -terms can be used to perform strong updates.

Figure A.1: Code example and corresponding control-flow graph

In what follows we briefly outline an idea of how strong updates can be performed in points-to analysis. However, this is not a complete study, and the following restrictions have to be applied: we restrict the field updates to the fields of local variables represented by abstract objects that reached the current method f through arguments, including *this* value. Thus, we exclude fields such as $a.f.h\dots$, etc. Thereby, we make sure that these variables represent concrete objects at runtime, since they all reached method f . We also exclude fields that correspond to arrays of elements. Additionally, any method $foo()$ is not a part of any recursive call, it can not be executed as part of program parallelization, and it is not called from a branch of a control-flow option. Following the definition of points-to contexts presented in Section 2.3, each method in a program has its own context stack that keeps track of the method's current context.

Let us consider a code example in Figure A.1. Let object field f be a field of *this* with *this* being a singleton object. In this example the points-to value of b at point P_3 could be even more precise ($pt(b) = o^9$), since abstract object o^2 should be overwritten by the *if-then-else* control statement.

The *merge* operation used previously for constructing new χ -terms for points-to set values is not applicable for the purpose of strong updates, since it keeps all the possible values (performs weak updates) by applying the joint operation on the points-to values corresponding to leaves of χ -terms.

In order to perform *update* we have to introduce a new operation on points-to set values, denoted as *context-sensitive update* $\rightarrow$, that will not merge

all the possible values, but will update them with new values. This operation should be applied on the χ -terms representing the points-to values in the same way as the $\sqcup$ operator was applied in order to construct a resulting χ -term in Section 2.3.

Definition 12. Let $v_1 \in V$ and $v_2 \in V$ be the points-to values contained in the leaves of χ -terms t_1 and t_2 . The context-sensitive update operation $\tilde{\rightarrow}$ is a direction-sensitive operation, where the direction of an arrow denotes which points-to values have to be updated. The $\tilde{\rightarrow}$ -join of v_1 and v_2 is a points-to value where:

$$\begin{aligned} &\text{if } v_1 \neq \perp \Rightarrow v_1 \tilde{\rightarrow} v_2 = v_1 \wedge, \text{ if } v_1 = \perp \Rightarrow v_1 \tilde{\rightarrow} v_2 = v_2 \\ &\text{if } v_2 \neq \perp \Rightarrow v_1 \tilde{\leftarrow} v_2 = v_2 \wedge, \text{ if } v_2 = \perp \Rightarrow v_1 \tilde{\leftarrow} v_2 = v_1 \end{aligned}$$

For instance,

$$\chi^1(\perp, \chi^2(8, \perp)) \tilde{\rightarrow} \chi^1(5, 6) = \chi^1(5, \chi^2(8, 6))$$

Let us come back to the previous code example from Figure A.1 and perform analysis in a similar way as we did before; the only difference we make is that every time when a new points-to value has to be assigned to an object field f we apply $\tilde{\rightarrow}$ instead of applying $\sqcup$.

In what follows we will show the points-to sets after each store operation on f . Notice that stack manipulation is the same as it was presented in points-to analysis performing weak updates.

- $f = \text{new } A(1)$ (line 2): $cx = (0, 0)$

$$pt(f) = \{o^2\} \tilde{\rightarrow} \perp = \{o^2\};$$

- $f = \text{new } A(2)$ (line 4): $cx = (1, 1)$

$$pt(f) = \chi^1(o^4, \perp) \tilde{\rightarrow} o^2 = \chi^1(o^4, o^2);$$

- $f = \text{new } C(1)$ (line 7): $cx = (1, 2), (2, 1)$

$$pt(f) = \chi^1(\perp, \chi^2(o^7, \perp)) \tilde{\rightarrow} \chi^1(o^4, o^2) = \chi^1(o^4, \chi^2(o^7, o^2));$$

- $f = \text{new } B(3)$ (line 9): $cx = (1, 2)(2, 2)$

$$pt(f) = \chi^1(\perp, \chi^2(\perp, o^9)) \tilde{\rightarrow} \chi^1(o^4, \chi^2(o^7, o^2)) = \chi^1(o^4, \chi^2(o^7, o^9)).$$

Considering these results, the points-to value of f at point P_2 is:

$$pt(f) = \chi^1(o^4, \chi^2(o^7, o^9))$$

Figure A.2: A simple control-flow graph representation for loop

Therefore, by restricting this value to the current context $cx = [(1, 2), (2, 2)]$ of program point P_2 , we get a more precise value for b :

$$pt(b) = \chi^1(o^4, \chi^2(o^7, o^9))|_{(1,2)} = \chi^2(o^7, o^9)|_{(2,2)} = o^9.$$

Notice, it is now equal to the runtime execution value.

Using this operation we can update the values not only to one new object but to the new set of objects represented by a χ -term, e.g., $pt(f)$ is completely overwritten by a new set of points-to values.

Loop Handling

Naturally, the question arises what would happen if the strong or weak update of χ -term points-to value occurs in a loop. The control-flow graph representing a loop differs from the simple *if-then-else* representation (Figure A.2). The difference is in the location of the ϕ -node: it forms a cycle and, therefore, occurs before a splitting node γ that represents a loop control-flow statement (e.g. *while*, *for*, etc.). An additional reason for such a placement of the splitting node is that a value processed inside the loop can escape any time. Therefore, the χ -term representation of an object field has to keep every value that can possibly escape the loop. Thus, it will result in a χ -term with a finite depth that corresponds to the number of loop iterations. The termination of loop analysis occurs when a fix point is reached. The splitting point γ_1 and the join point ϕ_1 implement the same behavior as was described

for the *if-then-else* statement in Section 2.3, with a minor change: additionally to the number of the basic block of the splitting point and branch index, we add a loop iteration index.

After the loop execution, the points-to set of object field f may equally point to the abstract object created before the loop and to the set of objects created inside the loop body. However, if the points-to value of f is read inside the loop, it can point only to the abstract object created in this loop iteration. Therefore, we are in the same situation that was described based on *if-then-else* control-flow statements: the restriction of the χ -term representing the $pt(f)$ by splitting point number, loop iteration index and branch number (if required) will result in the required value.

Although we can get a precise points-to set of f in each loop iteration, it can be computationally expensive since the number of loop iterations is unknown, and, therefore, there is no upper bound of the χ -term generated inside a loop. One of the possible solutions to handle this problem is χ -term loop approximation, which can have several possible approaches:

- all concrete runtime objects created at the same syntactical location are mapped to the same abstract object, and, therefore, the points-to value of f can be approximated to this abstract object;
- limit the number of loop iterations to be processed and thereby limit the depth of the produced χ -term.

Additional information on loop approximated χ -terms can be found in [47]. Moreover, the ideas presented here can be seen as a complement to the theoretical description of χ -term based points-to analysis presented by Lundberg in his licentiate thesis [47].

Bibliography

- [1] Jesper Andersson, Morgan Ericsson, Christoph Kessler, and Welf Löwe. Profile-guided composition. In Cesare Pautasso and Eric Tanter, editors, *Software Composition*, volume 4954 of *Lecture Notes in Computer Science*, pages 157–164. Springer Berlin / Heidelberg, 2008.
- [2] Jesper Andersson, Morgan Ericsson, Christoph Kessler, and Welf Löwe. Profile-guided composition. In *Proceedings of the 7th international conference on Software composition*, SC’08, pages 157–164, Berlin, Heidelberg, 2008. Springer-Verlag.
- [3] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI ’09, pages 38–49. ACM, 2009.
- [4] Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. *SIGPLAN Not.*, 38:103–114, May 2003.
- [5] Dimitri P. Bertsekas. *Constrained Optimization and Lagrange Multiplier Methods (Optimization and Neural Computation Series)*. Athena Scientific, 1 edition, 1996.
- [6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
- [7] Rastislav Bodík and Sadun Anik. Path-sensitive value-flow analysis. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’98, pages 237–251, New York, NY, USA, 1998. ACM.
- [8] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.
- [9] Randal E Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.

- [10] Gert Cauwenberghs and Tomaso Poggio. Incremental and Decremental Support Vector Machine Learning, 2001.
- [11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, New York, 2001.
- [13] Corinna Cortes and Vladimir Vapnik. Support-Vector Networks. *Machine Learning*, 20:273–297, September 1995.
- [14] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *Proceedings of the 2005 symposium on Dynamic languages*, DLS '05, pages 1–10, New York, NY, USA, 2005. ACM.
- [15] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages Systems*, 13:451–490, October 1991.
- [16] Manuvir Das, Sorin Lerner, and Mark Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 57–68, New York, NY, USA, 2002. ACM.
- [17] Klaus Denecke and Shelly L. Wismath. *Universal Algebra and Coalgebra*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2009.
- [18] Christopher P. Diehl and Gert Cauwenberghs. Svm incremental learning, adaptation and optimization. In *IJCNN*, 2003.
- [19] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. Springer-Verlag, 1985.
- [20] A. Frank and A. Asuncion. UCI machine learning repository, 2010. Software available at <http://archive.ics.uci.edu/ml>.
- [21] Jerome H. Friedman, Ron Kohavi, and Yeogirl Yun. Lazy decision trees, 1996.
- [22] Tobias Gutzmann, Antonina Khairova, Jonas Lundberg, and Welf Löwe. Towards comparing and combining points-to analyses. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code*

Bibliography

- Analysis and Manipulation*, SCAM '09, pages 45–54, Washington, DC, USA, 2009. IEEE Computer Society.
- [23] Tobias Gutzmann, Jonas Lundberg, and Welf Löwe. Towards path-sensitive points-to analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 59–68., 2007.
 - [24] Alaaeldin Hafez, Jitender Deogun, and Vijay V. Raghavan. The Item-Set Tree: A data structure for Data Mining. In *Data Warehousing and Knowledge Discovery, First International Conference, DaWaK 99*, pages 183–192. Springer, 1999.
 - [25] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 2nd edition, September 2000.
 - [26] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.
 - [27] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, March-April 2008, *ETH Zurich*, 7(3):125–151, 2008.
 - [28] Thomas Hofmann, Bernhard Schölkopf, and Alexander J. Smola. Kernel methods in machine learning. *Annals of Statistics*, 36(3):1171–1220, July 2008.
 - [29] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification. Technical report, National Taiwan University, Dep. of Comp. Science, 2003.
 - [30] C. Bielza J. A Fernandez del Pozo and M. Gomez. A list-based compact representation for large decision tables management. *European Journal of Operational Research*, 160(3):638–662, February 2005.
 - [31] Finn V. Jensen and Thomas D. Nielsen. *Bayesian Networks and Decision Graphs*. Springer Publishing Company, Incorporated, 2nd edition, 2007.
 - [32] Zhongwei Li Jianpei Zhang and Jing Yang. A divisional incremental training algorithm of support vector machine. In *Mechatronics and Automation, 2005 IEEE International Conference*, 2005.
 - [33] George H. John. Robust linear discriminant trees. In *In AI& Statistics-95*, pages 285–291. Springer-Verlag, 1994.

- [34] Samuel Johnson. Branching programs and binary decision diagrams: theory and applications by ingo wegner society for industrial and applied mathematics, 2000 408 pages. *SIGACT News*, 41:36–38, September 2010.
- [35] Hillol Kargupta and Haimonti Dutta. Orthogonal decision trees. In *Proceedings of The Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 1028–1042, 2004.
- [36] Eamonn J. Keogh and Michael J. Pazzani. Learning augmented bayesian classifiers: a comparison of distribution-based and classification-based approaches, 1999.
- [37] Christoph Kessler and Welf Löwe. A framework for performance-aware composition of explicitly parallel components. In *ParCo-2007, Jülich/Aachen, Germany, Sep. 2007*. In C. Bischof et al. (eds.): *Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing Series, Volume 15*, IOS Press, pages 227–234, 2008.
- [38] Christoph Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. In *15th Workshop on Compilers for Parallel Computing (CPC)*, 2010.
- [39] P.J.H. King. Decision tables. *The Computer Journal*, 10(2):135–142, August 1967.
- [40] Kurt A. Kramer, Lawrence O. Hall, Dmitry B. Goldgof, Andrew Remsen, and Tong Luo. Fast support vector machines for continuous data. *IEEE Transactions on Systems, Man, and Cybernetics*, 39:989–1001, 2009.
- [41] Kwok-Wa Lam and Victor C. S Lee. Building decision trees using functional dependencies. In *ITCC '04: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2*, page 470, Washington, DC, USA, 2004. IEEE Computer Society.
- [42] Ondřej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. *SIGPLAN Not.*, 46:3–16, January 2011.
- [43] Ondřej Lhoták, Stephen Curial, and José Nelson Amaral. Using ZBDDs in Points-to Analysis. In Vikram Adve, María Jesús Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, pages 338–352. Springer-Verlag, Berlin, Heidelberg, 2008.
- [44] Ondřej Lhoták and Laurie Hendren. Jedd: a BDD-based relational extension of Java. *SIGPLAN Not.*, 39:158–169, June 2004.

- [45] Xiaoming Li, Maria J. Garzarán, and David Padua. A dynamically tuned sorting library. In *Proceedings CGO'04*, pages 111–124, 2004.
- [46] Welf Löwe and Jonas Lundberg. A Scalable Flow-Sensitive Points-to Analysis. Technical report, Växjö University, School of Mathematics and Systems Engineering, 2006.
- [47] Jonas Lundberg. Points-to analysis for software engineering. Licentiate Thesis. Växjö University, School of Mathematics and Systems Engineering, 2004.
- [48] Jonas Lundberg, Tobias Gutzmann, Marcus Edvinsson, and Welf Löwe. Fast and precise points-to analysis. *Journal of Information and Software Technology*, 51:1428–1439, October 2009.
- [49] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [50] Mikhail Moshkov. Algorithms for constructing of decision trees. In *PKDD '97: Proceedings of the First European Symposium on Principles of Data Mining and Knowledge Discovery*, pages 335–342, London, UK, 1997. Springer-Verlag.
- [51] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, Bryan Singer, Manuela Veloso, and Jianxin Xiong. Generating platform-adapted DSP libraries using SPIRAL. In *High Performance Embedded Computing (HPEC)*, 2001.
- [52] José M. F. Moura, Jeremy Johnson, Robert W. Johnson, David Padua, Viktor K. Prasanna, Markus Püschel, and Manuela Veloso. SPIRAL: Automatic implementation of signal processing algorithms. In *High Performance Embedded Computing (HPEC)*, 2000.
- [53] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [54] O. J. Murphy and R. L. McCraw. Designing storage efficient decision trees. *IEEE Transactions on Computers*, 40(3):315–320, 1991.
- [55] Nils J. Nilsson. Introduction to machine learning: An early draft of a proposed textbook, 1996. Software available at <http://robotics.stanford.edu/people/nilsson/mlbook.html>.
- [56] Jonathan J. Oliver. Decision graphs - an extension of decision trees, 1993.
- [57] Marek Olszewski and Michael Voss. An install-time system for the automatic generation of optimized parallel sorting algorithms. In *Proceedings PDPTA'04, Vol. 1*, jun 2004.

- [58] M. Steinbach P.-N. Tan and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.
- [59] Giulia Pagallo and David Haussler. Boolean feature discovery in empirical learning. *Machine Learning*, 5(1):71–99, 1990.
- [60] Ling Chen Ping He and Xiao-Hua Xu. Fast C4.5. In *Proc Int Conf Machine Learning Cybernetics, ICMLC 2007*, volume 5, pages 2841–2846, Hong Kong, China, August 2007.
- [61] Prakash Prabhu and Priti Shankar. Field flow sensitive pointer and escape analysis for Java using heap array SSA. In *Proceedings of the 15th international symposium on Static Analysis, SAS '08*, pages 110–127, Berlin, Heidelberg, 2008. Springer-Verlag.
- [62] J. R. Quinlan. Generating production rules from decision trees. In *IJCAI'87: Proceedings of the 10th international joint conference on Artificial intelligence*, pages 304–307, San Francisco, CA, USA, 1987. Morgan Kaufmann Publishers Inc.
- [63] J. R. Quinlan. Simplifying decision trees. *International Journal of Human-computer Studies / International Journal of Man-machine Studies*, 27(3):221–234, 1987.
- [64] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.
- [65] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [66] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theory and Applications*. World Scientific, Singapore, 2008.
- [67] S. Ruggieri. Efficient c4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14:438–444, 2002.
- [68] Stefan Rüping. Incremental learning with support vector machines. In *Proceedings of the 2001 IEEE International Conference on Data Mining, ICDM '01*, pages 641–642, Washington, DC, USA, 2001. IEEE Computer Society.
- [69] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th international conference on Compiler construction, CC'03*, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

- [70] Ganesh Sittampalam, Oege de Moor, and Ken Friis Larsen. Incremental execution of transformation specifications. *SIGPLAN Not.*, 39:26–38, January 2004.
- [71] Nadeem Ahmed Syed, Syed Huan, Liu Kah, and Kay Sung. Incremental learning with support vector machines, 1999.
- [72] Yan Tang and Robert Meersman. *On Constructing Semantic Decision Tables*, volume 4653, pages 34–44. Springer, Berlin/Heidelberg, 2007.
- [73] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '05, pages 277–288, New York, NY, USA, jun 2005. ACM.
- [74] Martin Trapp. *Optimierung Objektorientierter Programme*. PhD thesis, Universität Karlsruhe, Karlsruhe, December 1999.
- [75] Ricardo Vilalta, Gunnar Blix, and Larry Rendell. Global data analysis and the fragmentation problem in decision tree induction. In *In 9th European Conference on Machine Learning*, pages 312–326. Springer-Verlag, 1997.
- [76] Martin von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 143–156, New York, USA, 2007. ACM.
- [77] Lipo Wang and Xiuju Fu. *Data Mining with Computational Intelligence (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
- [78] Yi-Min Wen and Bao-Liang Lu. Incremental learning of support vector machines by classifier combining. In *Proceedings of the 11th Pacific-Asia conference on Advances in knowledge discovery and data mining*, PAKDD'07, pages 904–911, Berlin, Heidelberg, 2007. Springer-Verlag.
- [79] John Robert Wernsing and Greg Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 115–124, New York, NY, USA, 2010. ACM.

- [80] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Proceedings of Programming Languages and Systems: Third Asian Symposium*, 2005.
- [81] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, 2004.
- [82] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. *SIGPLAN Not.*, 34, October 1999.
- [83] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.
- [84] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37, 2007.
- [85] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. *SIGSOFT Software Engineering Notes*, 28:327–336, September 2003.
- [86] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 17:1084–1096, 2006.
- [87] Du Zhang and Jeffrey J.P. Tsai. *Machine Learning Applications in Software Engineering*. World Scientific Publishing Company, Incorporated, River Edge, NJ, USA, 2005.
- [88] Jianwen Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design, ICCAD '02*, pages 150–157, New York, NY, USA, 2002. ACM.