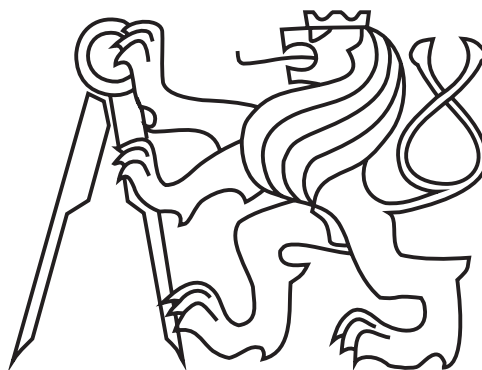


CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING
DEPARTMENT OF CYBERNETICS



Vojtěch Aschenbrenner

**Deep Relational Learning with
Predicate Invention**

Master Thesis

Prague, Jan, 2013

Study Programme: Open Informatics
Branch of Study: Artificial Intelligence

Advisor: Ing. Ondřej Kuželka

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Vojtěch A s c h e n b r e n n e r

Study programme: Open Informatics

Specialisation: Artificial Intelligence

Title of Diploma Thesis: Deep Relational Learning with Predicate Invention

Guidelines:

1. Familiarize yourself with foundations of inductive logic programming, with methods for hypothesis search and methods for theta-subsumption. Study basics of algorithms for constraint satisfaction.
2. Design and implement efficient algorithms for query answering in a subset of Datalog – nonrecursive Datalog.
3. Using the developed query answering algorithms, implement a method for learning non-recursive theories specified by the thesis advisor.


Bibliography/Sources: Will be provided by the supervisor.

Diploma Thesis Supervisor: Ing. Ondřej Kuželka

Valid until: the end of the summer semester of academic year 2012/2013


prof. Ing. Vladimír Mařík, DrSc.
Head of Department




prof. Ing. Pavel Ripka, CSc.
Dean

Prague, July 3, 2012

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Vojtěch Aschenbrenner
Studijní program: Otevřená informatika (magisterský)
Obor: Umělá inteligence
Název tématu: Relační strojové učení s postranní indukci pomocných konceptů

Pokyny pro vypracování:

1. Seznamte se se základy induktivního logického programování, s metodami pro hledání hypotéz a rozhodování theta-subsumpce. Dále se seznamte se základy CSP (constraint satisfaction problems).
2. Navrhněte a naprogramujte co možná nejefektivnější algoritmy pro zodpovídání dotazů pro podmnožinu Datalogu bez rekurze.
3. S pomocí algoritmů pro zodpovídání dotazů (bod 2) naimplementujte metodu pro učení se nerekurzivních teorií specifikovanou vedoucím práce.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Ondřej Kuželka

Platnost zadání: do konce letního semestru 2012/2013


prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry




prof. Ing. Pavel Ripka, CSc.
děkan

*We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.*

— Donald E. Knuth [20]

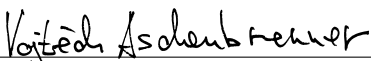
ACKNOWLEDGMENTS

I would like to thank my advisor Ondřej Kuželka for his invaluable advice and his role of faultless advisor. Also, I would like to thank the computer in Intelligent Data Analysis group named Otrok. My computations were one of the latests before he went to the silicon heaven. Finally, I am really beholden to all creators of good open-source software, especially vim, eclim, latex and git.

DECLARATION

I declare, that I have created this thesis on my own and I have also quoted every used information source. This has been done according to the methodical directive about keeping ethical principles during preparation of final projects at university.

Prague, January 3, 2013

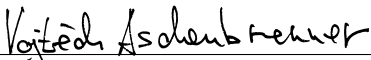


Vojtěch Aschenbrenner

PROHLÁŠENÍ

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Praha, 3. ledna 2013



Vojtěch Aschenbrenner

ABSTRACT

This thesis deals with a design of novel classification model from the field of relational machine learning, which uses a structure inspired by artificial neural networks for representation of logic program. The logic program syntax is a subset of Datalog but extended with disjunctions with weight coefficients. The weights are used for learning the model with modified backpropagation method known from artificial neural networks. Then the work presents experimental implementation of the proposed model and validates its accuracy. The accuracy is close to the best algorithms from the literature. Moreover, the presented system can be further extended in various directions. So far, it implements disjunction weights learning but not structure learning, which could potentially boost its performance significantly. This remains a challenge for future work.

ABSTRAKT

Tato práce se zabývá návrhem nového klasifikačního modelu z oblasti relačního strojového učení, který pro reprezentaci logického programu využívá strukturu inspirovanou umělými neuronovými sítěmi. Syntax logického programu je podmnožinou Datalogu, avšak rozšířenou o váhový koeficient u pravidel disjunkce. Právě tyto váhy jsou využity pro učení modelu pomocí upravené metody backpropagation. Dále práce poskytuje experimentální implementaci navrženého modelu a ověřuje jeho přesnost klasifikace. Ta se velmi blíží nejlepším algoritmům z literatury. Navíc tento systém může být dále rozšiřován. Dosud totiž implementuje pouze učení vah disjunkcí, ale už ne učení struktury. Právě učení struktury může velmi znatelně zlepšit přesnost klasifikace a bude tak cílem budoucí práce.

KEYWORDS

Relational Learning, Inductive Logic Programming, Constraint Satisfaction Problem, Theta-Subsumption, Predicate Invention, Datalog Queries, Artificial Neural Networks

KLÍČOVÁ SLOVA

Relační učení, Induktivní logické programování, Problémy s omezujícími podmínkami, Theta subsumpce, Vytváření predikátů, Datalog dotazy, Umělé neuronové sítě

CITE AS

Vojtěch Aschenbrenner, *Deep Relational Learning with Predicate Invention*, Master Thesis, Prague, CZ, Czech Technical University in Prague, 2013, p. 80.

BIB_TE_X:

```
@MASTERSTHESIS{
  author   = {Vojtěch Aschenbrenner},
  title    = {Deep Relational Learning
             with Predicate Invention},
  pages    = {80}
  school   = {Czech Technical University in Prague},
  year     = {2013},
  language = {English},
  location = {Prague, CZ}
}
```



CONTENTS

1	THESIS OVERVIEW	1
1.1	Introduction	1
1.2	Aims and Objectives	3
1.3	Thesis Organization	3
2	THEORETICAL FOUNDATIONS	5
2.1	Inductive Logic Programming	5
2.1.1	Terminology	5
2.1.2	θ -subsumption	7
2.1.3	Inductive Logic Programming	8
2.2	Constraint Satisfaction Problem and Combinatorial Optimization . .	11
2.2.1	Introduction	11
2.2.2	Binarization of Constraints	12
2.2.3	Consistency Techniques	12
2.2.4	Forward Checking	13
2.2.5	Variable Ordering	15
2.2.6	Branch and Bound	16
2.2.7	Restarted Strategy	18
2.3	Artificial Neural Networks	18
2.3.1	Feed-forward Architecture	18
2.3.2	Learning of Artificial Neural Network	19
2.4	Fuzzy Logic	21
2.4.1	Łukasiewicz Logic	22
3	STATE OF THE ART IN RELATED AREAS	23
3.1	Datalog	23
3.1.1	Existing Datalog Solvers	23
3.1.2	Suitability for Algorithm	24
3.2	θ -subsumption Engines	25
3.2.1	Resumer2	26
4	PROPOSED MODEL	27
4.1	$\lambda\kappa$ -program	28
4.2	$\lambda\kappa$ -template	29
4.3	$\lambda\kappa$ -network	31
4.4	Two Examples	34
4.4.1	First Example	34

4.4.2	Second Example	38
4.5	Learning	41
4.5.1	Finding Maximal Substitution	41
4.5.2	Gradient Descent	42
4.6	Classification Network	44
4.6.1	Multi-criteria Classification	44
5	PROPOSED ALGORITHM	47
5.1	Learning Phase	47
5.1.1	Threshold Learning	49
5.2	Classification Phase	49
5.3	Effective Implementation Details	51
5.3.1	Biggest Bottleneck	51
5.3.2	Basic Approach for Solving $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -network.	51
5.3.3	Sample Representation	52
5.3.4	Variable Ordering	53
5.3.5	Forward Checking	53
5.3.6	Branch and Bound	54
5.3.7	Caching	55
6	EXPERIMENTS	56
6.1	$\lambda\kappa$ -programs	56
6.2	Learning Analysis	59
6.3	Results	59
7	CONCLUSION AND FUTURE WORK	61
7.1	Conclusion	61
7.2	Future Work	62
	BIBLIOGRAPHY	63
A	CONTENT OF CD	67

LIST OF FIGURES

Figure 2.1	Completeness and Consistency of Hypothesis	10
Figure 2.2	Four Queens Problem State-space	14
Figure 2.3	Branch and Bound Example	17
Figure 2.4	Simplified Model of Neuron	19
Figure 2.5	Feed-forward Artificial Neural Network	20
Figure 2.6	Temperature Example in Fuzzy Logic	22
Figure 4.1	Proposed n -layered $\lambda\kappa$ -template	31
Figure 4.2	Node	34
Figure 4.3	Maximal Substitution Example	35
Figure 4.4	5-layered $\lambda\kappa$ -template for the Most Blue Path Problem	37
Figure 4.5	5-layered $\lambda\kappa$ -network for the Most Blue Path Problem	37
Figure 4.6	5-layered $\lambda\kappa$ -template for More Complex Example	40
Figure 4.7	5-layered $\lambda\kappa$ -network for More Complex Example	40
Figure 4.8	Threshold Finding	45
Figure 4.9	Multi-criteria Classification Model	45
Figure 5.1	Learning Schema	48
Figure 5.2	Classification Schema	50
Figure 6.1	Dispersion	59
Figure 6.2	Learning Error	59
Figure 6.3	PTC-MR Accuracy	60
Figure 6.4	Mutagenesis Accuracy	60

LIST OF TABLES

Table 2.1	Examples for Basic Definitions from Logic	7
Table 2.2	Inductive Logic Programming Example	10
Table 2.3	Backpack Problem Specification	17
Table 3.1	Performance Comparison of θ -subsumers	25
Table 5.1	Literal Partition	53
Table 6.1	Parameters of Algorithm	60

Table 6.2	Accuracies	60
-----------	----------------------	----

LISTINGS

Listing 2.1	Node Consistency Algorithm	12
Listing 2.2	Arc Revision Algorithm	13
Listing 2.3	Arc Consistency Algorithm AC ₃	13
Listing 2.4	Forward Checking	14
Listing 2.5	On-line Backpropagation Learning	21
Listing 4.1	Datalog Program	29
Listing 4.2	3-layered $\lambda\kappa$ -program	29
Listing 4.3	Maximal Substitution	33
Listing 4.4	$\lambda\kappa$ -program for the Most Blue Path Problem	36
Listing 4.5	More Complex $\lambda\kappa$ -program	39
Listing 4.6	Learning Algorithm	41
Listing 5.1	Caching Results	55
Listing 6.1	PTC-MR Bottom Layers	57
Listing 6.2	Common Upper Layers	57
Listing 6.3	Mutagenesis Bottom Layers	58

ACRONYMS

- ILP Inductive Logic Programming
- CSP Constraint Satisfaction Problem
- ANN Artificial Neural Network
- BB Branch and Bound
- RPROP Resilient Backpropagation
- AI Artificial Intelligence
- SRL Statistical Relational Learning

API Application Interface

1 | THESIS OVERVIEW

1.1 INTRODUCTION

The ultimate goal of Artificial Intelligence (AI) is to create computational agents, that would behave intelligently. In this context, the word *intelligently* means, that the behaviour of these agents should lead to maximizing the chance to succeed. Therefore, it is important for the agents to be able to capture the main specifics of the environments, in which they operate in their mental models in order to be able to reason the environments and the effects of their actions. Models of most non-trivial real-life environments are usually characterised by having both a rich complex structure and having a significant uncertainty component (probabilistic or other). (Poole and Mackworth [36])

In the past, two large and separated paradigms emerged: logical (a.k.a. symbolic) and statistical AI. These two paradigms have started to converge only recently, giving rise to a field known as statistical relational AI, whose most rapidly developing subfield is Statistical Relational Learning (SRL).

The logical AI uses logic (often predicate logic, but also more complex logics) as a tool for representation and reasoning the real world. This approach is suitable for capturing the complexity of the world, because it is able to naturally describe relations among objects. It is often called relational AI because of these relations. Many approaches arose in this type of AI, e. g., logical programming, description logics, classical planning, symbolical parsing, rule induction etc. (Poole [35]) These methods are widely used in computational biology, natural language processing, business intelligence etc. The disadvantage of this approach is that it cannot affect the uncertainty in a natural way.

Statistical AI is the paradigm where the probability theory is used for describing and reasoning the environment. Probability can naturally express the uncertainty of the world with the random variables, where each random variable expresses some phenomenon with non-negative probability. (Talbot [46]) Into this group, we can place approaches such as Bayes Networks, Hidden Markov Models, Artificial Neural Network (ANN), Markov Decision Process, Gaussian Mixture Model etc. Their main usage is in the fields, where uncertainty plays the main role, e. g., robot

controlling, finance prediction, medical diagnosis, decision making maximizing expected utility, adaptive testing etc.

As we have already mentioned, the world is complex as well as uncertain. Therefore, many researchers asked whether the combination of the two approaches can be convenient. As a result, a new field combining statistical and logical AI, called statistical relational AI, emerged. Many new approaches originated in this new field through the so-far short course of its existence. A prominent example is Markov Logic, widely used for link prediction, entity resolution, information extraction etc. (Domingos et al. [9]) Markov Logic extends predicate logic with weights for logical clauses. This logic is then used as a template for Markov network with weights which can be learnt by various techniques. Also learning the structure of Markov network is possible. Other approaches to SRL are, for example, knowledge-based model construction (Wellman et al. [48], Ngo and Haddawy [32], Kersting and De Raedt [18]), stochastic logic programs (Muggleton et al. [30], Cussens [7]), probabilistic relational models (Getoor et al. [12]), relational Markov models (Anderson et al. [1]), relational Markov networks (Taskar et al. [47]), relational dependency networks (Neville and Jensen [31]) and structural logistic regression (Popescul and Ungar [37]).

Inductive Logic Programming (ILP) is a research area at the intersection of machine learning and logic programming. It aims at finding a hypothesis represented by a logic and the logic is also used for representation of examples and background knowledge. In the beginnings, basic ILP techniques established. These techniques were crisp which means that the probability theory was not used in them. Techniques searched hypothesis space in top-down (specialization) or bottom-up (generalization) manner. (Lavrač and Dzeroski [26]) Later, an approach called propositionalization appeared. Propositionalization is a method for transformation relational learning problem to an attribute-value problem. This transformed problem can be then solved by propositional learners and translated back to relational form. An example introducing probability to ILP is a combination of propositionalization with linear classifier, where constructed features are crisp but are weighted by linear classifier. (Paes et al. [33]) Our approach goes further in including fuzziness to ILP. The system is able to learn helpful non-crisp concepts (predicate invention), which can be used for definition of other concepts or final hypothesis. This is a crucial ability that is not offered by most ILP systems. We use methods from ANN for learning the model because of its structure similarity.

1.2 AIMS AND OBJECTIVES

The main aim of this work is to develop a novel algorithm for the field of *ILP* that, first, would be able to learn definitions of relational concepts and use these in the learning process and, second, that would be able to deal with uncertainty. The novel algorithm presented in this thesis is based on transforming modified Datalog program into network structure, which can be learnt by methods known from *ANN*. The main objectives are:

- Design a language similar to Datalog viable for transforming program written in that language into structural representation.
- Design and implement an effective reasoner for the proposed language. This means to create a reasoner, which contains an implementation of various optimization techniques e. g., forward checking, variable ordering or Branch and Bound (*BB*).
- Implement a learning algorithm inspired by *ANN* for the structures representing programs in proposed language.
- Create a classification algorithm using the developed reasoner and learning algorithm.
- Experimentally check accuracy of the given approach and discuss future extensions.

1.3 THESIS ORGANIZATION

This thesis is divided into chapters as follows.

1. THESIS OVERVIEW The first chapter provides an introduction to the work, its aims and organization.

2. THEORETICAL FOUNDATIONS The second chapter reviews theoretical foundations from different fields of computer science and mathematics needed for design and effective implementation of our algorithm. It explains concepts from *ILP*, Constraint Satisfaction Problem (*CSP*), *ANN* and Fuzzy Logic.

3. STATE OF THE ART IN RELATED AREAS The third chapter describes the state of the art in the fields related to our proposed model e. g., tools for answering

Datalog queries and engines for θ -subsumption. A section covering a state-of-the-art θ -subsumption algorithm in detail is provided in order to explain motivation behind our own θ -subsumption solver.

4. PROPOSED MODEL One of the main chapters of this work is the fourth chapter. It describes several theoretical concepts from proposed model. This description is extended with two examples showing the function of the model. Finally, a learning algorithm is derived and it is shown how to perform classification with the learnt models.

5. PROPOSED ALGORITHM The fifth chapter is about the implementation of the proposed model. It starts with defining concrete design of learning and classification phases of the algorithm and then it proposes an effective implementation.

6. EXPERIMENTS Next chapter contains experiments on publicly available data for SRL and a comparison with the best algorithms.

7. CONCLUSION AND FUTURE WORK The last chapter provides conclusion and summarization of this thesis. Finally we propose various extensions and our future work.

2 | THEORETICAL FOUNDATIONS

This chapter provides necessary theoretical foundations for understanding our learning model and algorithms. It contains three theoretical parts, each from different part of computer science.

The first part of the preliminaries is generally about [ILP](#). This part is described more widely than needed for this work, but as this work belongs to the [ILP](#) category it seems like a reasonable choice.

The second part is from the field of combinatorial optimization, specifically from [CSP](#). Techniques from [CSP](#) will be very useful in speeding up problems related to [ILP](#) e. g., finding θ -subsumption.

Next part is dedicated to [ANN](#). The structure of [ANN](#) is an inspiration for our model and also for its future extension. We use algorithms similar to forward propagation for determining the value of the network or backpropagation for learning the network in a slightly modified form, which is suitable for the relational learning model, that we solve. In [ANN](#), we can find inspiration for our future work because of its structure similarity. In the future, we can try advanced approaches from [ANN](#) on our model.

Also inspiration from fuzzy logic is used in one part of our model, thus a short note on this non-classical logic is a part of this section. We use only basic principles of fuzzy logic, concretely of the Łukasiewicz logic, thus this part will be elementary.

2.1 INDUCTIVE LOGIC PROGRAMMING

2.1.1 Terminology

In this subsection, we will mention fundamental definitions from logic and deductive databases. The knowledge of these definitions is required for understanding next sections, mainly the chapter about the proposed model (Chapter 4).

Definition 2.1.1 (Atom).

An atom is a predicate symbol immediately followed by a bracketted tuple of terms.

Definition 2.1.2 (Literal).

A literal is an atom (positive literal) or a negated atom (negative literal).

Definition 2.1.3 (Clause).

A clause is a formula of the form

$$\forall X_1, \forall X_2, \dots, \forall X_s (L_1 \vee L_2 \vee \dots \vee L_m) \quad (2.1)$$

where each L_i is a literal and X_1, X_2, \dots, X_s are all the variables occurring in $L_1 \vee L_2 \vee \dots \vee L_m$.

Definition 2.1.4 (Horn Clause).

A Horn clause is a clause, which contains at most one positive literal.

Definition 2.1.5 (Definite Program Clause).

A definite program clause is a clause, which contains exactly one positive literal. It is shown in Equation (2.2), where T, L_1, \dots, L_m are atoms.

$$T \leftarrow L_1, \dots, L_m \quad (2.2)$$

- We call the part on the left head and the part on the right body.
- If we omit the body, then we get a positive unit clause also known as fact in Prolog terminology.
- The empty head with non-empty body is called definite goal.

Definition 2.1.6 (Definite Program).

A set of definite program clauses is called a definite logic program.

Only atoms are allowed in the body of a definite logic program. However, in Prolog, the body can contain negated atoms in non-logical sense, called negation as failure (Clark [5]).

Definition 2.1.7 (Program Clause).

A program clause is a clause of the form shown in Equation (2.3), where T, L are an atoms, and L_1, \dots, L_m are in the form L or negated L .

$$T \leftarrow L_1, \dots, L_m \quad (2.3)$$

Definition 2.1.8 (Normal Program).

A normal program is a set of program clauses.

Definition 2.1.9 (Predicate Definition).

A predicate definition is a set of program clauses with the same predicate symbol and arity in their heads.

DEFINITION	EXAMPLE
Definite program clause	$\text{daughter}(X, Y) \leftarrow \text{female}(X), \text{mother}(Y, X)$
(Logic notation)	$\forall X \forall Y : \text{daughter}(X, Y) \vee \overline{\text{female}(X)} \vee \overline{\text{mother}(Y, X)}$
(Set notation)	$\{\text{daughter}(X, Y), \overline{\text{female}(X)}, \overline{\text{mother}(Y, X)}\}$
Program clause	$\text{daughter}(X, Y) \leftarrow \text{not}(\text{male}(X)), \text{mother}(Y, X)$

Table 2.1: Examples for Basic Definitions from Logic

Table 2.1 shows examples illustrating definitions mentioned above also with examples in different notations.

Definition 2.1.10 (Datalog Clause).

A clause without function symbols of arity n , where $n \geq 1$, and recursion is called a datalog clause.

Definition 2.1.11 (Datalog Program).

A datalog program is a set of datalog clauses.

Datalog with some minor extensions is the language used in our model. All modifications to the language will be discussed later in Chapter 4.

Definition 2.1.12 (Substitution).

A substitution is a tuple $s = V_1/t_1, \dots, V_n/t_n$, where V_i/t_i means an assignment of term t_i to variable V_i . When applying a substitution s to a term, atom or clause all occurrences of the variable V_i are simultaneously replaced by the term t_i .

2.1.2 θ -subsumption

One of the operations performed most often in ILP is testing, whether hypothesis H covers example e from the database assuming some background knowledge B . This is in fact a logic implication in the form $e \cup B \rightarrow H$. First-order-logic implication is undecidable problem (Schmidt-Schauss [45]), thus we need some kind of approximation (Plotkin [34]). The approximation of logic implication is a θ -subsumption and belongs to NP-complete complexity class (Kapur and Narendran [17]).

Definition 2.1.13 (θ -subsumption).

A clause C θ -subsumes a clause D ($C \vdash_{\theta} D$) if and only if there exists a substitution θ such that $C\theta \subseteq D$.

Example 2.1.1 (θ -subsumption).

$C : h(X_0) \leftarrow l1(X_0, X_1), l1(X_0, X_2), l1(X_0, X_3), l2(X_1, X_2), l2(X_1, X_3)$

$D : h(c_0) \leftarrow l1(c_0, c_1), l1(c_0, c_2), l2(c_1, c_2)$
 $C\theta$ subsumes D with $\theta = \{X_0/c_0, X_1/c_1, X_2/c_2, X_3/c_2\}$.

According to Santos and Muggleton [44], the standard way of solving θ -subsumption is based on SLD-resolution in Prolog. The algorithm performs an assignment from the literals in the clause C to the literals in clause D . This is done with depth-first search algorithm, where the literals are taken from left to right. Thus the ordering of variables has a huge impact on the size of the state-space.

2.1.2.1 Time Complexity

The standard algorithm for checking θ -subsumption problems $C \vdash_{\theta} D$ has the time complexity of $O(M^N)$, where M and N are the lengths of clauses C and D respectively. This is because we are mapping each literal from C to D . But in practice, the SLD-resolution tests, whether current binding is feasible during the substitution. Thus if $M \approx N$, then the problem may be overconstrained and easier (Santos and Muggleton [44]).

The mentioned work also discusses a complexity improvement. We can map the θ -subsumption problem to the problem of mapping from V to T , where V and T are the sets of distinct variables in C respectively D . This approach has a complexity $O(|T|^{|V|})$. Since $|T| \ll M$ and $|V| \ll N$, the complexity of this approach is better.

2.1.3 Inductive Logic Programming

ILP is a research area at the intersection of machine learning and logic programming (Lavrac and Dzeroski [26], Muggleton [29]).

Before we introduce basic ILP problems, we need to define a few basic terms. Below we describe terms like concept or inductive concept learning. Original definitions can be found in Lavrac and Dzeroski [26].

Definition 2.1.14 (Concept).

Let U be a universal set of objects. Then concept C is a subset of objects in U : $C \subseteq U$.

Example 2.1.2 (Concept).

Let U be a set of all patients in register and concept $C \subseteq U$ describing all patients having given disease.

Definition 2.1.15 (Inductive concept learning).

A concept C is described by examples from a set E^+ . Moreover, there is a set E^- representing negative examples. Inductive concept learning tries to find a hypothesis H in given language L , such that:

- every positive example $e \in E^+$ is covered by H ,
- no negative example $e \in E^-$ is covered by H .

Example 2.1.3 (Inductive concept learning).

Let us have the example e (2.4) and hypothesis H (2.5). The hypothesis H covers example e , because the example e is entailed by the hypothesis H .

$$e : [\text{Suit}_1 = \text{diamonds}] \wedge [\text{Rank}_1 = 7] \wedge [\text{Suit}_2 = \text{hearts}] \wedge [\text{Rank}_2 = 7] \quad (2.4)$$

$$\begin{aligned} H : \text{pair if } & [\text{Rank}_1 = 7] \wedge [\text{Rank}_2 = 7] \vee \\ & [\text{Rank}_1 = 8] \wedge [\text{Rank}_2 = 8] \vee \\ & \vdots \\ & [\text{Rank}_1 = a] \wedge [\text{Rank}_2 = a] \end{aligned} \quad (2.5)$$

In practice, there is often no hypothesis covering all positive examples and none of the negatives because of noise in the data (at least if we want to avoid overfitting). Hypothesis is complete if it covers all positive examples, otherwise it is incomplete. We say that a hypothesis is consistent if it does not cover a negative example, otherwise it is inconsistent. According to completeness and consistency we distinguish four types of hypothesis. 1) Consistent and complete, 2) consistent and incomplete, 3) inconsistent and complete, and 4) inconsistent and incomplete. These possibilities are depicted in Figure 2.1.

ILP adds background knowledge to inductive concept learning, which can help when learning more complex hypothesis. Thus the aim of ILP is to define an unknown relation using relations from the background knowledge, from examples and constrained by the hypothesis language. The language for description of hypothesis, background knowledge and the examples is a logic program e. g., Prolog. This means, that creating a hypothesis is in fact Prolog program synthesis. The main usage of ILP tends to be in bioinformatics and natural language processing.

Example 2.1.4 (ILP from [26]).

The task is to define the target relation $\text{daughter}(X, Y)$, which states that person X is a daughter of person Y , in terms of the background knowledge relations female and parent . These relations are given in Table 2.2. There are two positive and two negative examples of the target relation.

It is possible to formulate the definition of the target concept (2.6) in the hypothesis language of Horn clauses.

$$\text{daughter}(X, Y) \leftarrow \text{female}(X), \text{parent}(Y, X) \quad (2.6)$$

This definition is consistent and complete with respect to the background knowledge and the training examples.

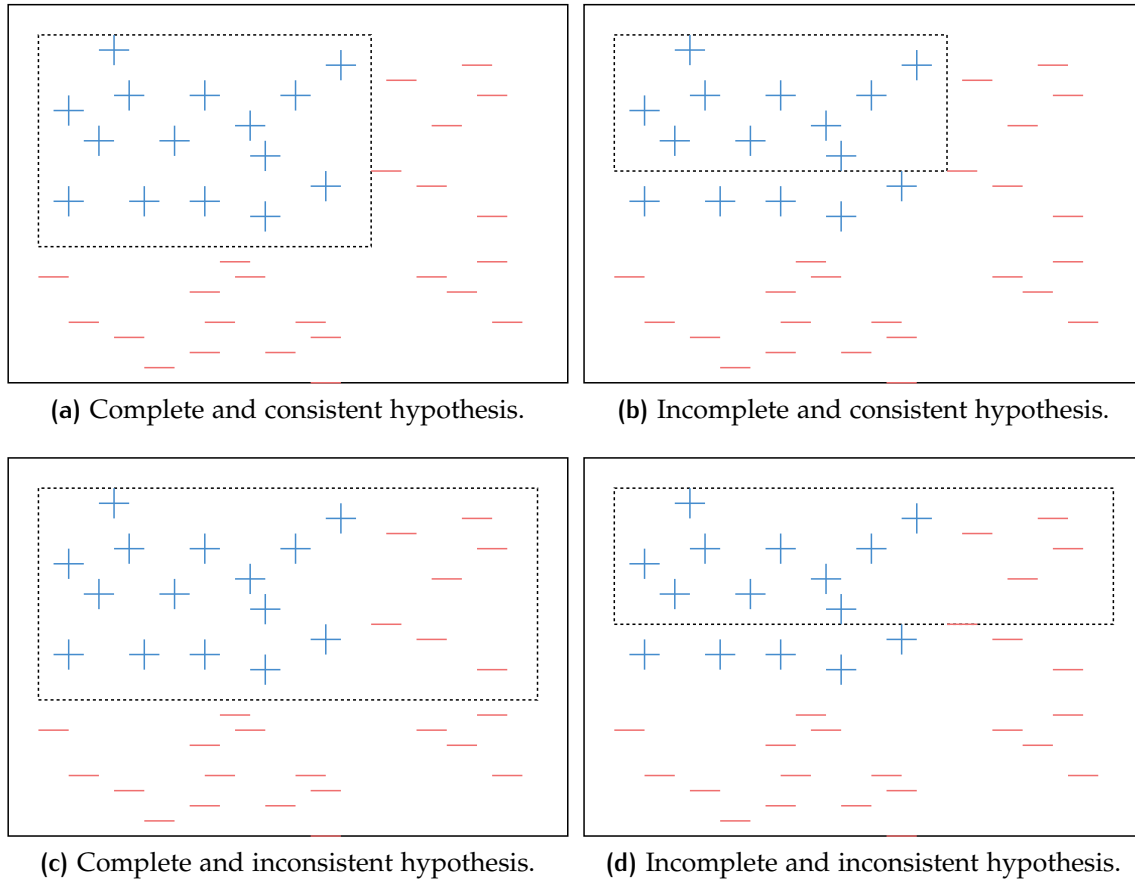


Figure 2.1: Completeness and consistency of a hypothesis. The hypothesis H is drawn in dotted style, the positive examples are dotted as $+$ and the negatives like $-$.

Training examples	Background knowledge	
$+$ daughter(mary, ann).	parent(ann, mary).	female(ann).
$+$ daughter(eve, tom).	parent(ann, tom).	female(marry).
$-$ daughter(tom, ann).	parent(tom, eve).	female(eve).
$-$ daughter(eve, ann).	parent(tom, ian).	

Table 2.2: A simple *ILP* problem: learning the daughter relation. Positive examples marked as $+$, negative as $-$.

2.2 CONSTRAINT SATISFACTION PROBLEM AND COMBINATORIAL OPTIMIZATION

Constraint programming or **CSP** is an elegant style of declarative programming widely used in **AI**, especially suitable for solving combinatorial problems. As we could see in **Kuželka and Železný [21]**, combining **CSP** with **ILP** can lead to very good results. Main material for construction of this part and studying **CSP** algorithms is (**Barták [4]**).

2.2.1 Introduction

Definition 2.2.1 (CSP).

A **CSP** is a triple (V, D, C) where

- V is a finite set of variables $V = \{V_1, \dots, V_n\}$,
- D is a finite set of domains $D = \{D_1, \dots, D_n\}$, where each domain D_i contains possible values for V_i and
- C is a finite (possibly empty) set of constraints on an arbitrary subset of variables in V .

As a solution to **CSP**, we want to find an assignment for all variables x_i from D_i satisfying the given constraints. The solutions of this problem can vary, and it depends on us, what criteria we choose. We have following possibilities:

- Find any solution.
- Find all solutions.
- Find a solution with certain quality according to some objective function.
- Find the best solution.

In our algorithm, we will want to find the best (optimal) solution.

Of course, the solution of **CSP** can be found by systematic search through the whole state-space generated by variables and their assignments. Unfortunately the state-space is often very large and thus cannot be searched exhaustively in practice. Algorithms from **CSP** are the answer for this problem, because they can reduce the state-space rapidly.

2.2.2 Binarization of Constraints

Most of CSP algorithms are designed for binary or unary constraints. Unary constraints are not so interesting, because they can be easily solved by simply removing unfeasible values for variable V_i from domain D_i .

On the other hand, a binary CSP is more interesting, because all n -ary constraints can be converted to equivalent binary CSP. Binary CSP is often depicted as constraint graph (G) where

- each node is a variable (V_i) and
- each arc (V_i, V_j) is a constraint between two variables connected with this arc.

The idea of converting a general CSP problem with constraints of arbitrary arity to binary CSP is based on introducing new variable that encapsulates the set of constrained variables. The encapsulated variable has a domain that is a Cartesian product of the domains of individual variables. Explaining all details about binarization algorithms is not necessary for our work and interested readers can read Rossi et al. [42] or Bacchus and Van Beek [2]. All further described techniques will refer to binary CSP.

2.2.3 Consistency Techniques

For description of CSP algorithms we need to define two consistency techniques.

1. Node consistency is the simplest consistency test. A constraint graph is node consistent if for every node Equation (2.7) holds. The algorithm for performing node consistency is shown in Listing 2.1.

$$\forall x \in D_i : V_i = x \wedge \text{consistent}(V_i) \quad (2.7)$$

```

procedure nc
  for each V in nodes(G)
    for each X in the domain D of V
      if any unary constraint on V is inconsistent with X
        delete X from D
  
```

Listing 2.1: Node Consistency Algorithm

2. Arc consistency ensures, that all arcs are consistent i. e., all binary constraints are satisfied. Let us have an arc between V_i and V_j labeled as (V_i, V_j) . The arc is consistent if Equation (2.8) holds.

$$\forall x \in D_i, \exists y \in D_j : V_i = x \wedge V_j = y \wedge \text{consistent}(V_i, V_j) \quad (2.8)$$

Arc consistency can be enforced by deleting values from the domain of V_i , for which Equation (2.7) does not hold. This is done by Listing 2.2. However repeating this algorithm for every node will not make the constraint graph arc consistent. E. g., if this algorithm reduces domain of variable V_i , then we need to check the arc (V_j, V_i) again. Thus more sophisticated algorithms emerged, which uses revise algorithm cleverly (Listing 2.3).

```

procedure revise( $V_i, V_j$ )
  deleted = false;
  for each  $X$  in  $D_i$  do
    if there is no such  $Y$  in  $D_j$  such that  $(X, Y)$  is consistent
      delete  $X$  from  $D_i$ 
      deleted = true
  return deleted

```

Listing 2.2: Arc Revision Algorithm

```

procedure AC3
  queue =  $(V_i, V_j)$  in arcs( $G$ ),  $i \neq j$ 
  while not queue empty
    select and delete any arc  $(V_k, V_m)$  from queue
    if revise( $V_k, V_m$ )
      queue = queue union  $(V_i, V_k)$  such that  $(V_i, V_k)$  in arcs( $G$ ),  $i \neq k, i \neq m$ 

```

Listing 2.3: Arc Consistency Algorithm AC₃

2.2.4 Forward Checking

One of the simplest methods for reducing the size of state-space is forward checking. Despite its simplicity, this algorithm reduces state-space rapidly and it has almost no computational overhead. The algorithm's pseudocode can be seen in Listing 2.4.

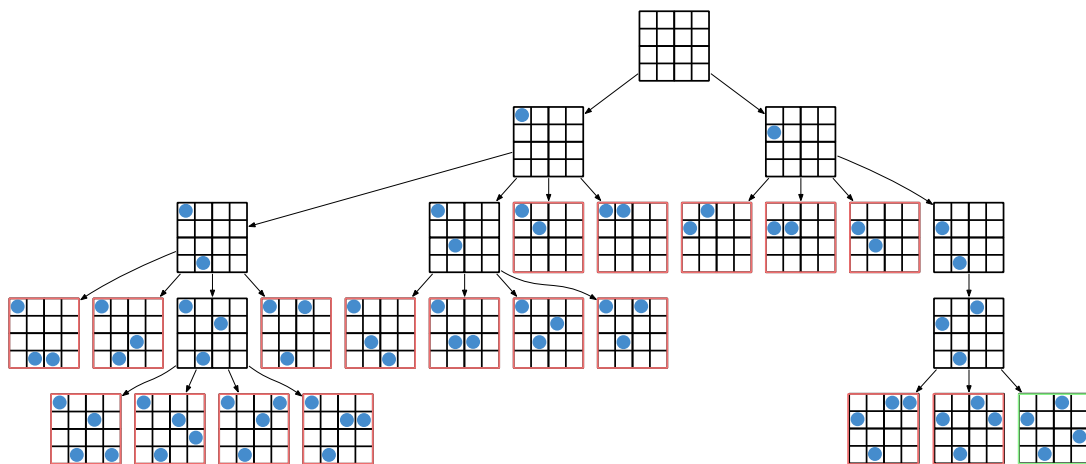
Typical example of usage of forward checking is the n queen problem. Let us have $n \times n$ chessboard and we ask how to place n queens on chess board, so that none of them can hit any other in one move. The example on Figure 2.2 shows how the algorithm works on this problem when $n = 4$. Notice how much is the state-space reduced with this simple and computably easy method.

```

procedure forward_checking(cv)
  queue = {(Vi,Vcv) in arcs(G),i>cv}
  consistent = true
  while not queue empty AND consistent
    select and delete any arc (Vk,Vm) from queue
    if revise(Vk,Vm) then
      consistent = not Dk empty
  return consistent

```

Listing 2.4: Forward Checking



(a) Without Forward Checking.

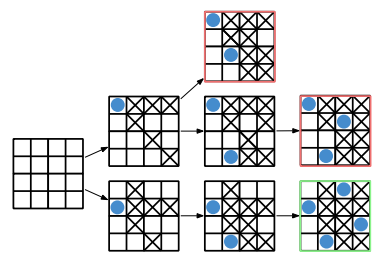
(b) With Forward Checking.
Pruned positions depicted as x.

Figure 2.2: Four queens problem state-space. The red square means unfeasible solution and the green means correct solution.

2.2.5 Variable Ordering

A performance of an algorithm for solving CSP highly depends on the order, in which the variables are taken, thus by choosing good heuristic for variable ordering, we can rapidly improve performance of our algorithm. There are two types of ordering methods:

- A static ordering, where the order of variables is specified before the search algorithm starts and is never changed.
- A dynamic ordering, in which the ordering is specified during the search process.

2.2.5.1 First-Fail Principle

A basic idea how to select a next variable in the search process is based on the first-fail principle, which tries to fail as soon as possible if a branch leading to infeasible solution is searched. In each state, we pick the variable with the fewest possibilities remaining dynamically. It is based on the assumption, that if all values are equally likely to be presented in the solution, then more values there are, the higher probability of the variable to be successfully binded. Simple analysis of this heuristic shows that:

- If the current partial solution (branch) does not lead to the correct solution, then the sooner we realise that the better.
- If the current branch leads to a correct solution, then the variable with the smallest domain has the smallest probability to be successful hence we will probably discover the failure sooner.

Thus this heuristics reduces the state-space of CSP by discovering the failure sooner and we can expect reduced average depth of branches.

2.2.5.2 Most Constrained Variable

Another heuristic with similar idea is to choose those variables, which are the most constrained. This is especially straightforward heuristic for reasoning in logic programs and will be used in our implementation, which will be described in Chapter 5.

2.2.6 Branch and Bound

When solving an NP-hard problem, explicit enumeration of all solutions is infeasible, because the number of candidate solution grows exponentially. **BB** is one of the most used algorithms for solving such difficult problems (Clausen [6]). Let us have an optimization task T with the payoff function f and the set of solutions S . If we want to use **BB**, we have to be able

- to create subtasks T_1, T_2, \dots, T_n with the same payoff function such as $S(T) = S(T_1) \cup S(T_2) \cup \dots \cup S(T_n)$ and compute the optimal value for subtask T_n or show, that the optimal value does not exist or is worse than the so far optimal value;
- to compute lower and upper bound for the optimal value within S_i . The lower bound is the best solution so far and the upper bound is the optimistic estimation of the current solution, i. e., the best value, which we can get in the current state.

Then the algorithm is pretty straightforward, because it only constructs the tree of solutions, computing the lower and upper bound for each created node and if the upper bound is lower than the lower bound, we can skip the current node.

Example 2.2.1 (Branch and Bound).

Figure 2.3 shows, how the algorithm works with following backpack problem (Demel [8]). Let us have a backpack with capacity $C = 50$ and five objects. Their weights and prices are shown in Table 2.3.

- Upper bound estimation is a sum of all possible objects in descending order in a cost/weight ratio. E. g., the upper bound for the first node is $50 * 1 + 26/30 * 107 = 142.73$. The value has to be an integer thus the result is 142.
- Branching is done by creating two subtasks, where the first will add an object i to the backpack and the second subtask will not add the object to the backpack. When creating a new node, its upperbound $_i$ is computed.
- Each state needs to be
 - closed if the solution is not feasible,
 - pruned if some node has better upperbound or
 - branched otherwise.
- The lowerbound here is the best feasible value from other nodes.

OBJECT i	A	B	C	D	E
COST c_i	50	107	31	31	69
WEIGHT w_i	14	30	9	10	27

Table 2.3: Backpack problem specification. Objects are sorted according to cost/weight ratio in descending order.

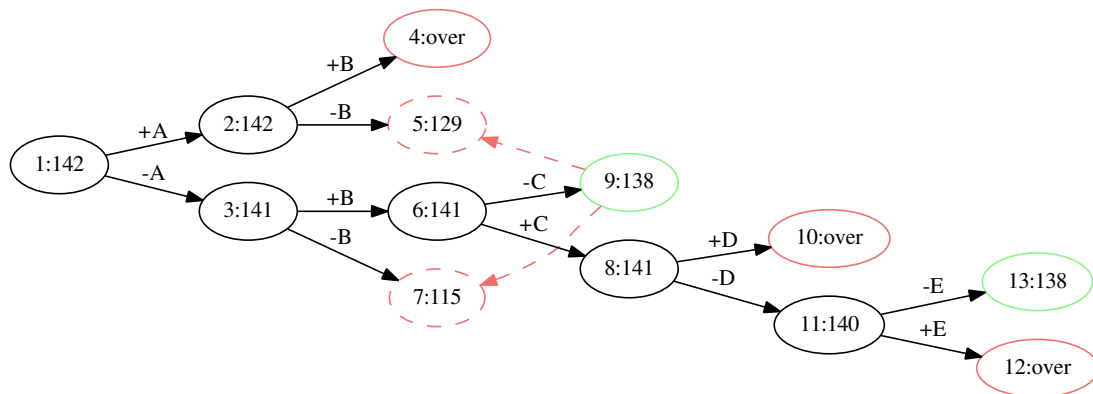


Figure 2.3: Branch and Bound example. The state consists from the ID : upperbound. Edges shows if the object i will be added (+) or deprecated (-). Green states are optimal and red states are infeasible. Red dashed states are closed (pruned) by the result from better node, which is connected with dashed backward line.

2.2.7 Restarted Strategy

A restarted strategy is a method widely used in various algorithms where randomization¹ is used (Baptista and Marques-Silva [3]). Randomness affects selected variable (in the sense of CSP), thus it can change the depth of branches.

The basic idea is to set a cutoff, where the algorithm will be terminated and then the best result so far is saved. Then the algorithm is started again from scratch. This can be repeated several times and in the end, the best result from all restarts is returned.

The cutoff value can be

- fixed, where the algorithm will stop after given number of backtracks or
- dynamic, where the number of backtracks is adjusted during the computation.

2.3 ARTIFICIAL NEURAL NETWORKS

ANN tries to mimic a real neural network by highly simplified mathematical model. It creates another approach for solving difficult problems than classic algorithms. This non-algorithmic approach belongs to the connectionist field of AI, which models problems as networks composed from simple units. The network learns its parameters and tries to adapt the network according to provided examples. ANNs are very widely used in pattern recognition, compression, control tasks etc. Nowadays, with great boom of parallel computing it has big advantage, because this approach is natively parallel thus can be implemented very fast.

The most widely used ANNs have two phases,

- a learning phase, where a number of examples is given and the network adapts its parameters to these examples, and
- an evaluation phase, where the network produces some information according to learned parameters.

2.3.1 Feed-forward Architecture

As we said, ANNs try to mimic simplified mathematical model of real neural network, where the basic element is neuron. Model of biological neuron is highly

¹ With randomization we mean the algorithm using random during its computation.

complicated, thus we use a simplified version, which is depicted in Figure 2.4. Simplified neuron has several parts:

- n independent inputs, with modifiable weight w_i , where i is the index of the input,
- transfer function, which sums all weighted inputs together,
- activation function with threshold, which produces output of neuron. Activation function must be differentiable and the most common one is a sigmoid.

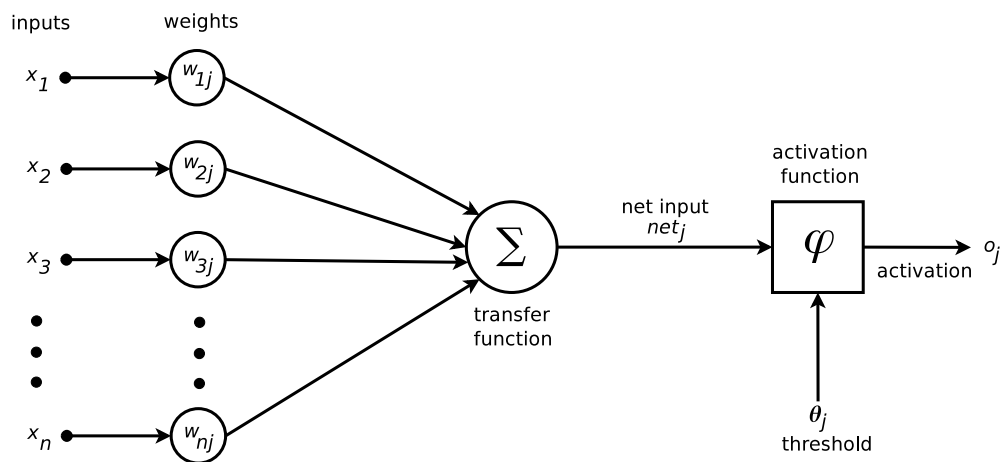


Figure 2.4: Simplified version of model of neuron. The inputs are weighted and then summed to the result. The activation function is applied on this result.

The way, how the network is composed from neurons i. e., how they are connected, is not always the same. Here we will describe just the type of ANN architecture, which is related to our model – feed-forward architecture.

ANN with this architecture has one input layer with i neurons, where i is the number of features from an example. Then it has one output layer with o neurons, where o is a size needed for our style of output representation. This architecture also can have zero or more hidden layers, each with independent count of neurons. Only neurons between two adjacent layers are fully connected in the forward manner, which means direction from input to output. The other type of connections are forbidden.

2.3.2 Learning of Artificial Neural Network

To learn an ANN, we need to modify its parameters, which are the weights of particular neurons. We want to adjust the weights, in order to minimize total error of the network on the current sample. One of the most often used algorithms for

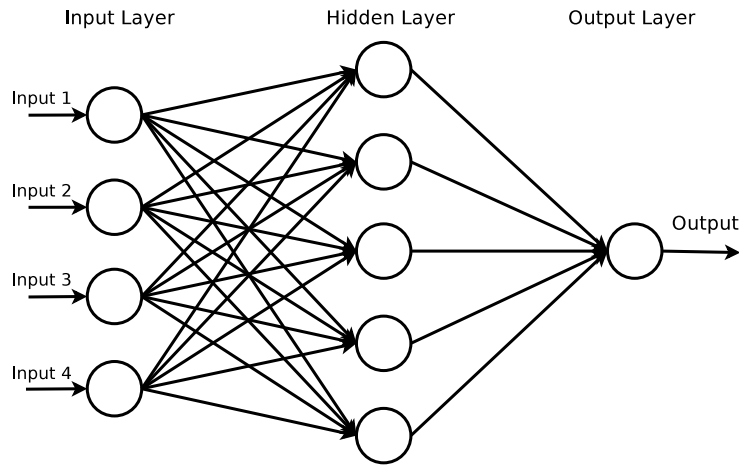


Figure 2.5: Feed-forward artificial neural network with four inputs, five neurons in one hidden layer and one output neuron.

learning parameters of an ANN is called backpropagation and has many variations. The next paragraph will briefly describe basic version of backpropagation.

The aim of the backpropagation algorithm is to minimize total error (sometimes called energy), which is shown in Equation (2.9). Equation (2.10) shows the error for one particular sample, where

- d_i^o is desired output i of the network for the sample with N_o outputs and
- y_i^o is actual output i of the network for the sample with N_o outputs.

$$E_{\text{total}} = \sum_p E_p \quad (2.9)$$

$$E_p = \frac{1}{2} \cdot \sum_{i=1}^{N_o} (d_i^o - y_i^o)^2 \quad (2.10)$$

The last question is, how to modify weights w_{jk} , to minimize the error of the network. Here backpropagation uses gradient descent, where the weights are changed according to Equation (2.11). η is a learning parameter, which should be tuned according to given problem. It also can be set by some other technique, like Resilient Backpropagation (RPROP) (Riedmiller and Braun [41]) or similar (Riedmiller [40]), which modifies η on the fly, according to the current situation in gradient descent. These automatic methods can lead to better convergence in the ANN learning.

$$\Delta w_{jk} = -\eta \cdot \frac{\partial E}{\partial w_{jk}} \quad (2.11)$$

ON-LINE LEARNING

When the weights are changed immediately after determining new differences between new and old weights then we speak of so-called On-line Learning. According to [Russell et al. \[43\]](#), On-line Learning requires more iterations of learning algorithm than batch learning. The algorithm of on-line backpropagation is shown in Listing 2.5.

```

procedure backprop
  forever
    for instance in dataset
      compute output for instance
      compare outputs with desired values
      modify the weights with gradient descend
    if testing error < wanted
      return

```

Listing 2.5: On-line Backpropagation Learning

BATCH LEARNING

In Batch Learning, the weights are cumulated during the learning process and the update is performed after all samples were processed.

2.4 FUZZY LOGIC

Fuzzy Logic is one of many non-classical logics, more precisely, it is many-valued logic sometimes also described as probabilistic logic. In contrast to classical logic, where values are from the set $\{0, 1\}$, in fuzzy logic it is an interval $[0, 1]$. Thus we can express vague information, which is so common in real life. Fuzzy Logic is mainly used in Control Theory and Artificial Intelligence. Lofti Zadeh is regarded as the founder of Fuzzy Logic with paper [51].

There exist many operations in fuzzy logic (sometimes taken as separate logics) ([Klir and Yuan, DuBois and Prade \[19, 10\]](#)) but in our work, we took inspiration only from Łukasiewicz logic, concretely from its conjunction and disjunction. Thus we will describe only this logic.

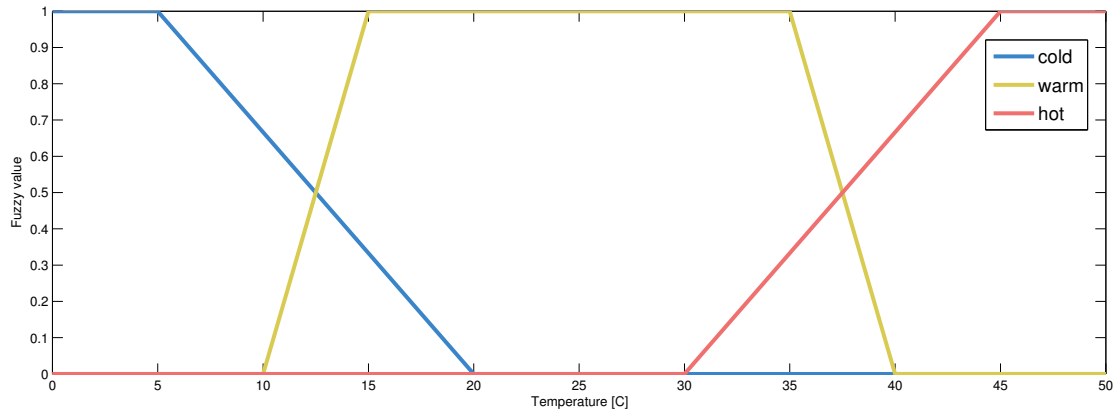


Figure 2.6: Fuzzy logic used for showing current temperature. For example temperature 15°C is taken as cold for 0.35, as warm for 0.65 and as hot for 0. In other words, we can say it is *cold a little* or *quite warm* or *definitely not hot*.

2.4.1 Łukasiewicz Logic

In general, Łukasiewicz logic (Giles [13]) is type of non-classical, many valued logic, which introduces two, for us, interesting operations. Łukasiewicz disjunction depicted in Equation (2.12) and conjunction expressed by Equation (2.13) where

- α and β are values from fuzzy logic i. e., from interval $[0, 1]$.

$$\alpha \underset{L}{\vee} \beta = \begin{cases} \alpha + \beta & \text{if } \alpha + \beta < 1, \\ 1 & \text{else.} \end{cases} \quad (2.12)$$

$$\alpha \underset{L}{\wedge} \beta = \begin{cases} \alpha + \beta - 1 & \text{if } \alpha + \beta - 1 > 0, \\ 0 & \text{else.} \end{cases} \quad (2.13)$$

3

STATE OF THE ART IN RELATED AREAS

This chapter will review the state of the art in the fields relevant to our model proposal and problems partly already mentioned in Chapter 2. It will serve as an inspiration and source of ideas, how to make our algorithm scalable.

3.1 DATALOG

We already mentioned the syntax of Datalog in Definition 2.1.11 from Section 2.1. Here, we will focus on Datalog more practically.

Datalog is a subset of Prolog and is mainly used for querying computer databases. The differences from Prolog are following (Wikipedia [50]):

- Functors are not allowed. For example, one cannot type the query shown in Equation (3.1), where `sonOf` is a functor, `married` a predicate and `john`, `ann`, `lucy` are constants.

$$? - \text{married}(\text{sonOf}(\text{john}, \text{ann}), \text{lucy}). \quad (3.1)$$

- Datalog restricts the usage of negation and recursion as follows.
 - Every variable that appears in the positive literal of a clause has to appear in some non-negated negative literal of a clause.
 - If the variable appears in the negated negative literal then it has to appear also in some non-negated negative literal.

3.1.1 Existing Datalog Solvers

Nowadays, there exist many reasoners for Datalog language but only a few of them are still actively developed, which is required for staying up to date with recent research findings. Most of current actively developed reasoners are university projects, which are used mainly for academic purposes. Here are the three well-known, up to date and still actively developed reasoners:

IRIS

IRIS is a Datalog reasoner with interface to Java language, which is still actively developed at University of Innsbruck by [IRIS Development Team](#) [16]. This project was the best candidate for modifying it to support our proposed language. It is written in Java, which can guarantee higher speed than in the case of high level languages and has very nice Application Interface (API). Unfortunately a code complexity of this reasoner was too high for patching it for our purposes.

BDDBDDDB

bddbddb was written by John Whaley during his Ph.D. studies at Stanford University. This work was source of several papers dealing mainly with connection between program analysis and Datalog queries [49, 24].

PYDATALOG

pyDatalog is the newest reasoner on the scene and tries to bring Datalog programming into python language. More details can be found in [38].

Besides these open-source licenced tools, there are various commercial products, which are not accessible for free and we were not able to try them.

3.1.2 Suitability for Algorithm

Existing reasoners described above are not suitable for our algorithm because of two main reasons:

1. **Language Differences:** The first reason is the lack of ability to specify weight for each rule or modify current reasoners to accept these weights. The modified syntax with added weights is show on 3.2.

$$[w] [\text{Datalog formula}] \quad (3.2)$$

Character w means a weight of the rule and has a huge impact on the algorithm for reasoning. Actually it completely changes the reasoning algorithm with all of its optimizations.

2. **Reasoner Differences:** All reasoners mentioned in section 3.1.1 ([Existing Datalog Solvers](#)) suffer from the fact, that Datalog language and its reasoners are designed and optimized for querying databases and thus try to find all possible occurrences. This property is useless for us (we need only one solution) and makes the computation unfeasible for large sets of rules and large ground knowledge. For example all of the reasoners failed for insufficiency

of memory when we tried to use them to reasoning about datasets from Chapter 6.

The conclusion about existing Datalog reasoners is that they are not applicable for our purposes. This means that we have to write our own simplified Datalog reasoner. Our simplified Datalog reasoner must

- work with weight extension in Datalog and
- be optimized for finding only one (best) solution.

3.2 θ -SUBSUMPTION ENGINES

We will study the most recent θ -subsumption engines in this section. Nowadays, there exist three competitive θ -subsumption engines. They are

- Django by [Maloberti and Sebag \[28\]](#),
- Subsumer by [Santos and Muggleton \[44\]](#) and
- Resumer by [Kuželka and Železný \[21\]](#).

The study of these engines gives us a guide, how to implement θ -subsumption effectively. Many ideas and optimizations are shared between all reasoners, thus we chose only the most successful one. The overall comparison of these engines on the Phase Transition dataset ([Giordana and Saitta \[14\]](#)) is taken from [Santos and Muggleton \[44\]](#) and depicted in Table 3.1.

Engine	Yes		Region		PT		Overall	
	CPU	RAM	CPU	RAM	CPU	RAM	CPU	RAM
Django	4,404	2,248	N/A	N/A	78,736	3,037	15,023	2,361
Resumer1	99	608	544	1,167	225	749	301	855
Resumer2	75	578	154	1,136	120	875	119	883
Subsumer	190	75	442	141	292	92	316	105

Table 3.1: Performance comparison between Django, Resumer1, Resumer2 and Subsumer. The results are summary of more detailed results from [Santos and Muggleton \[44\]](#) performed on the Phase Transition dataset. CPU times are in seconds and usage of RAM is in megabytes.

According to the results from Table 3.1, the winner is Resumer and we will focus on this reasoner in detail, because it outperformed remaining two engines and is

written in imperative language as well as our algorithm. Note that there are two Resumer1 in the table. Resumer2 is a version with enabled restart strategy and Resumer1 without this function.

3.2.1 Resumer2

Resumer2 can be considered as state of the art in the field of θ -subsumption ([Santos and Muggleton \[44\]](#)). We found the following features viable for an inspiration and applicable to our algorithm.

- Restarted strategy.
- Usage of CSP algorithms.
- Heuristic for variable ordering when performing substitution to the logic variables.
- Written in Java. Selection of typed programming language is important for an effective implementation. Also presence of some advanced data structures can be favourably utilized. For example caching possibilities in Java are very comfortable e. g., with weak hash maps.

4 | PROPOSED MODEL

We propose a novel machine learning model combining parts from [ILP](#) and [ANN](#). It tries to learn disjunction's weights in the logic program with imaging the logic program as a network and then performing the backpropagation algorithm. By this experimental model, we want to find out if an inspiration from [ANN](#) implemented into the environment of logic programming can be advantageous.

This chapter will completely describe the proposed model in the following sections.

- 4.1 We begin with a language used for model description. This language is called n-layered $\lambda\kappa$ -program and is mainly inspired by Datalog. Here, we describe all parts of this language including an extension of disjunctions with weight parameter.
- 4.2 Then we continue with a construction of network model called n-layered $\lambda\kappa$ -template. It transforms n-layered $\lambda\kappa$ -program into a network composed from two types of nodes corresponding to the literals. The network is composed from interleaving layers of two types.
- 4.3 When all nodes in n-layered $\lambda\kappa$ -template are grounded then we speak about n-layered $\lambda\kappa$ -network. This grounded structure is described in next section as well as its main elements – grounded nodes. Also an important term maximal substitution is described.
- 4.4 After that we provide two examples for better understanding terms specified so far and for providing an insight into model computation.
- 4.5 A learning algorithm is presented in the one of the last sections. One learning step is divided into two phases. In the first phase it transforms n-layered $\lambda\kappa$ -template to n-layered $\lambda\kappa$ -network by maximal substitution. After it a modified backpropagation algorithm is designed.
- 4.6 Finally an extension of the model is proposed to correctly perform a classification task.

4.1 $\lambda\kappa$ -PROGRAM

Definition 4.1.1 (n-layered $\lambda\kappa$ -program).

n-layered $\lambda\kappa$ -program is a tuple $(H_0, H_1, H_2, \dots, H_{n-1})$ of sets of function-free Horn clauses, extended with weights, satisfying the following constraints:

1. Let P be a predicate symbol. Let H_p^+ be the set of all clauses of the program, which contain a positive literal based on the predicate symbol P and, similarly, let H_p^- be the set of all clauses of the program, which contain a negative literal based on P . It must hold that if $C \in H_p^- \cap H_i$ and $C' \in H_p^+ \cap H_j$ then $j < i$ and $i + j$ is an odd number.
2. If $C \in H_{2k+1}$, P is the predicate symbol of the positive literal in C and if P appears in another clause $C' \in H_i$ then it must hold $i > 2k + 1$.
3. If $C \in H_{2k}$ then C has at most one negative literal. The predicate symbol of any negative literal in C can occur as positive literal in clause C' if for all sets H_{2k} such that $C' \in H_j$ it holds $j < 2k$ and $2k + j$ is an odd number.

A clause contained in a set H_{2k+1} (H_{2k}) where $k \in \mathbb{N}$ is called λ -clause (κ -clause). A literal is a λ -literal (κ -literal, respectively) if it appears as a positive literal in a λ -clause (κ -clause) or as a negative literal in a κ -clause (λ -clause).

Informally, a $\lambda\kappa$ -program corresponds to a Datalog program with weighted clauses, composed of two types of clauses: λ -clauses and κ -clauses. The literals in the bodies of λ -clauses can be only literals *defined* by some κ -clauses and vice versa. Moreover, the literals in the bodies of λ -clauses and κ -clauses must be defined by clauses contained in the sets of clauses (H_i) with lower indices (*i.e. in lower layers of the program*). This condition effectively forbids *recursion*. No two λ -clauses can have the same predicate symbol of the positive literal. Any κ -clause can have at most one negative literal. These last two conditions pose no problem for generality of the approach because any non-recursive Datalog program can be apparently transformed to a $\lambda\kappa$ -program as Example 4.1.1 illustrates.

Example 4.1.1.

Datalog program (Listing 4.1) can be transformed to $\lambda\kappa$ -program (Listing 4.2). Disjunction is emulated by two κ -clauses with weights $w = 1$ and auxiliary λ -literals. λ -literal α_1 emulates negative literals (body) of the first Datalog clause and α_2 negative literals of the second one.

```
a(X) :- b(X), c(Y), d(Z).
a(X) :- e(X), f(X).
```

Listing 4.1: Datalog Program

```
/* kappa-clauses */
a(X) :- a1(X).
a(X) :- a2(X).
/* lambda-clauses */
a1(X) :- b(X), c(Y), d(Z).
a2(X) :- e(X), f(X).
```

Listing 4.2: 3-layered $\lambda\kappa$ -program

Definition 4.1.2 (Join of two $\lambda\kappa$ -programs).

Let us have an n -layered $\lambda\kappa$ -program $P_1 = (H_0, H_1, \dots, H_{n-1})$ and an m -layered $\lambda\kappa$ -program $P_2 = (J_0, J_1, \dots, J_{m-1})$, where $m, n \in \mathbb{N}$. The join (\oplus) of these two $\lambda\kappa$ -programs is an o -layered $\lambda\kappa$ -program P in Equation (4.1), where $o = \max\{m, n\}$.

$$P = P_1 \oplus P_2 = \begin{cases} (H_0 \cup J_0, H_1 \cup J_1, \dots, H_{m-1} \cup J_{m-1}) & \text{if } m = n, \\ (H_0 \cup J_0, H_1 \cup J_1, \dots, H_{m-1} \cup J_{m-1}, \dots, H_{n-1}) & \text{if } m < n, \\ (H_0 \cup J_0, H_1 \cup J_1, \dots, H_{n-1} \cup J_{n-1}, \dots, H_{m-1}) & \text{if } m > n. \end{cases} \quad (4.1)$$

Definition 4.1.3 ($\lambda\kappa$ -sample).

$\lambda\kappa$ -sample is a 1-layered $\lambda\kappa$ -program $P = (H_0)$. H_0 is non-empty set containing only ground facts representing the sample.

Definition 4.1.4 ($\lambda\kappa$ -rules).

$\lambda\kappa$ -rules is an n -layered $\lambda\kappa$ -program $P = (H_0, H_1, \dots, H_{n-1})$ where $H_0 = \emptyset$ and $n > 1$.

4.2 $\lambda\kappa$ -TEMPLATE

Definition 4.2.1 (n -layered $\lambda\kappa$ -template).

n -layered $\lambda\kappa$ -template is a weighted directed acyclic graph $G = (V, E, c)$ corresponding to an n -layered $\lambda\kappa$ -program $P = (H_0, H_1, \dots, H_{n-1})$ as described below:

1. Let $L(K)$ be the set of all predicate symbols of λ -literals (κ -literals) in P . Then $V = L \cup K \cup V_{\text{bias}}$, where $V_{\text{bias}} = \{\forall_{i=0}^{n-2} v_i\}$.
2. $E = E_0 \cup E_{\text{bias}}$, where:
 - $E_0 = \{e = (x, y) \mid x \in H_p^+ \cap H_i \wedge y \in H_p^- \cap H_i\}$
 - $E_{\text{bias}} = \{e = (v_i, y) \mid y \in H_p^+ \cap H_i\}$

3. $c(e)$ is a weight function, which assign weight to every edge $e = (x, y)$:

$$c(e) = \begin{cases} 1.0 & \text{if } x \in K \wedge y \in L, \\ w^e & \text{if } x \in L \wedge y \in K, \\ q_0^y & \text{if } x \in V_{\text{bias}} \wedge y \in L, \\ w_0^y & \text{if } x \in V_{\text{bias}} \wedge y \in K. \end{cases}$$

where

- w^e is weight of κ -clause with positive literal y and negative literal x .
- w_0^y is bias for node representing κ -literal y .
- $q_0^y = -\text{deg}^+(y)$ is bias for node representing λ -literal y . The inspiration comes from approximation of fuzzy logic conjunction and disjunction by the sigmoid function.

A node $n \in L$ ($n \in K$) is called λ -node (κ -node) and corresponds to a λ -literal (κ -literal) in an n -layered $\lambda\kappa$ -program. All λ -nodes (κ -nodes) corresponding to the positive literals from the same sets H_{2k+1} (H_{2k}) create λ -layer $_{2k}$ (κ -layer $_{2k+1}$).

Informally, an n -layered $\lambda\kappa$ -template corresponds to a weighted directed graph constructed from an n -layered $\lambda\kappa$ -program. Graph nodes are all literals plus bias nodes and the edges are connections between negative and positive literals from the same clauses plus bias edges. This means, when the node n_l represents a λ -literal l (called λ -node) then all nodes representing negative κ -literals of λ -clause with a positive literal l are connected to n_l with weight 1. If the node n_k represents a κ -literal k (called κ -node) then all nodes representing λ -literals from bodies (negative literals) of all κ -clauses with literal k are connected to n_k with weight of corresponding κ -clause. Moreover, every node representing a positive literal has a bias. The bias for every κ -node representing a positive κ -literal is a real number, which can be changed in the learning process. The bias for λ -node is set in a static way and is computed as shown in Definition 4.2.1. This is a model with structure similar to ANN as one can see in Example 4.2.1.

Example 4.2.1 (5-layered $\lambda\kappa$ -template).

We can notice several things from the example of 5-layered $\lambda\kappa$ -network depicted in Figure 4.1.

- This particular network has two hidden λ -layers and one hidden κ -layer.
- The λ -clauses create black connections, which cannot be learnt and emulate conjunctions.
- The κ -clauses create blue connections, which can be learnt and emulate disjunctions.

- Each node from one layer is connected to each node from an adjacent layer. Note that this is not required. Some connections can be dropped, as well as connections between non-adjacent layers are permitted, but still only between two different types of layers.
- The learning ability consists in adjusting the weights of blue lines. The learning phase will be described in Section 4.5.

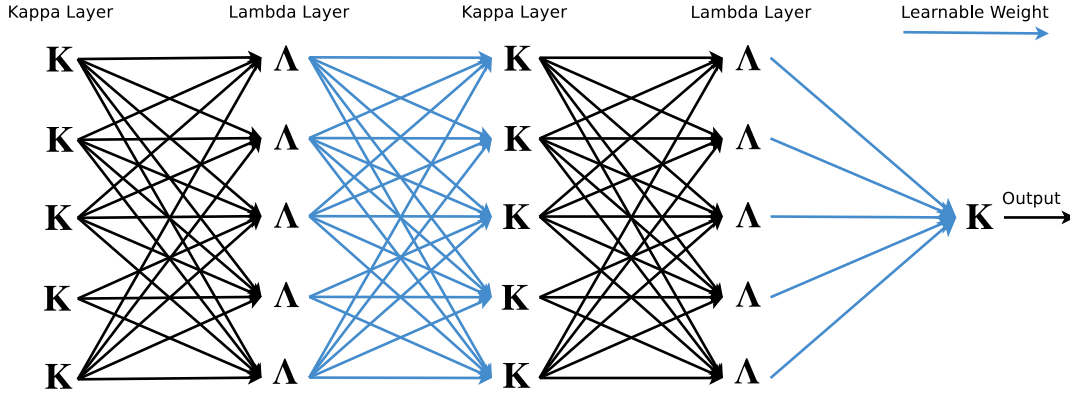


Figure 4.1: Proposed n -layered $\lambda\kappa$ -template. κ -nodes are changing with λ -nodes and each type of node has its own type of connection. Weights of black connections cannot learn and they represent conjunctions in the clause. Otherwise weights of blue connections can learn and simulate disjunctions. The first layer is composed from κ -nodes, which are the elementary predicates contained in the examples.

4.3 $\lambda\kappa$ -NETWORK

Definition 4.3.1 (n -layered $\lambda\kappa$ -network).

n -layered $\lambda\kappa$ -network is a fully grounded n -layered $\lambda\kappa$ -template. The nodes in an n -layered $\lambda\kappa$ -network are called grounded λ -nodes (κ -nodes).

Definition 4.3.2 (Ground node output (n_{out})).

Let f_{κ} (g_{κ}) be nondecreasing differentiable real function with finite limits in $-\infty$ and ∞ . Function f_{κ} (g_{κ}) is an activation function for κ -layer $_{2k}$ (λ -layer $_{2k+1}$). Further, let $in(n_{2k})$ ($in(n_{2k+1})$) denote the set of all ground nodes x connected to ground

node n_{2k} (n_{2k+1}) with edge $e = (x, n_{2k})$ ($e = (x, n_{2k+1})$). Then the output of a ground node n from layer $2k$ ($2k + 1$) is:

$$\text{nout}(n_{2k}) = \begin{cases} 1 & \text{if } k = 0, \\ 0 & \text{if } \forall n \in \text{in}(n_{2k}) \text{ output}(n) = 0, \\ f_k \left(w_0^{n_{2k}} + \sum_{x \in \text{in}(n_{2k})} w^e \cdot \text{output}(x) \right) & \text{else.} \end{cases} \quad (4.2)$$

$$\text{nout}(n_{2k+1}) = \begin{cases} 0 & \text{if } \exists n \in \text{in}(n_{2k+1}) \text{ output}(n) = 0, \\ g_k \left(q_0^{n_{2k+1}} + \sum_{x \in \text{in}(n_{2k+1})} \text{output}(x) \right) & \text{else.} \end{cases} \quad (4.3)$$

If we apply the nout function to the node N_{out} representing a positive literal from layer $n - 1$ (last layer) in an n -layered $\lambda\kappa$ -network M , we recursively compute the output (out) of M : $\text{out}(M) = \text{nout}(N_{\text{out}})$. Note that $w_0^{n_{2k}}$ ($q_0^{n_{2k+1}}$) means the bias for node n_{2k} (n_{2k+1}) and not the exponents.

Definition 4.3.3 (Maximal substitution (max)).

The maximal substitution (max) represents groundings for all nodes in n -layered $\lambda\kappa$ -template (literals in n -layered $\lambda\kappa$ -program) such the resulting n -layered $\lambda\kappa$ -network M has maximal output $\text{out}(M)$ among all possible groundings.

Definition 4.3.4 (n -layered $\lambda\kappa$ -program output (out)).

Let P be an n -layered $\lambda\kappa$ -program and M a corresponding n -layered $\lambda\kappa$ -network having the maximal substitution property. Then the output $\text{out}(P) = \text{out}(M)$.

The definitions above say that an n -layered $\lambda\kappa$ -template represents a *template* for an n -layered $\lambda\kappa$ -network, where *all* nodes are fully grounded. This is the solution of an n -layered $\lambda\kappa$ -program and one can imagine it as a structure similar to proof tree from logical programming (Flach [11]). The output of each ground node in an n -layered $\lambda\kappa$ -network emulates output of an artificial neuron (Figure 4.2). Moreover, the property of logical conjunction (disjunction) is preserved by the first condition in Equation (4.2) (Equation (4.3) eventually). κ -nodes from layer 0 represents ground facts and the ground fact's output is always 1. With this knowledge, we can compute the output of an n -layered $\lambda\kappa$ -network representing the solution of an n -layered $\lambda\kappa$ -program. The solution with the highest value for given n -layered $\lambda\kappa$ -program is called *maximal substitution* and the algorithm for computing it is in Listing 4.3.

```
procedure lambda_maximal_substitution(lambda-node N)
  best = 0
  for all possible substitutions S in N
    current = 0
    for all nodes M connected with N with respect to S
      if M is contained in the example
        current += 1
      elif M is not contained in the example
        current += 0
      else (M is node with another connections)
        current += kappa_maximal_substitution(M)

    current = activation_function(current)
    if current > best
      best = current

  return best

procedure kappa_maximal_substitution(kappa-node N)
  best = 0
  for all possible substitutions S in N
    current = 0
    for all nodes M connected with N with respect to S
      current += lambda_maximal_substitution(M) * weight(M,N)

    current = activation_function(current)
    if current > best
      best = current

  return best
```

Listing 4.3: Maximal Substitution

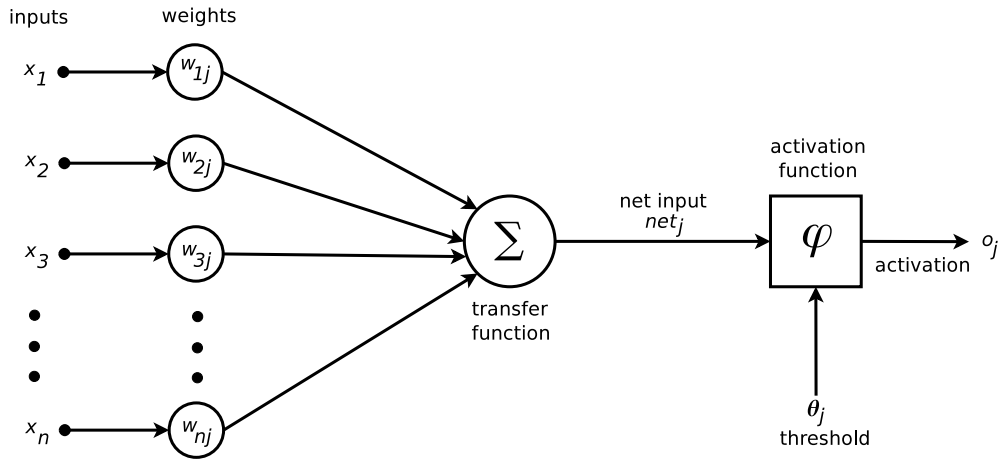


Figure 4.2: A node, which is an equivalent to ANN neurons. The inputs to the node have to be nodes of different type. If the node is κ -node then the weights $\mathbf{w} \in \mathbb{R}^n$. For λ -nodes, the weights are equal to one. Inputs are weighted and then summed to the result. The activation function is applied on this result.

4.4 TWO EXAMPLES

In this section, we will provide two examples for better understanding the terms described so far e. g., n -layered $\lambda\kappa$ -program, n -layered $\lambda\kappa$ -template, n -layered $\lambda\kappa$ -network, maximal substitution etc. For each example we provide listing with an n -layered $\lambda\kappa$ -program and figures with generated n -layered $\lambda\kappa$ -template and n -layered $\lambda\kappa$ -network.

4.4.1 First Example

The first example should mainly demonstrate the term maximal substitution. This demonstration is done on the following simple example. Let us have a graph depicted in Figure 4.3. Each node has different color and the task is to find a path from the node 1 to the node 4 through colored nodes, which contains the biggest fraction of the blue color.

A 5-layered $\lambda\kappa$ -program describing the situation is in Listing 4.4. The program is divided into five parts separated by a blank line. Note that the first one can be separately considered as a $\lambda\kappa$ -sample and other lines describe $\lambda\kappa$ -rules.

1. Description of a graph depicted in Figure 4.3 i. e., edges between nodes and nodes colors.
2. λ -clauses with connection to the literals presented in the sample.

3. κ -clauses defining the intensity of the blue color in each node.
4. λ -clause with definition of the most blue path from node 1 to node 4.
5. κ -clause for the output.

A structural representation of the 5-layered $\lambda\kappa$ -program mentioned in the previous paragraph is depicted in Figure 4.4 in the form of a 5-layered $\lambda\kappa$ -template. Then it is fully grounded to 5-layered $\lambda\kappa$ -program (Figure 4.5). Remind that the grounding is holding maximal substitution definition. We can see, that this graph contains the solution for given problem. It found a correct path 1 – 6 – 5 – 4 with output value computed by Equation (4.4), where f_i represents an activation function for i -th layer and M is given 5-layered $\lambda\kappa$ -program.

$$\text{out}(M) = f_4(g_3(f_2(g_1(1) \cdot 0.8 + g_1(1) \cdot 0.9)) \cdot 1.0) \quad (4.4)$$

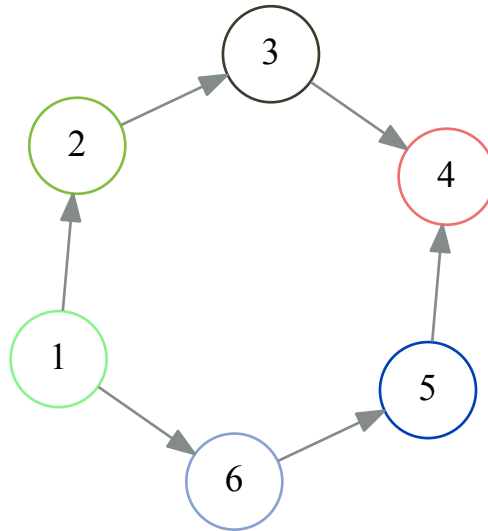


Figure 4.3: Directed graph with colored nodes. The task is to find a path from node number 1 to node number 4 such as the colors of visited nodes will be as blue as possible.

```
/* kappa-clauses */
edge(1,6),edge(6,5),edge(5,4),edge(1,2),edge(2,3),edge(3,4), light_green(1),
    dark_green(2),dark(3),red(4),dark_blue(5),light_blue(6).

/* lambda-clauses */
lg(X) :- light_green(X).
dg(X) :- dark_green(X).
d(X) :- dark(X).
r(X) :- red(X).
db(X) :- dark_blue(X).
lb(X) :- light_blue(X).

/* kappa-clauses */
0.4 blue(X) :- lg(X).
0.9 blue(X) :- db(X).
0.6 blue(X) :- dg(X).
0.7 blue(X) :- d(X).
0.5 blue(X) :- r(X).
0.8 blue(X) :- lb(X).

/* lambda-clauses */
most_blue_path :- edge(1,B), edge(B,C), edge(C,4), blue(B), blue(C).

/* kappa-clause */
1.0 output :- most_blue_path.
```

Listing 4.4: $\lambda\kappa$ -program for the Most Blue Path Problem

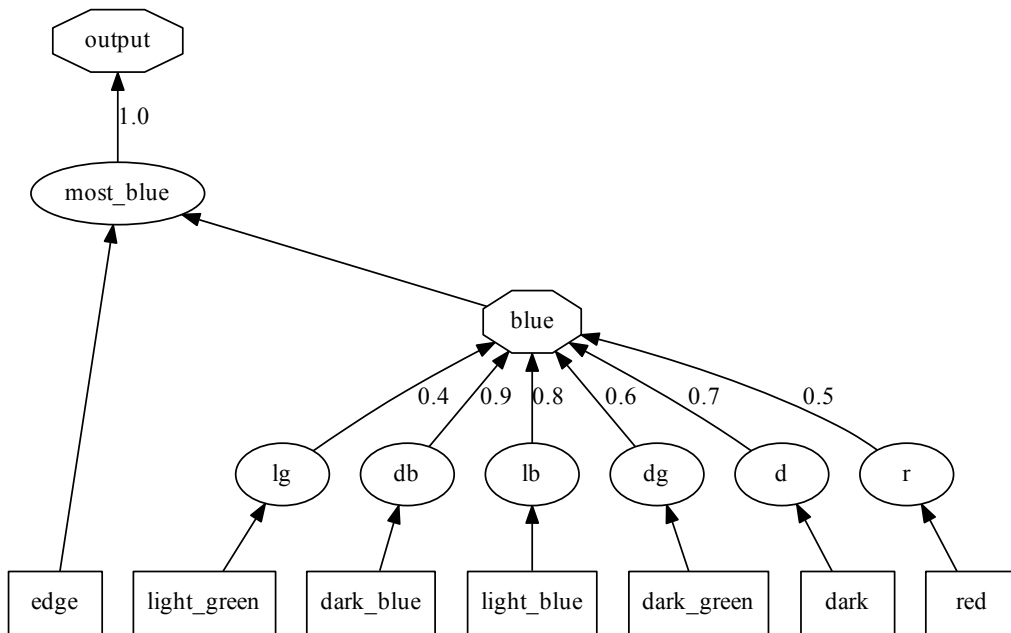


Figure 4.4: 5-layered $\lambda\kappa$ -template for the most blue path problem. Boxes are nodes representing the literals from $\lambda\kappa$ -sample, ovals are λ -nodes and octagons are κ -nodes.

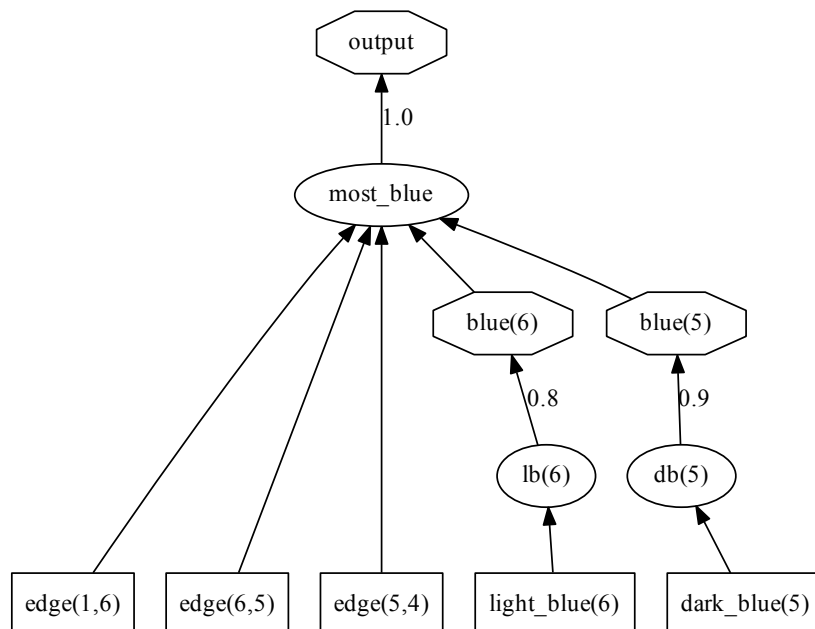


Figure 4.5: 5-layered $\lambda\kappa$ -network for the most blue path problem. Boxes are grounded nodes representing the literals from $\lambda\kappa$ -sample, ovals are grounded λ -nodes and octagons are grounded κ -nodes.

4.4.2 Second Example

The second example is fully artificial and comes from one of our unit tests. It should fully show syntactic possibilities of n -layered $\lambda\kappa$ -program (Listing 4.5). Again, the first part can be separately considered as $\lambda\kappa$ -sample and the rest as $\lambda\kappa$ -rules.

Then we can see 5-layered $\lambda\kappa$ -template (Figure 4.6) and 5-layered $\lambda\kappa$ -network (Figure 4.7), which are much more complicated than in the previous example. Here we can notice several things.

- Connections can be made also between non-adjacent layers. But always between layers of different types.
- One node from 5-layered $\lambda\kappa$ -template can have several groundings in the form of grounded nodes in a 5-layered $\lambda\kappa$ -network.
- 5-layered $\lambda\kappa$ -network as well as 5-layered $\lambda\kappa$ -template need not to be fully connected like feed-forward ANN. It is only on us how we construct the n -layered $\lambda\kappa$ -program but the more connections there are, higher chance to learn significant structures.

```
/* kappa-clauses */
bond(b,b), bond(a,b), bond(b,c), bond(c,a), bond(c,d), bond(c,e), bond(d,e),
  atom(a,c), atom(b,c), atom(c,c), atom(d,cl), atom(d,br).

/* lambda-clauses */
l21(X) :- atom(X,cl), bond(Z,Y).
l22(X) :- atom(X,br), atom(X,cl).
l23(X) :- atom(X,cl), bond(X,Y).
l24(X) :- bond(X,b).

/* kappa-clauses */
1.1 k21(X) :- l21(X).
2.3 k21(X) :- l24(X).
1.2 k22(X) :- l21(X).
1.3 k22(X) :- l22(X).
1.4 k22(X) :- l23(X).
1.5 k23(X) :- l23(X).
2.5 k24(X) :- l24(X).

/* lambda-clauses */
l11 :- k21(X), k22(Y), k21(Y).
l12 :- k21(a), k22(Y), k24(Z).
l13 :- k22(X), bond(X,Y).
l14 :- k23(X), bond(X,Y).

/* kappa-clauses */
0.1 k11 :- l11.
0.2 k11 :- l12.
0.3 k11 :- l13.
0.4 k11 :- l14.
```

Listing 4.5: More Complex $\lambda\kappa$ -program

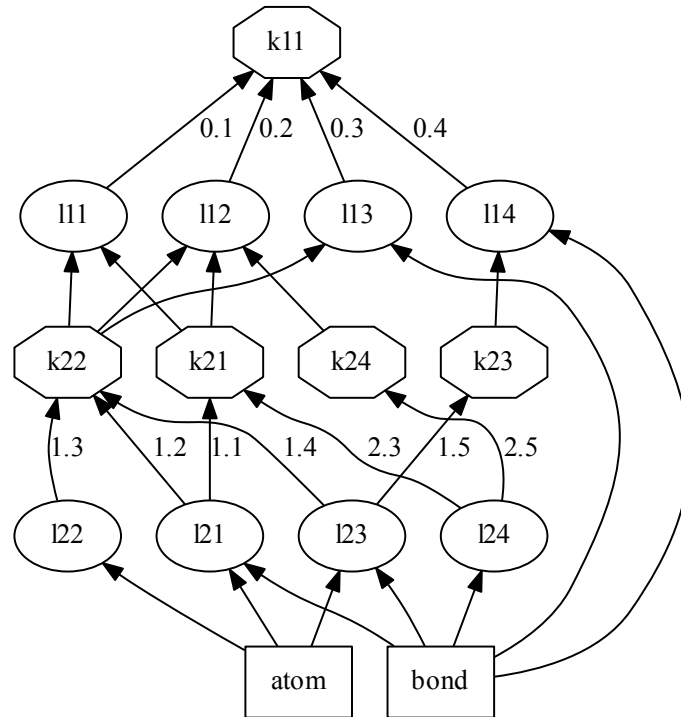


Figure 4.6: 5-layered $\lambda\kappa$ -template for more complex example. Boxes are nodes representing the literals from $\lambda\kappa$ -sample, ovals are λ -nodes and octagons are κ -nodes.

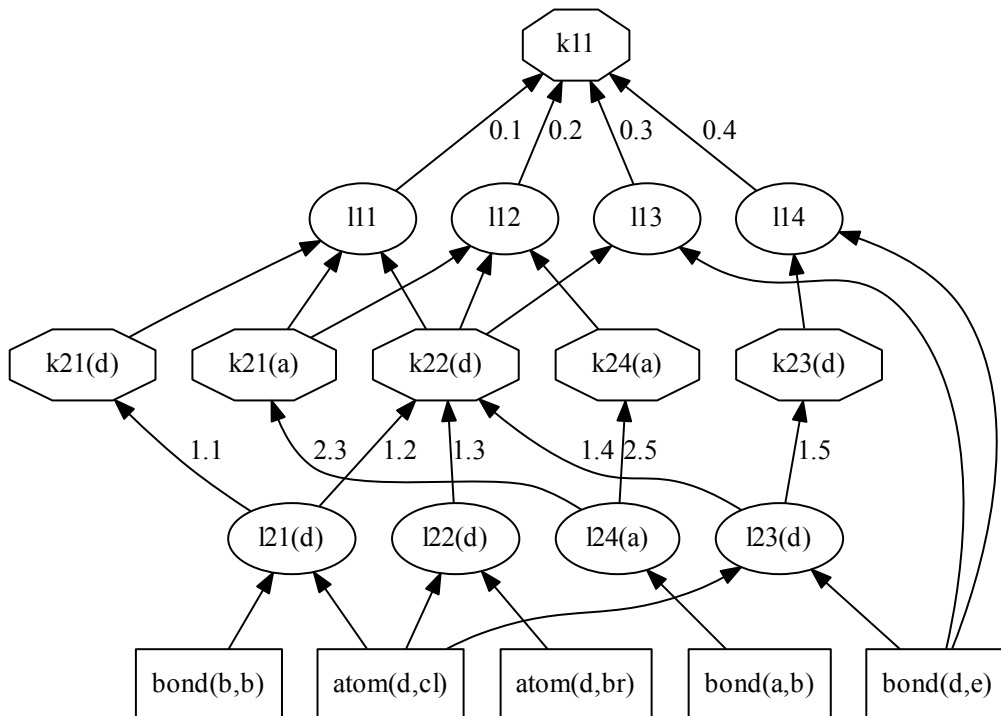


Figure 4.7: 5-layered $\lambda\kappa$ -network for more complex example. Boxes are grounded nodes representing the literals from $\lambda\kappa$ -sample, ovals are grounded λ -nodes and octagons are grounded κ -nodes.

4.5 LEARNING

Definition 4.5.1 ($\lambda\kappa$ -rules learning).

$\lambda\kappa$ -rules learning is searching for $\lambda\kappa$ -rules P^* from the set of all possible $\lambda\kappa$ -rules P with respect to Equation (4.5). E is a set of all training samples in the form of $\lambda\kappa$ -samples and $\text{ref}(e)$ denotes a referential output of sample e .

$$\arg \min_{P^* \in P} \sum_{e \in E} (\text{out}(P^* \oplus e) - \text{ref}(e))^2 \quad (4.5)$$

Informally, the aim of learning process is to learn weights of disjunctions in an n -layered $\lambda\kappa$ -program such that the difference between the reference output and the output of an n -layered $\lambda\kappa$ -program composed from $\lambda\kappa$ -rules and $\lambda\kappa$ -sample is minimal over all training samples. The situation is not simple as in the case of ANN (optimizing just weights) because of the logical nature of the model. Modified weights can cause a change of maximal substitution, thus updating maximal substitution during learning is needed. One learning algorithm (Listing 4.6) step is composed from two phases:

1. For given n -layered $\lambda\kappa$ -template find n -layered $\lambda\kappa$ -network with maximal substitution.
2. Learn weights in n -layered $\lambda\kappa$ -network created in step 1.

```

procedure learn
  forever
    lk-network = findMaximalSubstitution(lk-template)
    for given number of steps
      modifyWeightsWithBackprop(lk-network)

    updateWeights(lk-template, lk-network)

    if learningErr < wanted OR maximum steps reached
      return

```

Listing 4.6: Learning Algorithm

4.5.1 Finding Maximal Substitution

The algorithm for finding a maximal substitution was described in Section 4.3.

4.5.2 Gradient Descent

We propose a modified backpropagation algorithm known from ANN for learning weights of n -layered $\lambda\kappa$ -network. This algorithm performs a gradient descent on the cost function shown in Equation (4.6). Note that the gradient descent is performed on n -layered $\lambda\kappa$ -network composed from $\lambda\kappa$ -rules P and $\lambda\kappa$ -sample e . Modified weights from n -layered $\lambda\kappa$ -network are projected back to n -layered $\lambda\kappa$ -template. This also means, that weights missing in n -layered $\lambda\kappa$ -network are not updated in the corresponding n -layered $\lambda\kappa$ -template.

$$J = \frac{1}{2} \cdot (\text{ref}(e) - \text{out}(P \oplus e))^2 \quad (4.6)$$

Below we provide all elements needed for minimizing the cost function with modification of weights. For better readability of provided equations let us define following (note that upper indices are not exponents):

- κ_i^j (λ_i^j) means an output of i -th ground node from layer $j = 2k$ ($j = 2k + 1$).
- w_{mn}^j (v_{mn}^j) is a weight of edge between m -th and n -th node in j -th and $(j + 1)$ -th layer. If $m = 0$ than it is a bias.
- m_j is number of nodes in j -th layer

Let κ_0^{n-1} be the last κ -node in n -layered $\lambda\kappa$ -network ($P \oplus e$). Equation (4.7) displays partial derivative of cost function with respect to a weight w .

$$\begin{aligned} \frac{\partial J}{\partial w_{ab}^z} &= -2 \cdot (\text{ref}(e) - \text{out}(P \oplus e)) \cdot \frac{\partial \text{out}(P \oplus e)}{\partial w_{ab}^z} = \\ &= -2 \cdot (\text{ref}(e) - \text{out}(P \oplus e)) \cdot \frac{\partial \kappa_0^{n-1}}{\partial w_{ab}^z} \end{aligned} \quad (4.7)$$

We see that the next step is to compute partial derivatives of the κ_0^{n-1} output with respect to the given weight. Here, two options emerge.

1. One for computing partial derivative with respect to the bias weight i.e., we are modifying bias weight in current step. This is computed by Equations (4.8) to (4.10).
2. The second option is for derivation with respect to the weight from λ -node to κ -node. This can be computed by Equations (4.11) to (4.13).

PARTIAL DERIVATIVES WITH RESPECT TO BIAS WEIGHT

$$\frac{\partial \kappa_i^j}{\partial w_{0i}^{j-1}} = f_j' \left(w_{0i}^{j-1} + \sum_{a=1}^{m_{j-1}} w_{ai}^{j-1} \cdot \lambda_a^{j-1} \right) \quad (4.8)$$

$$\frac{\partial \kappa_i^j}{\partial w_{0b}^z} = f'_j \left(w_{0i}^{j-1} + \sum_{x=1}^{m_{j-1}} w_{xi}^{j-1} \cdot \lambda_x^{j-1} \right) \cdot \sum_{x=1}^{m_{j-1}} w_{xi}^{j-1} \cdot \frac{\partial \lambda_x^{j-1}}{\partial w_{0b}^z}, \quad z \neq j-1 \quad (4.9)$$

$$\frac{\partial \lambda_i^j}{\partial w_{0b}^z} = g'_j \left(v_{0i}^{j-1} + \sum_{x=1}^{m_{j-1}} \kappa_x^{j-1} \right) \cdot \sum_{x=1}^{m_{j-1}} \frac{\partial \kappa_x^{j-1}}{\partial w_{0b}^z}, \quad z \neq j-1 \quad (4.10)$$

PARTIAL DERIVATIVES WITH RESPECT TO WEIGHT BETWEEN NODES

$$\frac{\partial \kappa_i^j}{\partial w_{ki}^{j-1}} = f'_j \left(w_{0i}^{j-1} + \sum_{a=1}^{m_{j-1}} w_{ai}^{j-1} \cdot \lambda_a^{j-1} \right) \cdot \lambda_k^{j-1} \quad (4.11)$$

$$\frac{\partial \kappa_i^j}{\partial w_{ab}^z} = f'_j \left(w_{0i}^{j-1} + \sum_{x=1}^{m_{j-1}} w_{xi}^{j-1} \cdot \lambda_x^{j-1} \right) \cdot \sum_{x=1}^{m_{j-1}} w_{xi}^{j-1} \cdot \frac{\partial \lambda_x^{j-1}}{\partial w_{ab}^z}, \quad z \neq j-1 \quad (4.12)$$

$$\frac{\partial \lambda_i^j}{\partial w_{ab}^z} = g'_j \left(v_{0i}^{j-1} + \sum_{x=1}^{m_{j-1}} \kappa_x^{j-1} \right) \cdot \sum_{x=1}^{m_{j-1}} \frac{\partial \kappa_x^{j-1}}{\partial w_{ab}^z}, \quad z \neq j-1 \quad (4.13)$$

FINAL WEIGHT CHANGE

All these equations were needed for final computation of new weight differential shown in Equation (4.14). This update value is computed for each weight, which participates on the network output i.e., output value of corresponding κ -node is non-zero.

$$\Delta w_{jk}^i = -\eta \cdot \frac{\partial J}{\partial w_{jk}^i} \quad (4.14)$$

where

- η is a learning rate and the rest of notation was already introduced in description of equations above.

4.6 CLASSIFICATION NETWORK

Output value for an n -layered $\lambda\kappa$ -program ($P = R \oplus s$) composed of $\lambda\kappa$ -rules R and a $\lambda\kappa$ -sample s computed by our algorithm is a real number ($\text{out}(P) \rightarrow \mathcal{R}$). When dealing with a two-class classification problem, this value can be assigned the following meaning:

- The lower value we get the higher chance for classification to 0.
- The higher value we get the higher chance for classification to 1.

Thus, for classification, we also need a threshold th such that:

$$\begin{aligned} \text{classify as 1} & \quad \text{if } \text{out}(P) \geq \text{th} \\ \text{classify as 0} & \quad \text{if } \text{out}(P) < \text{th}. \end{aligned} \tag{4.15}$$

We propose a simple algorithm for finding threshold th , which works as follows:

1. Sort all training $\lambda\kappa$ -samples S with respect to the outputs $\text{out}(P = R \oplus s)$, where $s \in S$. This can be done in $O(n \cdot \log n)$, where $n = |S|$.
2. Select $\lambda\kappa$ -sample s' such that threshold $\text{th}' = \text{out}(P = R \oplus s')$ minimizes the error of classification function 4.15.
3. If the selected $\lambda\kappa$ -sample is the first or the last in the sorted sequence, then the final threshold is $\text{th} = \text{out}(P = R \oplus s')$ and the algorithm ends here.
4. With respect to the selected $\lambda\kappa$ -sample s we have two choices:
 - If $\text{ref}(s) = 1$, then pick $\lambda\kappa$ -sample s'' , which is just on the left in the sorted sequence.
 - Else ($\text{ref}(s) = 0$) pick $\lambda\kappa$ -sample s'' , which is just on the right in the sorted sequence.
5. The final threshold th is the mean between these two outputs as shown in 4.16. Let $\text{th}' = \text{out}(R \oplus s')$ and $\text{th}'' = \text{out}(R \oplus s'')$. A schematic picture is in Figure 4.8.

$$\text{th} = \min(\text{th}', \text{th}'') + \frac{|\text{th}' - \text{th}''|}{2} \tag{4.16}$$

4.6.1 Multi-criteria Classification

Our model is also usable for multi-criteria classification. The only difference between single and multi-criteria classification is in the number of κ -nodes in the

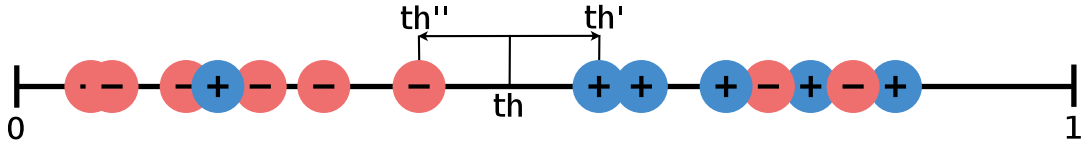


Figure 4.8: Determining the threshold th . Picture shows output values for samples depicted as balls (positive – blue, negative – red). The algorithm finds the best threshold (th') from $\lambda\kappa$ -sample's outputs. Then it locates its neighbour with opposite classification and output th'' . Finally the computed threshold th is in the middle between th' and th'' .

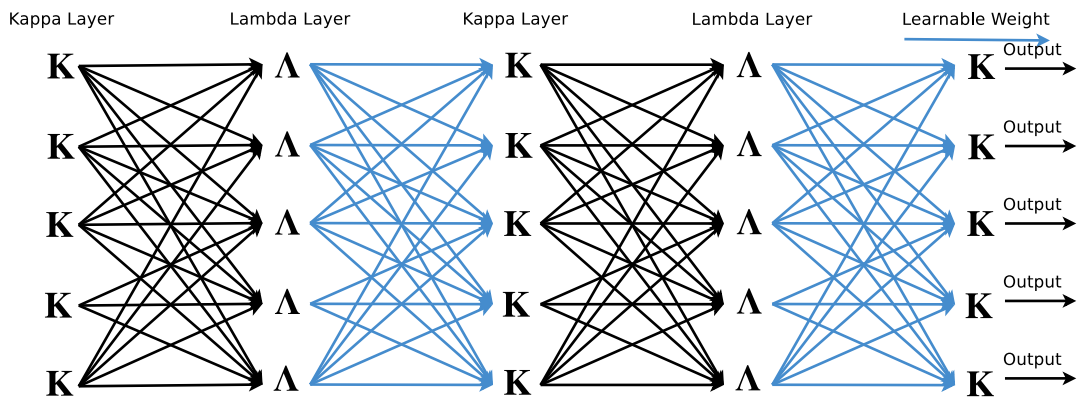


Figure 4.9: Figure that shows version of model for multi-criteria classification. The only difference is the output layer, where n kappas is presented. n is number of criteria.

output layer as known from [ANN](#). A network suitable for multi-criteria classification is in [Figure 4.9](#).

5 | PROPOSED ALGORITHM

This chapter describes implementation details of the proposed algorithm in three sections. In this chapter we omit *n-layered* before terms $\lambda\kappa$ -program, $\lambda\kappa$ -template and $\lambda\kappa$ -network because of space issues in algorithm schemas.

- 5.1 The first section describes an implementation of the learning algorithm. Each part of the algorithm is picked up and described in detail.
- 5.2 The second section describes a classification phase of the algorithm in a similar manner.
- 5.3 Finally, the main part of this chapter is about algorithm bottleneck detection and about designing improvements for speeding it up. This is necessary for making the proposed learning model scalable and ready for real experiments.

5.1 LEARNING PHASE

Schema of a learning phase used in our algorithm can be seen in Figure 5.1. The inputs to the learning algorithm are $\lambda\kappa$ -rules and $\lambda\kappa$ -sample. The output of learning algorithm is a trained $\lambda\kappa$ -rules with modified disjunction weights according to $\lambda\kappa$ -sample.

$\lambda\kappa$ -RULES \rightarrow $\lambda\kappa$ -TEMPLATE The algorithm of learning phase starts with translating $\lambda\kappa$ -rules into an internal structural representation. This is shown as transition $\lambda\kappa$ -program \rightarrow $\lambda\kappa$ -template. Internal representation provides an effective way, how to perform various operations e. g., determining the maximal substitution.

$\lambda\kappa$ -SAMPLE \rightarrow CHUNKS Similar operation with $\lambda\kappa$ -samples is then performed. Special structures representing the samples are created. These structures are called chunks. More about this structure granting effective queries is described later in Section 5.3.3.1.

$\lambda\kappa$ -TEMPLATE \rightarrow $\lambda\kappa$ -NETWORK $\lambda\kappa$ -network arises from $\lambda\kappa$ -template by grounding all nodes with respect to maximal substitution. In this phase, several $\lambda\kappa$ -

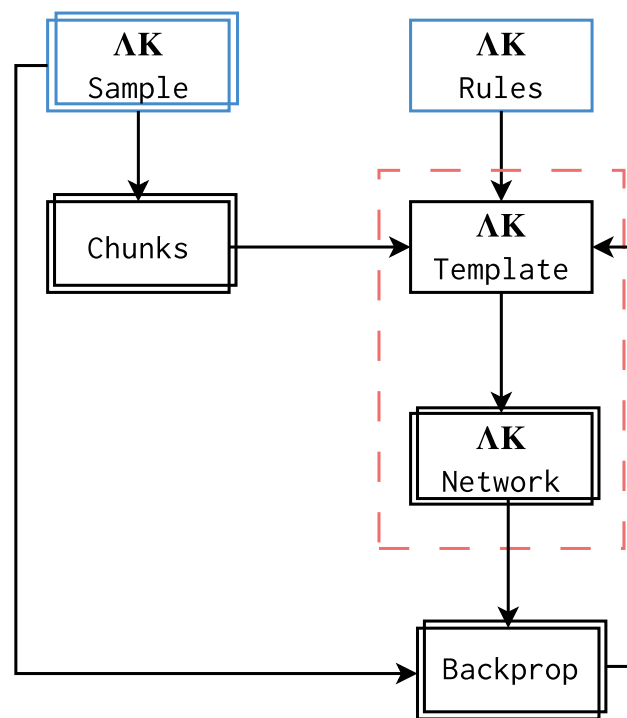


Figure 5.1: A learning schema. Blue rectangles means inputs into learning process and the red rectangle means the bottleneck of algorithm. Double squares represent multiple objects in that place. Lines show transition between phases or participation of phases.

networks emerge, for each $\lambda\kappa$ -sample joined with $\lambda\kappa$ -template one. This operation is the bottleneck of our algorithm and will be discussed in detail later (Section 5.3).

$\lambda\kappa$ -NETWORK \rightarrow BACKPROP Main part of learning is done in the next phase by a modified on-line backpropagation algorithm (Section 4.5). It is performed on each $\lambda\kappa$ -network composed in the previous phase. Modified weights are immediately updated in the $\lambda\kappa$ -template, which means, that weights in all other $\lambda\kappa$ -networks are updated too. Similarly to backpropagation from ANN several cycles are performed.

BACKPROP \rightarrow $\lambda\kappa$ -TEMPLATE After performing modified backpropagation to all $\lambda\kappa$ -networks the algorithm returns to the place depicted as $\lambda\kappa$ -template. At this moment, the transition $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -network is repeated but with new modified weights. The new weights often cause change of the maximal substitution. Several cycles of Backprop \rightarrow $\lambda\kappa$ -template are performed in the learning algorithm.

5.1.1 Threshold Learning

When the $\lambda\kappa$ -template is learnt well, we finish the learning process by threshold learning as described in Section 4.6.

5.2 CLASSIFICATION PHASE

A classification phase of our algorithm is depicted in Figure 5.2. The inputs to this phase are: (1) learnt $\lambda\kappa$ -rules, (2) $\lambda\kappa$ -sample, which we want to classify and (3) the output threshold. A classification into one of two classes is then performed.

All operations are performed in the same manner as in the learning case with one exception, the transition $\lambda\kappa$ -network \rightarrow classify.

$\lambda\kappa$ -NETWORK \rightarrow CLASSIFY The classification is done according to the $\lambda\kappa$ -network output computed with maximal substitution and comparing with output threshold.

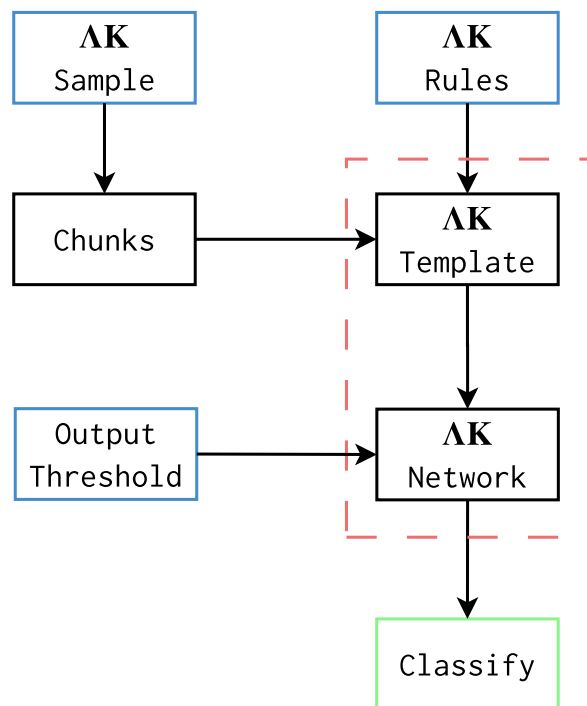


Figure 5.2: A classification schema. Blue rectangles means inputs into classification process and the red rectangle means the bottleneck of algorithm. Lines show transition between phases or participation of phases. The green rectangle shows the final phase.

5.3 EFFECTIVE IMPLEMENTATION DETAILS

In this section, we will provide an analysis of the learning and classification algorithms with the aim of identifying the biggest performance issues. Then we will design a basic approach how we can solve the bottleneck in a naive way, impractical for large instances (real data experiments). After it, we propose several improvements to the basic approach. These improvements make the algorithm fast enough to perform experiments on real data.

5.3.1 Biggest Bottleneck

Both, the classification as well as the learning algorithm can be implemented in a straightforward way with one exception. It is the transition $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -network. As we know, this transition requires computing of a maximal substitution for the $\lambda\kappa$ -template composed from $\lambda\kappa$ -sample and $\lambda\kappa$ -rules. This part of algorithm involves searching for a solution in a huge search-space and in addition, it is repeated several times. Thus it is the crucial part of proposed algorithms and we need to solve it in an effective way.

5.3.2 Basic Approach for Solving $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -network.

When solving a $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -program transition in a typical naive approach, the procedure is as follows. Take the output node representing κ -clause from output layer and start solving it in a typical recursive manner. This means to bind all variables from the clause and solve all clauses, which are connected to this node. This is done recursively until the input layer is reached and presence of grounded nodes from input layer is checked in the $\lambda\kappa$ -sample. This approach induces a few questions, which can affect speed of algorithm rapidly.

- In which order to take variables when grounding a given clause?
- How to skip a binding, which cannot lead to success?

Next sections are about answering these questions and designing a few optimizations, which speedup a basic approach for solving $\lambda\kappa$ -template \rightarrow $\lambda\kappa$ -network.

5.3.3 Sample Representation

One of the most often performed action in our algorithm is querying, whether a given literal presents in the $\lambda\kappa$ -sample. Thus this action should be fast and designing a special data structure is needed. Two types of queries can be performed.

1. Grounded query means that we want to know, whether a literal $p(a, b, c, d, e, f)$ is contained in the $\lambda\kappa$ -sample. This can be trivially satisfied by saving all literals from sample to some data structure with fast existential queries e.g., hash set.
2. Partially grounded query. This situation is much more interesting. If we want to find a literal $p(a, *, *, d, *, f)$ effectively, where $*$ means anything, then all possible combinations with $*$ must be enumerated in this structure. Unfortunately we can see a combinatorial explosion, when we save all possible combinations. Thus some suitable algorithm for solving this problem is needed.

5.3.3.1 Literal Partitioning

Every literal in the $\lambda\kappa$ -sample is partitioned into several smaller pieces with n original terminals. The number n can be adjusted according to given sample, but $n = 3$ seems to work well in most cases. These parts are then numbered and linked together with unique name of the literal. For the unique naming a hash map is constructed with mapping $\text{name} \rightarrow \text{id}$. Finally all parts are saved into a hash set.

Searching for some concrete literal is analogical to saving a literal, described in the paragraph above. The literal is divided into n numbered parts and all parts are checked if they are in the database. The searching is done for every id from possible ids in the hash map for given literal name.

Example 5.3.1.

Let's have an sample where literal $p(a, b, c, d, e, f)$ is presented. We will save chunks depicted in Table 5.1 into hash set instead of all combinations of original literal.

Number of all combinations for given literal can be computed as $\binom{6}{0} + \binom{6}{1} + \dots + \binom{6}{6} = 64$. But if we divide this literal into 2 parts, then the number of saved items is $2 \cdot \left(\binom{3}{0} + \binom{3}{1} + \binom{3}{2} + \binom{3}{3} \right) = 16$. A space complexity reduction is substantial and all possible combinations with $*$ are findable in a fast way with nothing like combinatorial explosion. This is particularly useful when forward checking algorithm asks if it should continue in variables binding with some already fixed variables. Forward checking algorithm for our model will be mentioned later (Section 5.3.5).

1 st part	2 nd part
$p_1(1, a, b, c)$	$p_1(2, d, e, f)$
$p_1(1, *, b, c)$	$p_1(2, *, e, f)$
$p_1(1, a, *, c)$	$p_1(2, d, *, f)$
$p_1(1, a, b, *)$	$p_1(2, d, e, *)$
$p_1(1, *, *, c)$	$p_1(2, *, *, f)$
$p_1(1, *, b, *)$	$p_1(2, *, e, *)$
$p_1(1, a, *, *)$	$p_1(2, d, *, *)$
$p_1(1, *, *, *)$	$p_1(2, *, *, *)$

Table 5.1: Partitioned literal $p(a, b, c, d, e, f)$. All parts are saved in one hash set. An id for literal p is here p_1 . If another literal with the same name would be added then it will have id p_2 etc. When searching for given literal p then all ids have to be processed.

5.3.4 Variable Ordering

As we have already said in Section 2.2.5 (Variable Ordering), a smart variable ordering can bring benefits in the form of speeding up a combinatorial algorithm. We chose simple algorithm inspired by Kuželka [23] for picking which variable should be grounded in every round.

1. Compute a score for each not bound variable in a given clause. This score expresses a number, how many variables are bound in shared literals. I. e., for all literals, where the computed variable presents, count all already bound variables. This is the score of the variable.
2. Select a variable with the highest score. If more variables have equal score then the one is picked uniformly.

This in fact means, that the most constraint variable is picked, thus the highest chance to fail early if we are not in the feasible branch, which is convenient as we mentioned earlier.

5.3.5 Forward Checking

A naive algorithm tests, whether a substitution succeeds after it binds all variables in all clauses recursively and the input layer with literals from $\lambda\kappa$ -sample are checked. This can lead to exploration of search-space, which cannot be feasible, because of one bad early bound variable. All bounds after this bad bound can-

not finish with success. E. g., if the first variable in some literal from $\lambda\kappa$ -sample is bound to a value and a literal with this value does not occur in the $\lambda\kappa$ -sample, then all other bindings are useless.

It is good to figure out, whether we bound a bad value immediately. Thus we do following check.

- After each bound, do a recursively check, if the partial query can succeed. This performs a partial query to the structure, which contains sample representation from previous section. If check fails, then we bind another value.

Here we see, why it was useful to create a data structure for effective partial queries. A sample of such query is $p(a, *, *)$. If we figure out that no such literal exists, we refuse this binding.

Forward checking results can be saved in a cache very well for later usage. Thus, if we figure out that literal $p(a, *, *)$ failed or succeeded we save it. Next identical check need not to be computed again.

5.3.6 Branch and Bound

When finding a maximal substitution in the clause, we can use **BB** for skipping bindings, which cannot get better score.

- Lower bound (l) for given clause is the best computed value so far, or $-\infty$ if no computation was done.
- Upper bound (u) is computed as

$$u = \sum_{\text{literal} \in \text{computed}} \text{literal} + \sum_{\text{literal} \in \text{not computed}} \text{estimate}(\text{literal})$$

$$\lambda \leftarrow \underbrace{\kappa_1, \kappa_2, \kappa_3}_{\substack{\text{Bound} \\ \text{Exact}}} \underbrace{\kappa_4, \kappa_5}_{\substack{\text{Unbound} \\ \text{Estimate}}} \quad (5.1)$$

where:

- The first sum is summing up the values of these literals, which are fully grounded and the output value is known.
- The second sum is summing up estimation for those literals, which are not fully grounded. This estimation has to overestimate the real value for keeping correctness because of pruning if $l \geq u$. An implementation of our algorithm has very simple estimation function, which sets an estimation of each literal to 1.

Biggest advantage of this really simple approach is its computational complexity, which outweighs more complex solutions in final. A schematic equation for three fully bound literals is in Equation (5.1).

5.3.7 Caching

When solving a maximal substitution, many computations can be performed repeatedly, because of the same grounded literal can occur in various clauses. Thus remembering already computed values for grounded literal is convenient. For this purpose, we have used a weak hash map, a very suitable data structure with automatic memory management. I. e., items from weak hash maps are automatically deleted whether a memory is running low. Usage of cache is shown in Listing 5.1 in a general way. Caching is not only useful for computing output values of literals, but already for results from forward checking or during the backpropagation algorithm.

```
procedure cached_solve(problem)
  if problem's output is in cache
    return problem's output
  else
    perform computation
    save result to cache
  return problem's output
```

Listing 5.1: Caching Results

6 | EXPERIMENTS

In this chapter, we are describing experiments with the proposed algorithm. It will be presented on datasets Mutagenesis (Lodhi and Muggleton [27]) and Predictive Toxicology Challenge (Helma et al. [15]).

- Mutagenesis dataset contains descriptions of 188 molecules labelled according to their mutagenicity.
- Predictive Toxicology Challenge dataset contains four datasets with labelled molecules according to their toxicity to female mice, male mice, female rats and male rats. We tested male rats as the most difficult dataset.

Results will be compared to state-of-the-art relational learning algorithms taken from Kuželka et al. [22], where the accuracies were estimated by 10-fold cross-validation. The state-of-the-art relational learners are:

- nFOIL (Landwehr et al. [25]), combining FOIL's (Quinlan and Cameron-Jones [39]) top-down approach for hypothesis search and Naive Bayes classifier.
- A novel bottom-up method based on Plotkin's least general generalization operator denoted as *Bottom* by Kuželka et al. [22].

6.1 $\lambda\kappa$ -PROGRAMS

$\lambda\kappa$ -programs for both datasets are in Listings 6.1 to 6.3. Notice that upper layers of $\lambda\kappa$ -programs are the same for both datasets.

2 bottom layers for each dataset are in Listing 6.1 (Listing 6.3). They contain bindings to the literals from samples and their propagation to higher layer with learnable weights. Note that already here, in bottom layers, fuzziness of our method appears and continues to upper layers.

```

atomLambda1(X) :- o(X).
atomLambda4(X) :- p(X).
atomLambda7(X) :- cu(X).
atomLambda10(X) :- c1(X).
atomLambda13(X) :- ca(X).
atomLambda16(X) :- zn(X).
atomLambda19(X) :- ba(X).

atomLambda2(X) :- n(X).
atomLambda5(X) :- sn(X).
atomLambda8(X) :- c(X).
atomLambda11(X) :- i(X).
atomLambda14(X) :- in(X).
atomLambda17(X) :- pb(X).

atomLambda3(X) :- f(X).
atomLambda6(X) :- br(X).
atomLambda9(X) :- k(X).
atomLambda12(X) :- na(X).
atomLambda15(X) :- h(X).
atomLambda18(X) :- s(X).

bondLambda1(X) :- 2(X).
bondLambda4(X) :- 1(X).

bondLambda2(X) :- 3(X).

bondLambda3(X) :- 7(X).

o.o bondKappa1(X) :- bondLambda1(X). o.o bondKappa1(X) :- bondLambda2(X). o.o bondKappa1(X) :- bondLambda3(X).
o.o bondKappa1(X) :- bondLambda4(X). o.o bondKappa2(X) :- bondLambda1(X). o.o bondKappa2(X) :- bondLambda2(X).
o.o bondKappa2(X) :- bondLambda3(X). o.o bondKappa2(X) :- bondLambda4(X). o.o bondKappa3(X) :- bondLambda1(X).
o.o bondKappa3(X) :- bondLambda2(X). o.o bondKappa3(X) :- bondLambda3(X). o.o bondKappa3(X) :- bondLambda4(X).

o.o atomKappa1(X) :- atomLambda1(X). o.o atomKappa1(X) :- atomLambda2(X). o.o atomKappa1(X) :- atomLambda3(X).
o.o atomKappa1(X) :- atomLambda4(X). o.o atomKappa1(X) :- atomLambda5(X). o.o atomKappa1(X) :- atomLambda6(X).
o.o atomKappa1(X) :- atomLambda7(X). o.o atomKappa1(X) :- atomLambda8(X). o.o atomKappa1(X) :- atomLambda9(X).
o.o atomKappa1(X) :- atomLambda10(X). o.o atomKappa1(X) :- atomLambda11(X). o.o atomKappa1(X) :- atomLambda12(X).
o.o atomKappa1(X) :- atomLambda13(X). o.o atomKappa1(X) :- atomLambda14(X). o.o atomKappa1(X) :- atomLambda15(X).
o.o atomKappa1(X) :- atomLambda16(X). o.o atomKappa1(X) :- atomLambda17(X). o.o atomKappa1(X) :- atomLambda18(X).
o.o atomKappa1(X) :- atomLambda19(X). o.o atomKappa2(X) :- atomLambda1(X). o.o atomKappa2(X) :- atomLambda2(X).
o.o atomKappa2(X) :- atomLambda3(X). o.o atomKappa2(X) :- atomLambda4(X). o.o atomKappa2(X) :- atomLambda5(X).
o.o atomKappa2(X) :- atomLambda6(X). o.o atomKappa2(X) :- atomLambda7(X). o.o atomKappa2(X) :- atomLambda8(X).
o.o atomKappa2(X) :- atomLambda9(X). o.o atomKappa2(X) :- atomLambda10(X). o.o atomKappa2(X) :- atomLambda11(X).
o.o atomKappa2(X) :- atomLambda12(X). o.o atomKappa2(X) :- atomLambda13(X). o.o atomKappa2(X) :- atomLambda14(X).
o.o atomKappa2(X) :- atomLambda15(X). o.o atomKappa2(X) :- atomLambda16(X). o.o atomKappa2(X) :- atomLambda17(X).
o.o atomKappa2(X) :- atomLambda18(X). o.o atomKappa2(X) :- atomLambda19(X). o.o atomKappa3(X) :- atomLambda1(X).
o.o atomKappa3(X) :- atomLambda2(X). o.o atomKappa3(X) :- atomLambda3(X). o.o atomKappa3(X) :- atomLambda4(X).
o.o atomKappa3(X) :- atomLambda5(X). o.o atomKappa3(X) :- atomLambda6(X). o.o atomKappa3(X) :- atomLambda7(X).
o.o atomKappa3(X) :- atomLambda8(X). o.o atomKappa3(X) :- atomLambda9(X). o.o atomKappa3(X) :- atomLambda10(X).
o.o atomKappa3(X) :- atomLambda11(X). o.o atomKappa3(X) :- atomLambda12(X). o.o atomKappa3(X) :- atomLambda13(X).
o.o atomKappa3(X) :- atomLambda14(X). o.o atomKappa3(X) :- atomLambda15(X). o.o atomKappa3(X) :- atomLambda16(X).
o.o atomKappa3(X) :- atomLambda17(X). o.o atomKappa3(X) :- atomLambda18(X). o.o atomKappa3(X) :- atomLambda19(X).

```

Listing 6.1: PTC-MR Bottom Layers

```

lambdao :- atomKappa1(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa1(B2), atomKappa1(Z).
lambda1 :- atomKappa1(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa2(B2), atomKappa1(Z).
lambda2 :- atomKappa1(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa3(B2), atomKappa1(Z).
lambda3 :- atomKappa1(X), bond(X,Y,B1), bondKappa2(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa1(B2), atomKappa1(Z).
.
.
.
lambda6 :- atomKappa1(X), bond(X,Y,B1), bondKappa3(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa1(B2), atomKappa1(Z).
.
.
.
lambda9 :- atomKappa1(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa1(B2), atomKappa2(Z).
lambda10 :- atomKappa1(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa2(B2), atomKappa2(Z).
.
.
.
lambda100 :- atomKappa2(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa2(B2), atomKappa3(Z).
lambda101 :- atomKappa2(X), bond(X,Y,B1), bondKappa1(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa3(B2), atomKappa3(Z).
lambda102 :- atomKappa2(X), bond(X,Y,B1), bondKappa2(B1), atomKappa1(Y), bond(Y,Z,B2), bondKappa1(B2), atomKappa3(Z).
.
.
.
lambda220 :- atomKappa3(X), bond(X,Y,B1), bondKappa2(B1), atomKappa3(Y), bond(Y,Z,B2), bondKappa2(B2), atomKappa1(Z).
.
.
.
lambda242 :- atomKappa3(X), bond(X,Y,B1), bondKappa3(B1), atomKappa3(Y), bond(Y,Z,B2), bondKappa3(B2), atomKappa3(Z).

o.o finalKappa :- lambdao.
o.o finalKappa :- lambda1.
.
.
.
o.o finalKappa :- lambda242.

```

Listing 6.2: Common Upper Layers

6.2 LEARNING ANALYSIS

During learning the network, we measured the learning error as well as a value called dispersion. Dispersion is the difference in the output of two training samples: one with the highest output and one with the lowest output. Figure 6.1 shows how dispersion changes during the learning algorithm and one can see, that our model separates the training samples more and more during the learning phase. This confirms the ability to learn. Figure 6.2 shows the learning error for the learning phase with 30 learning cycles. Notice the coherence between low learning error and high dispersion.

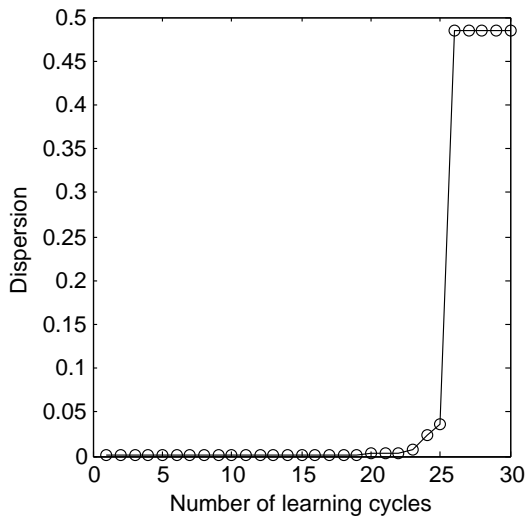


Figure 6.1: Dispersion for randomly selected experiment. The graph shows the difference between outputs of two samples: a sample with the highest output and a sample with the lowest output. The values are measured when a new maximal substitution for given $\lambda\kappa$ -template is found.

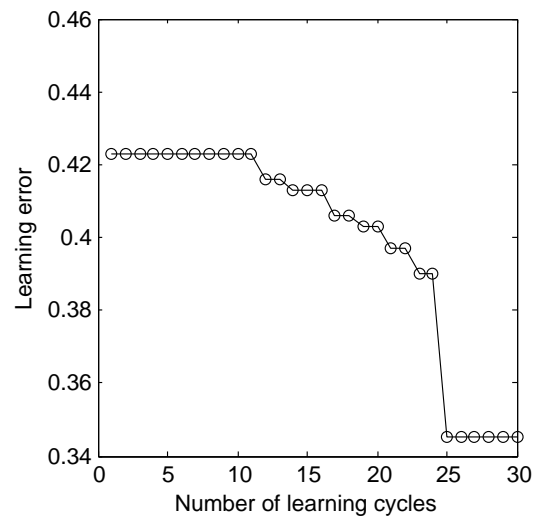


Figure 6.2: Learning error for randomly selected experiment. The graph shows the learning error. The values are measured when a new maximal substitution for given $\lambda\kappa$ -template is found.

6.3 RESULTS

The settings of experiments are in Table 6.1. *Learning cycles* denotes the number of accomplished maximal substitutions. During one learning cycle, several runs of

the modified backpropagation algorithm are executed. This number is expressed by *Learning steps* value. *Atom kappas* (*Bond kappas*) denotes the number of generated κ -literals for literals with predicate symbols *atom* (*bond*) in a hidden layer (H_2 in these 5-layered $\lambda\kappa$ -programs). These two parameters have a big impact on the generated n -layered $\lambda\kappa$ -program size (see examples of $\lambda\kappa$ -programs). Setting *Initial weights* was done in a random way, we set 90% of weights to 0.1 and the remaining 10% to 0.9. *Learning rate* was experimentally tuned to 0.02.

Each experiment was started 10 times and the final value is computed as a median of these runs. The median is shown in Table 6.2 and compared with results from Kuželka et al. [22]. Box plots of results are depicted in Figures 6.3 and 6.4.

Learning cycles	30
Learning steps	50
Atom kappas	3
Bond kappas	3
Initial weights	0.1[90%] 0.9[10%]
Learning rate	0.02

Table 6.1: Parameters of Algorithm

	PTC-MR	Muta
$\lambda\kappa$ -prg	59.6	75.0
nFoil	57.3	76.6
Bottom	62.2	78.9

Table 6.2: Accuracies

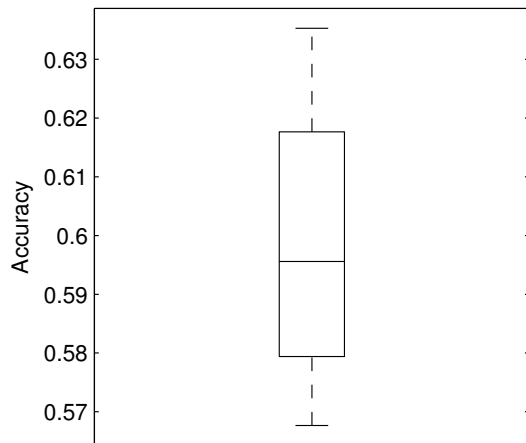


Figure 6.3: PTC-MR Accuracy

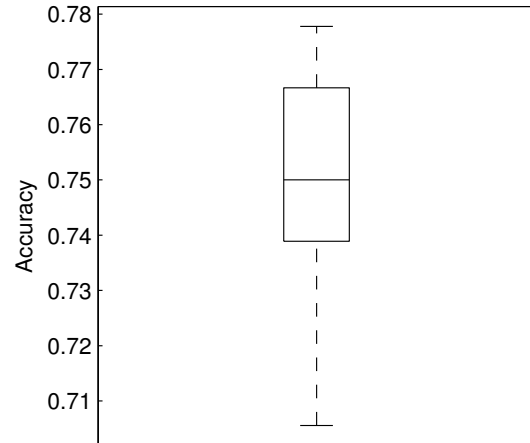


Figure 6.4: Mutagenesis Accuracy

7 | CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

The main motivation of this thesis was to develop an algorithm for [ILP](#), that would use fuzziness more than approaches used so far e.g., propositionalization with linear classifier. The algorithm will be able to learn helpful non-crisp concepts, which we call a predicate invention, because it can be used for definition of other concepts or final hypothesis.

We proposed a novel machine learning model, introduced theoretical foundations of this model, implemented a learning and classification algorithms and experimentally evaluated the accuracy of this model.

The theoretical part of this thesis dealt with setting theoretical foundations of the new model. We proposed a new language called n -layered $\lambda\kappa$ -program. It extends Datalog with weighted disjunctions and introduces constraints on its clauses. Thanks to these constraints, we could transform an n -layered $\lambda\kappa$ -program to an n -layered $\lambda\kappa$ -template. It represents the n -layered $\lambda\kappa$ -program as a weighted directed graph with nodes extended with function similar to an artificial neuron. After grounding the n -layered $\lambda\kappa$ -template with maximal substitution, we got an n -layered $\lambda\kappa$ -network. The output of the n -layered $\lambda\kappa$ -network we can compute. Based on these graph representations we developed a learning algorithm interleaving the logical part (maximal substitution) and the continuous part (modified backpropagation). Also a classification algorithm based on the output of the n -layered $\lambda\kappa$ -network (holding maximal substitution property) was proposed.

The implementation part has focused on an effective implementation of the model. During the implementation, we had to build an effective n -layered $\lambda\kappa$ -program reasoner from scratch. We had to integrate techniques such as forward checking, variable ordering, branch and bound, literal partitioning, emulated dynamic programming with caching etc. The scalability of the reasoner was crucial for performing experiments on real data. Moreover, the learning and classification algorithms were implemented.

The experiments were performed on two classical [ILP](#) datasets with satisfactory results. At one dataset (PTC-MR), our model beats the state-of-the art relational

learner *nFOIL* but was not good enough to beat novel method called *Bottom*. The second dataset (Mutagenesis) ends with worse results.

Considering the novelty of our approach, which means that many fine-tunings and other optimization will be done in the future, we can consider our work a successful foundation of new learning algorithm with a big potential for extensions, other improvements or even modifications during future research.

7.2 FUTURE WORK

As we have already said, the work on the proposed model does not end with this thesis. It will be further fine-tuned, modified and extended. Here, we provide some recent ideas for extensions.

STRUCTURE LEARNING

The highest potential performance boost can be achieved by adding the structure learning ability. It means, that clauses for n -layered $\lambda\kappa$ -program will not be generated for all combinations of literals, like so far, but will be chosen in a more sophisticated way. We can use existing tools for mining patterns from data and compose layers with their help.

λ -CLAUSES WEIGHTS

In the proposed model, we kept λ -clause weights at 1. Another possibility is to make λ -clauses weights learnable. This will make the learning process slower but it can also bring some accuracy improvements.

BIBLIOGRAPHY

- [1] C.R. Anderson, P. Domingos, and D.S. Weld. Relational markov models and their application to adaptive web navigation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 143–152. ACM, 2002. (Cited on page 2.)
- [2] F. Bacchus and P. Van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 311–318. JOHN WILEY & SONS LTD, 1998. (Cited on page 12.)
- [3] L. Baptista and J. Marques-Silva. Using randomization and learning to solve hard real-world instances of satisfiability. *Principles and Practice of Constraint Programming–CP 2000*, pages 489–494, 2000. (Cited on page 18.)
- [4] Roman Barták. On-line guide to constraint programming, 10 2012. URL <http://ktiml.mff.cuni.cz/~bartak/constraints>. (Cited on page 11.)
- [5] K.L. Clark. Negation as failure. *Logic and data bases*, 1:293–322, 1978. (Cited on page 6.)
- [6] J. Clausen. Branch and bound algorithms-principles and examples. 1999. (Cited on page 16.)
- [7] J. Cussens. Loglinear models for first-order probabilistic reasoning. In *Proceedings of the Fifteenth conference on Uncertainty in artificial intelligence*, pages 126–133. Morgan Kaufmann Publishers Inc., 1999. (Cited on page 2.)
- [8] J. Demel. *Graphs and their applications*. Czech, Academia Praha, 2002. (Cited on page 16.)
- [9] P. Domingos, S. Kok, H. Poon, M. Richardson, and P. Singla. Unifying logical and statistical ai. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 2–7, 2006. (Cited on page 2.)
- [10] D. DuBois and H.M. Prade. *Fuzzy sets and systems: theory and applications*, volume 144. Academic Pr, 1980. (Cited on page 21.)
- [11] P.A. Flach. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc., 1994. (Cited on page 32.)

- [12] L. Getoor, N. Friedman, D. Koller, and A. Pfeffer. Learning probabilistic relational models. *Relational data mining*, page 307, 2001. (Cited on page 2.)
- [13] R. Giles. Łukasiewicz logic and fuzzy set theory. *International Journal of Man-Machine Studies*, 8(3):313–327, 1976. (Cited on page 22.)
- [14] A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000. (Cited on page 25.)
- [15] C. Helma, R.D. King, S. Kramer, and A. Srinivasan. The predictive toxicology challenge 2000–2001. *Bioinformatics*, 17(1):107–108, 2001. (Cited on page 56.)
- [16] IRIS Development Team. *IRIS: Integrated Rule Inference System*. IRIS Foundations, 2010. URL <http://www.iris-reasoner.org>. (Cited on page 24.)
- [17] D. Kapur and P. Narendran. Np-completeness of the set unification and matching problems. In *8th International Conference on Automated Deduction*, pages 489–495. Springer, 1986. (Cited on page 7.)
- [18] K. Kersting and L. De Raedt. Towards combining inductive logic programming with bayesian networks. *Inductive Logic Programming*, pages 118–131, 2001. (Cited on page 2.)
- [19] G.J. Klir and B. Yuan. *Fuzzy sets and fuzzy logic*. Prentice Hall New Jersey, 1995. (Cited on page 21.)
- [20] D.E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974. (Cited on page iv.)
- [21] O. Kuželka and F. Železný. A restarted strategy for efficient subsumption testing. *Fundamenta Informaticae*, 89(1):95–109, 2008. (Cited on pages 11 and 25.)
- [22] O. Kuželka, A. Szabóová, and F. Železný. Bounded least general generalization. Unpublished article, 2012. (Cited on pages 56 and 60.)
- [23] Ondřej Kuželka. Efficient construction of relational features for machine learning. Master’s thesis, Czech Technical University in Prague, 2009. (Cited on page 53.)
- [24] M.S. Lam, J. Whaley, V.B. Livshits, M.C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–12. ACM, 2005. (Cited on page 24.)
- [25] N. Landwehr, K. Kersting, and L.D. Raedt. Integrating naive bayes and foil. *The Journal of Machine Learning Research*, 8:481–507, 2007. (Cited on page 56.)

- [26] N. Lavrac and S. Dzeroski. *Inductive logic programming*. E. Horwood, 1994. (Cited on pages 2, 8, and 9.)
- [27] H. Lodhi and S. Muggleton. Is mutagenesis still challenging. *ILP-Late-Breaking Papers*, 35, 2005. (Cited on page 56.)
- [28] J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004. (Cited on page 25.)
- [29] S. Muggleton. Inductive logic programming. *New generation computing*, 8(4): 295–318, 1991. (Cited on page 8.)
- [30] S. Muggleton et al. Stochastic logic programs. *Advances in inductive logic programming*, 32:254–264, 1996. (Cited on page 2.)
- [31] J. Neville and D. Jensen. Collective classification with relational dependency networks. In *Proceedings of the Second International Workshop on Multi-Relational Data Mining*, pages 77–91. Citeseer, 2003. (Cited on page 2.)
- [32] L. Ngo and P. Haddawy. Answering queries from context-sensitive probabilistic knowledge bases. *Theoretical Computer Science*, 171(1):147–177, 1997. (Cited on page 2.)
- [33] A. Paes, F. Železný, G. Zaverucha, D. Page, and A. Srinivasan. Ilp through propositionalization and stochastic k-term dnf learning. *Inductive Logic Programming*, pages 379–393, 2007. (Cited on page 2.)
- [34] G.D. Plotkin. A further note on inductive generalization. *Machine intelligence*, 6(101-124), 1971. (Cited on page 7.)
- [35] D. Poole. Logic, probability and computation: Foundations and issues of statistical relational ai. *Logic Programming and Nonmonotonic Reasoning*, pages 1–9, 2011. (Cited on page 1.)
- [36] D.L. Poole and A.K. Mackworth. *Artificial Intelligence: foundations of computational agents*. Cambridge University Press, 2010. (Cited on page 1.)
- [37] A. Popescul and L.H. Ungar. Structural logistic regression for link analysis. *Departmental Papers (CIS)*, page 133, 2003. (Cited on page 2.)
- [38] pyDatalog Development Team. *pyDatalog: Datalog Programming in Python*, 2012. URL <https://sites.google.com/site/pydatalog/home>. (Cited on page 24.)
- [39] J. Quinlan and R. Cameron-Jones. Foil: A midterm report. In *Machine Learning: ECML-93*, pages 1–20. Springer, 1993. (Cited on page 56.)

- [40] M. Riedmiller. Advanced supervised learning in multi-layer perceptrons—from backpropagation to adaptive learning algorithms. *Computer Standards & Interfaces*, 16(3):265–278, 1994. (Cited on page 20.)
- [41] M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993. (Cited on page 20.)
- [42] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 550–556, 1990. (Cited on page 12.)
- [43] S.J. Russell, P. Norvig, J.F. Canny, J.M. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, NJ, 1995. (Cited on page 21.)
- [44] J. Santos and S. Muggleton. Subsumer: A Prolog theta-subsumption engine. In *Technical communications of the 26th Int. Conference on Logic Programming, Leibniz International Proc. in Informatics, Edinburgh, Scotland, 2010*. (Cited on pages 8, 25, and 26.)
- [45] M. Schmidt-Schauss. Implication of clauses is undecidable. *Theoretical Computer Science*, 59(3):287–296, 1988. (Cited on page 7.)
- [46] W.J. Talbott. Two principles of bayesian epistemology. *Philosophical Studies*, 62(2):135–150, 1991. (Cited on page 1.)
- [47] B. Taskar, P. Abbeel, and D. Koller. Discriminative probabilistic models for relational data. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 485–492. Morgan Kaufmann Publishers Inc., 2002. (Cited on page 2.)
- [48] M.P. Wellman, J.S. Breese, and R.P. Goldman. From knowledge bases to decision models. *The Knowledge Engineering Review*, 7(01):35–53, 1992. (Cited on page 2.)
- [49] J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices*, 39(6):131–144, 2004. (Cited on page 24.)
- [50] Wikipedia. Datalog — wikipedia, the free encyclopedia, 2012. URL <http://en.wikipedia.org/w/index.php?title=Datalog&oldid=521440808>. [Online; accessed 5-November-2012]. (Cited on page 23.)
- [51] LA Zadeh. Fuzzy sets, information and control, 8 (3): 338-353, 1965. (Cited on page 21.)

A

CONTENT OF CD

/	
├── doc/ THESIS TEXT
│ ├── src/ SOURCE CODES
│ └── thesis.pdf PDF FILE
├── in/ EXPERIMENTS INPUTS
│ ├── muta/ MUTAGENESIS
│ ├── ptcmr/ PTC-MR
│ └── README	
├── out/ EXPERIMENTS OUTPUTS
│ ├── muta/ MUTAGENESIS
│ ├── ptcmr/ PTC-MR
│ └── README	
├── prog/ THESIS PROGRAM
│ ├── src/ SOURCE CODES
│ ├── discoverer.jar JAR FILE
│ └── README	