

CZECH TECHNICAL UNIVERSITY  
FACULTY OF ELECTRICAL ENGINEERING  
DEPARTMENT OF CYBERNETICS



**MASTER'S THESIS**

**Efficient Construction of Relational  
Features for Machine Learning**


Prague, 2009

Author: Ondřej Kuželka

## Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu a SW) uvedené v příloženém seznamu.

V Praze dne 21.5.2009



\_\_\_\_\_  
podpis

## Acknowledgement

I would like to thank my supervisor, Ing. Filip Železný Ph.D., for his invaluable guidance, interesting discussions and infinite patience when explaining theoretical basics of ILP to me *again and again until I converged to at least some sort of understanding*. I would also like to thank him for the opportunity to work on interesting and actual problems. Of course, special thanks go also to my parents for their support.

## Abstrakt

Tato práce se zabývá efektivní konstrukcí relačních rysů v kontextu induktivního logického programování. Abychom zrychlili proces konstrukce relačních rysů, navrhujeme a implementujeme dva algoritmy pro problém  $\theta$ -subsumpce a dva algoritmy pro propozicionalizaci pomocí hierarchických rysů. Algoritmy pro  $\theta$ -subsumpci nazvané RE-SUMER a RECOVER testujeme na reálných datech a ukazujeme, že v mnoha, nicméně ne všech, případech překonávají algoritmus Django považovaný za jeden z nejrychlejších algoritmů pro  $\theta$ -subsumpci. Algoritmy RELF a HiFi určené pro propozicionalizaci se opírají o teoretickou analýzu vlastností hierarchických rysů, která ukazuje, že *neredundance* a *relevance* jsou pro hierarchické rysy monotónní v určitém smyslu, což umožňuje těmto algoritmům výrazně zmenšit počet rysů, které musí být vygenerovány. V experimentech se třemi reálnými problémy je demonstrováno, že oba tyto algoritmy jsou schopny konstruovat rysy s délkou nedosažitelnou existujícími systémy RSD a Progol. Dále je ukázáno, že prediktivní přesnost pro tyto tři problémy je často blízká nejlepším výsledkům uváděným v literatuře. Dále je ještě studována výpočetní složitost problému generování rysů splňujících určité syntaktické podmínky.

## Abstract

This thesis aims at efficient construction of relational features in the context of inductive logic programming. In order to speed up construction of relational features, we devise and implement two algorithms for  $\theta$ -subsumption and two algorithms for propositionalization for so called hierarchical features. The algorithms for  $\theta$ -subsumption called RESUMER and RECOVER are tested on real-life data. They are shown to be faster than state-of-the-art  $\theta$ -subsumption algorithm Django in many, but not all cases. The propositionalization algorithms called RELF and HiFi are based on a theoretical analysis, which shows that *relevancy* and *irreducibility* properties of hierarchical features are monotone in a certain sense, which enables the algorithms to considerably reduce numbers of features, which need to be constructed. On three real-life datasets, RELF and HiFi are shown to construct features of lengths achievable neither by state-of-the-art propositionalization system RSD, nor by a general ILP system Progol. Predictive accuracies obtained on these datasets are close to best results reported in literature. Finally, complexity of feature construction is analyzed.

## ZADÁNÍ DIPLOMOVÉ PRÁCE

**Student:** Bc. Ondřej Kuželka

**Studijní program:** Elektrotechnika a informatika (magisterský), strukturovaný

**Obor:** Kybernetika a měření, blok KM2 – Umělá inteligence

**Název tématu:** Efektivní konstrukce relačních rysů pro strojové učení

### Pokyny pro vypracování:

Navrhněte způsoby, jak řádově zvýšit efektivitu současných algoritmů pro konstrukci relačních rysů. Zaměřte se například na:

1. Znáhodněné restartované strategie pro dokazování v predikátové logice.
2. Možnost pravděpodobnostních odhadů pokrytí relačního rysu.
3. Vhodná zúžení gramatiky rysů zaručující konstrukci v polynomiálním čase.

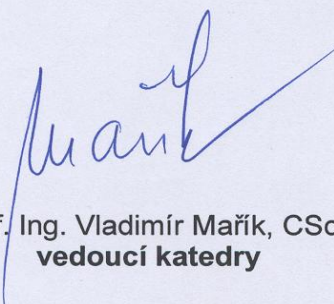
Vzhledem k výzkumné povaze této práce není nutné pokrýt všechny shora navržené směry a naopak je možné navrhnout další nové nápady.

### Seznam odborné literatury:

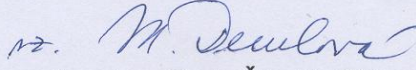
- [1] Džeroski, S.; Lavrač, N. (eds): Relational Data Mining. 2001, XIX, 398 p. 79 illus., Hardcover, ISBN: 978-3-540-42289-1
- [2] Železný, F.; Lavrač, N.: Propositionalization-Based Relational Subgroup Discovery with RSD. Machine Learning 2006 62(1-2):33-63

**Vedoucí diplomové práce:** Ing. Filip Železný, Ph.D.

**Platnost zadání:** do konce zimního semestru 2009/2010

  
prof. Ing. Vladimír Mařík, CSc.  
vedoucí katedry



  
doc. Ing. Boris Šimák, CSc.  
děkan

V Praze dne 3. 9. 2008

## DIPLOMA THESIS ASSIGNMENT

**Student:** Bc. Ondřej Kuželka  
**Study programme:** Electrical Engineering and Information Technology  
**Specialisation:** Cybernetics and Measurement – Artificial Intelligence  
**Title of Diploma Thesis:** Efficient Construction of Relational Features for Machine Learning

### Guidelines:

Develop methods to boost the efficiency of current relational feature construction algorithms by orders of magnitude. Explore namely the following exemplary strategies:

1. Randomized restarted strategies for proving in predicate logic.
2. Probabilistic estimation of relational feature coverage.
3. Suitably constraining the feature grammar to guarantee polynomial-time construction.

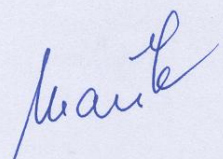
Given the research character of this work, it is not necessary to cover all the above topics; conversely, new ideas may be proposed.

### Bibliography/Sources:

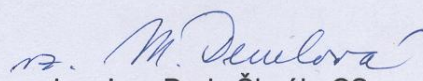
- [1] Džeroski, S.; Lavrač, N. (eds): Relational Data Mining. 2001, XIX, 398 p. 79 illus., Hardcover ISBN: 978-3-540-42289-1
- [2] Zelezny F., Lavrac N.: Propositionalization-Based Relational Subgroup Discovery with RSD. Machine Learning 2006 62(1-2):33-63

**Diploma Thesis Supervisor:** Ing. Filip Železný, Ph.D.

**Valid until:** the end of the winter semester of academic year 2009/2010

  
prof. Ing. Vladimír Mařík, CSc.  
**Head of Department**



  
doc. Ing. Boris Šimák, CSc.  
**Head**

Prague, September 3, 2008

# Contents

List of Figures	x
List of Tables	xi
<b>1 Introduction</b>	<b>1</b>
<b>2 Preliminaries</b>	<b>3</b>
2.1 A Few Concepts from Complexity Theory . . . . .	3
2.2 Constraint Satisfaction Problems . . . . .	4
2.3 $\theta$ -subsumption . . . . .	7
2.4 Propositionalization . . . . .	9
2.4.1 Feature Construction . . . . .	9
2.4.2 Extension Computation . . . . .	10
2.4.3 Feature Filtering . . . . .	11
2.4.4 State-of-the-Art Propositionalization Systems . . . . .	11
2.4.4.1 RSD . . . . .	11
2.4.4.2 WARMR . . . . .	12
2.4.4.3 Feature Description Logics . . . . .	12
2.4.4.4 Horn-SAT Reduction . . . . .	13
2.5 Runtime Distributions . . . . .	14
2.6 Restart strategies . . . . .	17
<b>3 Two <math>\theta</math>-subsumption Algorithms</b>	<b>21</b>
3.1 Runtime Distributions . . . . .	22
3.1.1 Basic Algorithm . . . . .	22
3.1.2 Subsumption Test Runtime Distributions . . . . .	24
3.2 RESUMER: A Restarted $\theta$ -subsumption Algorithm . . . . .	27
3.2.1 Designing a Restarted Subsumption Test Algorithm . . . . .	27



3.2.2	Experimental Evaluation . . . . .	34
3.2.2.1	Generated Data . . . . .	35
3.2.2.2	Predictive Toxicology Challenge Data . . . . .	37
3.3	RECOVER: A Restarted $\theta$ -subsumption Estimator . . . . .	39
3.3.1	Derivation of Coverage Estimator . . . . .	39
3.3.2	Experiments . . . . .	46
3.3.2.1	Sensitivity Analysis . . . . .	47
3.3.2.2	Experiments with Generated Graph Data . . . . .	47
3.3.2.3	Experiments with Real-World Data . . . . .	49
3.4	Discussion of Experiments with RESUMER and RECOVER . . . . .	52
<b>4</b>	<b>Two Propositionalization Algorithms</b>	<b>54</b>
4.1	Analysis of Hierarchical Features . . . . .	55
4.1.1	Hierarchical Features . . . . .	55
4.1.2	Irreducibility . . . . .	57
4.1.3	Relevancy . . . . .	59
4.2	RELF . . . . .	62
4.2.1	Algorithm . . . . .	62
4.2.2	Experiments . . . . .	65
4.2.2.1	Mutagenesis . . . . .	66
4.2.2.2	CAD Documents . . . . .	67
4.2.2.3	Predictive Toxicology Challenge . . . . .	68
4.3	HiFi . . . . .	68
4.3.1	Propositionalization Setting of HiFi . . . . .	69
4.3.2	The Propositionalization Algorithm . . . . .	69
4.3.3	Experimental Evaluation . . . . .	74
4.3.3.1	Predictive Toxicology Challenge . . . . .	75
4.3.3.2	Mutagenesis . . . . .	76
4.3.3.3	CAD Documents . . . . .	77
4.3.3.4	Evaluation of HiFi for Feature Construction . . . . .	78
4.4	Discussion of Experiments with RELF and HiFi . . . . .	79
<b>5</b>	<b>Complexity of Feature Construction</b>	<b>81</b>
5.1	A Negative Result . . . . .	81
5.2	A Positive Result . . . . .	87

<b>6</b>	<b>Conclusions</b>	<b>89</b>
<b>A</b>	<b>Algorithmic Details</b>	<b>I</b>
A.1	Generators of Random $\theta$ -subsumption problems . . . . .	I
A.2	Canonical ordering $\prec_c$ . . . . .	III
<b>B</b>	<b>List of Software Used</b>	<b>VI</b>
<b>C</b>	<b>The Enclosed CD Contents</b>	<b>VII</b>

# List of Figures

2.1	Erratic convergence of mean of a heavy-tailed distribution . . . . .	15
2.2	Pareto (+) and normal (◦) complementary probability distributions . . .	17
3.1	Complementary runtime distributions for random graph data . . . . .	26
3.2	Effect of restarts for satisfiable (left) and unsatisfiable (right) instances .	28
3.3	Effect of individual restart strategies . . . . .	33
3.4	Comparison of RESUMER2 (◦) and Django (◻) on Erdos-Rényi random graphs. . . . .	36
3.5	Simulation of RECOVER with increasing cutoff sequences . . . . .	44
3.6	Sensitivity of RECOVER to violations of Assumption 3.1 . . . . .	46
3.7	Precision of RECOVER (◦) and RECOVER-E (+) . . . . .	48
4.1	Illustration of reuse of pos features for computation of domains. . . . .	65
4.2	An example graph corresponding to template used in Example 4.5. Edge labels are computed from Eq. 4.1. . . . .	73
4.3	Feature construction times for RSD (solid) and HiFi (dashed) . . . . .	79
5.1	Illustration of proof of Theorem 5.2 . . . . .	84

# List of Tables

3.1	Average runtimes for hypothesis search for the PTC dataset and max. hypothesis size 10. . . . .	38
3.2	Average runtimes for hypothesis search for the PTC dataset and max. hypothesis size 100. . . . .	38
3.3	Avg. estimation runtimes for the configurations from Fig. 3.7. . . . .	49
3.4	Average runtimes of the learner (Algorithm 10, $p = 0.75$ , $Tries = 10$ ) for real-world datasets. . . . .	51
3.5	Quality of learned hypotheses for RECOVER . . . . .	51
3.6	Quality of learned hypotheses for Django . . . . .	52
4.1	Accuracies on Mutagenesis dataset. . . . .	67
4.2	Accuracies on CAD dataset. . . . .	67
4.3	Accuracies on PTC dataset for male rats. . . . .	68
4.4	Propositionalization runtimes and accuracies for the PTC dataset . . . .	75
4.5	Propositionalization results on PTC dataset for male rats. . . . .	76
4.6	Propositionalization runtimes and accuracies for the Mutagenesis dataset	76
4.7	Propositionalization results on Mutagenesis dataset. . . . .	77
4.8	Propositionalization runtimes and accuracies for the CAD dataset . . . .	78
4.9	Propositionalization results on CAD dataset. . . . .	78

# Chapter 1

## Introduction

Inductive logic programming (ILP) [10] is a branch of machine learning which focuses on learning first order theories from data. An ILP system is typically given a set of positive examples  $E^+$ , a set of negative examples  $E^-$ , a background theory  $B$  and a language bias and its task is to find a theory  $H$  such that  $\forall e^+ \in E^+ : (B \wedge H) \models e$  and  $\forall e^- \in E^- : (B \wedge H) \not\models e^-$ , i.e.  $H$  and  $B$  explain (*cover*) all positive examples and do not explain any negative example, and  $(H \wedge B) \not\models \square$ , i.e.  $H$  is consistent with  $B$ . This setting is, however, very hard (undecidable in general because of the  $\models$  relation), therefore several less complex and also less expressive modifications of the outlined setting have been proposed in literature.

One such approach to make ILP more tractable is to restrict hypotheses  $H$  to be function-free non-recursive Horn clauses with size smaller than some  $k$  and examples to be ground Horn clauses. In such case,  $\models$  relation can be replaced by so-called  $\theta$ -subsumption and the corresponding ILP problem becomes decidable, though its complexity remains extremely high:  $\mathbf{NP}^{\mathbf{NP}}$  [15]. The inherent hardness of this restriction of the general ILP problem is not the only reason why it is not practical for real-life data. Clearly, not many real-life datasets are noise-free and a theory, which would correctly separate positive examples from negative examples need not always exist. Even if such a theory exists, it need not perform very well on unseen data because it could be overfitted to training data. While this has been also studied in ILP since its beginning (e.g. in [31]), it has been studied more extensively in literature on attribute value learning. It would be therefore desirable to exploit the results from attribute value learning in ILP.

*Propositionalization* is a framework, which enables us to exploit results from attribute value learning for classification learning in the context of ILP. The basic idea behind *propositionalization* is to generate many first order formulas (called *features*) and to let

these formulas act as attributes for attribute value learners. While the basic concept of propositionalization is quite straightforward, developing an efficient propositionalization system remains a hard task because several subproblems, which are encountered in propositionalization, are generally **NP**-hard. A propositionalization system needs to solve basically the following three problems: feature construction, *extension* computation (i.e. computation of examples, which are *covered* by a feature) and feature filtering. In this thesis, we touch all three tasks, though, we do not address all of them in their full generality.

This thesis is organized as follows. In Chapter 2, we describe the necessary background and we briefly overview some existing propositionalization systems. In Chapter 3, we develop two novel algorithms for  $\theta$ -subsumption. In Chapter 4, we develop two novel propositionalization algorithms for a limited class of features with so called *hierarchical bias*. In Chapter 5, we study complexity of feature construction. Chapter 6 concludes the thesis.

## Bibliographical Notes

Since some parts of this thesis contain results which are based on material already presented in author's bachelor's thesis [23], we feel it is important to clarify originality of the results presented here. Nothing in Chapter 2 is novel as this chapter is intended to make the reader familiar with existing results, which are more or less exploited in the other chapters. Chapter 3 presents two algorithms for  $\theta$ -subsumption problem. Early versions of these algorithms appeared also in authors's bachelor thesis. However, the results in this chapter have been significantly extended. A new restarted strategy was developed and analyzed and new heuristics were developed and implemented. The algorithms were also reimplemented from scratch, which caused considerable speed-up. Finally, experiments with real world data were performed. Chapters 4 and 5 contain novel work, which was not contained in the bachelor's thesis.

Results presented in Chapter 3 were published in [26] and [24]. Results presented in Chapter 4 were published in [27] and [25].

# Chapter 2

## Preliminaries

### 2.1 A Few Concepts from Complexity Theory

In this section, we provide definitions of several concepts from complexity theory. We avoid definitions based on non-deterministic Turing machines, which, despite their elegance, are not necessary for problems discussed in this thesis. This section is mostly based on material from books [42, 2].

**Definition 2.1 (Big-O notation):** We write  $f(n) = O(g(n))$  if  $f(n) < k \cdot g(n)$  for some  $k$ , for all  $n > n_0$ . ▶

The next definition introduces the class of decision problems  $\mathbf{P}$ , which roughly<sup>1</sup> corresponds to a class of efficiently decidable problems.

**Definition 2.2 ( $\mathbf{P}$ ):** Let  $\mathbf{DTIME}(f(n))$  be the set of decision problems computable in time  $O(f(n))$  by a deterministic Turing machine. Class  $\mathbf{P}$  can be defined as  $\mathbf{P} = \cup_{c \geq 1} \mathbf{DTIME}(n^c)$ , where  $n$  refers to length of the input. ▶

In the following definition, the class of decision problems  $\mathbf{NP}$  is introduced.  $\mathbf{NP}$  is the class of decision problems, whose solution is easy to verify, i.e. in polynomial time. It is an open question whether  $\mathbf{P} = \mathbf{NP}$ , but it is believed by most computer scientists that  $\mathbf{P} \neq \mathbf{NP}$ .

**Definition 2.3 ( $\mathbf{NP}$ ):** A decision problem  $P$  is in  $\mathbf{NP}$  if there exists a polynomial  $p : N \rightarrow N$  and a polynomial-time Turing machine  $M$  (called *verifier*): such that for every

---

<sup>1</sup>We say *roughly corresponds*, because an algorithm with complexity  $n^{100}$  would be no more useful than an algorithm with exponential time complexity in practice.

input  $x$ , a string  $c$ ,  $|c| < p(|x|)$  (called *certificate*) **exists** such that  $M$  accepts  $(x, c)$  if and only if the answer for the input  $x$  is *yes*. ►

The class of decision problems **co-NP**, introduced through Definition 2.4, contains decision problems, for which the answer *no* can be easily verified, i.e. in polynomial time. It holds  $\mathbf{NP} \cap \mathbf{co-NP} \neq \emptyset$  and it is commonly believed that  $\mathbf{NP} \neq \mathbf{co-NP}$ .

**Definition 2.4 (co-NP):**<sup>2</sup>. A decision problem  $P$  is in **co-NP** if there exists a polynomial  $p : N \rightarrow N$  and a polynomial-time Turing machine  $M$  (called *verifier*): such that for every input  $x$  and for **every** string  $c$ ,  $|c| < p(|x|)$ ,  $M$  accepts  $(x, c)$  if and only if the answer for the input  $x$  is *yes*. ►

**Definition 2.5 (Polynomial-time reduction):** We say that a decision problem  $A$  is polynomial-time reducible to a decision problem  $B$  denoted by  $A \leq_p B$  if there is a polynomial-time computable function  $f$  such that for every instance  $x$  of problem  $A$ ,  $f(x)$  is an instance of problem  $B$  and  $x$  has a solution if and only if  $f(x)$  has solution. ►

**Definition 2.6 (NP-completeness):** We say that a decision problem  $B$  is **NP**-complete if  $A \leq_p B$  for every  $A \in \mathbf{NP}$  and  $B \in \mathbf{NP}$ . ►

## 2.2 Constraint Satisfaction Problems

Many combinatorial search problems, among them  $\theta$ -subsumption, which is important for ILP, can be solved within the constraint satisfaction framework. The exposition of constraint satisfaction, which is presented in this section, is based on the book [8].

The constraint satisfaction problem (CSP, sometimes also called *constraint network*) is defined as follows [38].

**Definition 2.7 (Constraint satisfaction problem):** Let  $V$  be a set of variables  $\{V_1, V_2, \dots, V_n\}$ ,  $D$  a set of non-empty domains  $\{D_1, D_2, \dots, D_n\}$  for each variable  $V_i \in V$  and let  $C$  be a set of constraints  $\{C_1, C_2, \dots, C_m\}$ . Each constraint  $C_i$  specifies the allowable combinations of values that can be assigned to variables constrained by it. The task is to find a mapping from  $V$  onto  $D$  such that after assigning values given by this mapping to the respective variables no constraint  $C_i \in C$  is violated. ►

---

<sup>2</sup>This is not the most commonly used definition of the class **co-NP**. However, a convenient property of this definition is that it differs from the definition of **NP** only slightly and thus provides more intuition about their relationship. This definition was taken from [2]



One of the main advantages of the CSP framework is that it enables the search algorithm to exploit the structure of constraints by so called constraint propagation methods. We will briefly mention two such methods: forward checking (FC) and arc consistency (AC), which are implemented in the  $\theta$ -subsumption algorithms developed Chapter 3 and in the state-of-the-art  $\theta$ -subsumption algorithm Django.

Forward checking is a simple constraint propagation method, which is used during backtracking search. Whenever a value is assigned to a variable  $V$  during the search, the FC procedure propagates this assignment to all unassigned variables that share at least one constraint with  $V$  and prunes their domains accordingly. The use of FC has several advantages over plain backtracking. One of such advantages is that it makes it possible to detect inconsistencies soon in the search process (when the domain of a variable becomes empty after FC). It also enables it to use variable ordering heuristics based on domain sizes.

Arc consistency is a method much more powerful than FC, though it pertains to higher overhead. The arc consistency algorithm removes all values from their respective domains, which are not 2-consistent according to Definition 2.8.

**Definition 2.8 (Arc consistency):** We say that value  $a \in D_i$  is arc consistent (or 2-consistent), if there is a value  $b_j \in D_j$  for every  $V_j$ , for which there is a constraint  $c(V_i, V_j)$ , and  $(a, b_j)$  satisfies the constraint  $c$ .

We say that a CSP is arc consistent if every variable  $V_i \in V$  has at least one arc consistent value. ▶

The efficiency of arc consistency lies in its ability to reduce the variable domains. Bessiere [3] shows that arc consistency can vastly outperform FC on hard problem instances. A pleasant property of arc consistency is that showing that a CSP is arc consistent is a necessary and sufficient condition to solve tree structured CSPs. Its drawback is, however, its relatively high time complexity. The best currently known arc consistency algorithms have worst case time complexity given by  $O(v^2c)$ , where  $v$  is the variable domain size and  $c$  the number of constraints. The most well-known algorithm is, however, the AC-3 algorithm (Algorithm 2), which has worst case time complexity  $O(v^3 \cdot c)$  but which has also quite favorable practical average performance [8]. AC-3 is used both in Django and in our  $\theta$ -subsumption algorithm.

Next, we define projection of constraints (projection networks), which can be used for constraint propagation for non-binary CSP problems. Even though performing arc-consistency propagation on a projected network is generally weaker than performing

---

**Algorithm 1** AC-3: Given a binary CSP problem, AC-3 filters domains of the CSP-variables

---

```

1: Input: Set of variables  $V$ , Set of variable domains  $D$ , Set of constraints  $C$ ;
2:  $queue \leftarrow \emptyset$ 
3: for every pair  $\{x_i, x_j\}$  which is contained in at least one constraint  $c_k \in C$  do
4:    $queue \leftarrow queue \cup \{(x_i, x_j), (x_j, x_i)\}$ 
5: end for
6: repeat
7:    $(x_i, x_j) \leftarrow Pop(queue)$ 
8:    $Revise(x_i, x_j)$ 
9:   if  $Revise(x_i, x_j)$  caused a change in some domain then
10:     $queue \leftarrow \{(x_k, x_i) : k \neq i, k \neq j\}$ 
11:   end if
12: until  $queue = \emptyset$ 

```

---

**Algorithm 2** Revise: Given a pair of of CSP-variables, filters  $D_1$  such that  $V_1$  is arc-consistent relative to  $V_j$

---

```

1: Input: Variables  $V_1, V_2$ , Domains  $D_1, D_2$ ;
2: for  $\forall a_i \in D_1$  do
3:   if there is no  $a_j \in D_2$  such that  $(a_i, a_j)$  is consistent then
4:     Delete  $a_i$  from  $D_1$ 
5:   end if
6: end for

```

---

general arc-consistency on the respective non-binary CSP problem, it is still quite useful and has smaller overhead.

**Definition 2.9 (Projection Network):** Let  $\rho$  be a constraint over  $V = \{V_1, V_2, \dots, V_n\}$ . The projected constraint network is a network composed from constraints  $c_k$  over all pairs of variables  $(V_i, V_j) \in V^2$  such that  $c_k$  contains pairs of values, which can be substituted to the respective variables  $V_i, V_j$  without making  $\rho$  inconsistent. ▶

The constraint propagation methods as well as other methods used for solving constraint satisfaction problems are treated in more detail in [8].

## 2.3 $\theta$ -subsumption

Inductive logic programming [32] relies heavily on testing whether a hypothesis  $C$  covers an example  $e$ . In the learning from entailment setting [31], an example  $e$  is said to be covered by a clause  $C$  if and only if  $C \models e$ . Because such tests are generally undecidable in the first order logic, Plotkin [34] defined  $\theta$ -subsumption, which can be viewed as an approximation of implication.

**Definition 2.10 ( $\theta$ -subsumption):** Let  $C$ ,  $D$  and  $e$  be clauses. The clause  $C$   $\theta$ -subsumes  $e$ , if and only if there is a substitution  $\theta$  such that  $C\theta \subseteq e$ . If  $D \preceq_{\theta} C$  and  $C \preceq_{\theta} D$ , we call  $C$  and  $D$   $\theta$ -equivalent (written  $C \approx_{\theta} D$ ).  $\blacktriangleright$

$\theta$ -subsumption is incomplete, which means that there are clauses  $C$  and  $E$ , such that  $C$  does not  $\theta$ -subsume  $e$ , but  $C \models E$  [40]. Nevertheless it always holds that if  $C$   $\theta$ -subsumes  $E$ , then  $C \models E$ , which immediately follows from the fact that both  $C$  and  $e$  are clauses, i.e. disjunctions of literals, and therefore whenever  $C$  is true in an interpretation,  $e$  must be true in this interpretation as well, because  $e$  contains  $C\theta$ .

In this thesis, we constrain ourselves to function-free non-recursive formulas, i.e. formulas, which contain no function symbols other than constants.

$\theta$ -subsumption can be solved in the constraint satisfaction framework introduced in the previous section. A constraint representation of the  $\theta$ -subsumption test is very straightforward, if we allow  $n$ -ary constraints as shown later in this section. The most advanced CSP algorithms were, however, developed for binary constraints. Luckily, as is shown in [38], any CSP with  $n$ -ary constraints can be rewritten to a representation with just unary and binary constraints. Two binarizations of  $\theta$ -subsumption are described in [30].

The  $\theta$ -subsumption algorithms presented in this thesis use neither binarization nor more general methods for  $n$ -ary constraints. Instead, they use forward checking, which applies to binary and non-binary cases in roughly the same manner, and arc-consistency on projections (Def 2.9) of the generally non-binary constraint networks.

Both  $\theta$ -subsumption and CSP problems with finite domain are instances of **NP**-complete problems. Therefore it is not surprising that they can be converted between each other in polynomial time. However,  $\theta$ -subsumption and finite-domain CSP are more similar to each other than e.g. the Hamiltonian path problem and finite-domain CSP. In certain sense, they are the same problems with two different names, which is manifested by the ease of conversion between them shown in the next two examples.

**Example 2.1 (Converting  $\theta$ -subsumption to CSP):** Let  $C = hasCar(C) \vee hasLoad(C, L) \vee box(L)$  be a hypothesis and let  $e = hasCar(c) \vee hasLoad(c, l_1) \vee hasLoad(c, l_2) \vee box(l_2)$  be an example. We now show how we can convert the problem of deciding  $C \preceq_{\theta} e$  to a CSP problem (with  $n$ -ary constraints in general).

Let  $V = \{C, L\}$  be a set of CSP-variables and let  $D = \{D_C, D_L\}$  be a set of domains of variables from  $V$  such that  $D_C = \{c\}$  and  $D_L = \{l_1, l_2\}$ . Further, let  $C = \{C_{hasCar(C)}, C_{hasLoad(C,L)}, C_{box(L)}\}$  be a set of constraints such that  $C_{hasCar(C)} \approx \{(c)\}$ ,  $C_{hasLoad(C,L)} \approx \{(c, l_1), (c, l_2)\}$  and  $C_{box(L)} \approx \{(l_1), (l_2)\}$ . Then the constraint satisfaction problem given by  $V$ ,  $D$  and  $C$  represents the  $\theta$ -subsumption problem  $C \preceq_{\theta} e$  as it has solution if and only if  $C \preceq_{\theta} e$  holds.  $\triangle$

The above example shows that converting  $\theta$ -subsumption to CSP is really straightforward. This is not very surprising. However, the conversion from CSP to  $\theta$ -subsumption is also similarly easy. In such a conversion, each CSP-variable becomes a first-order-logic (FOL) variable and each constraint becomes a FOL-literal. The allowed  $n$ -tuples of values determined by the constraints are then represented by literals in an example  $e$  - each  $n$ -tuple gives rise to one literal in  $e$ .

**Example 2.2 (Converting graph coloring to  $\theta$ -subsumption):** In this example, we will show how to convert an instance of graph coloring, which is easily solved using CSP, to an instance of  $\theta$ -subsumption. The conversion of graph coloring will also enlighten the conversion from CSP to  $\theta$ -subsumption. Let  $G$  be an undirected graph consisting of three vertices forming a cycle of length 3. We ask whether this graph can be colored with three colors. The next  $\theta$ -subsumption problem corresponds to this problem.

$$\begin{aligned} C &= diff(A, B) \vee diff(B, C) \vee diff(C, A) \\ e &= diff(red, green) \vee diff(green, red) \vee diff(red, blue) \vee diff(blue, red) \vee \\ &\quad \vee diff(blue, green) \vee diff(green, blue) \\ C &\preceq_{\theta} e \end{aligned}$$

It holds  $C \preceq_{\theta} e$  if the coloring exists and the valid colorings are given by the respective substitutions  $\theta$  such that  $C\theta \subseteq e$ .  $\triangle$

The example above showed how natural a conversion from a CSP instance can be. In fact, it showed a bit more. It showed that  $\theta$ -subsumption can be **NP**-complete even if we fix the clause  $e$ . Indeed, any instance of graph 3-coloring, which is an **NP**-complete problem, may be converted into a  $\theta$ -subsumption problem with the fixed  $e$  from the example above.

## 2.4 Propositionalization

In Chapter 1, we have described informally what is propositionalization and why it is of interest. In this section, we discuss propositionalization in more detail. First, we describe the basic parts of propositionalization: feature construction, extension computation and feature filtering. Then, we briefly describe basic principles of several state-of-the-art propositionalization systems.

### 2.4.1 Feature Construction

Feature construction is one of two core problems that every propositionalization system must face (the other one is extension computation). Input to a feature construction procedure is a set of syntactical constraints, which must be satisfied by the generated *features*. A form of syntactical constraints, which is very popular in ILP, are mode declarations, well-known from ILP system Progol [31]. For our purposes it will suffice to note that mode declarations define, which predicate symbols can be used in the generated features, and also the way in which the literals can be connected to each other (through share of variables) in these features. In [46], it has been shown that, for the purposes of feature construction, mode declarations can be replaced by so-called templates.

**Definition 2.11 (Template):** Given a set  $A$  of atoms, we denote  $Args(A) = \{(a, n) | a \in A, 1 \leq n \leq \text{arity}(a)\}$ , i.e.  $Args(A)$  is the set of all argument places in  $A$ . A pre-template is a pair  $(\gamma, \mu)$  where  $\gamma$  is a finite set of ground atoms and  $\mu \subseteq Args(\gamma)$ . Elements of  $\mu$  ( $Args(\gamma) \setminus \mu$ ) are called *inputs* (*outputs*) in  $\gamma$ . A pre-template  $(\gamma, \mu)$  is a *template* if there is a partial irreflexive order  $\prec$  on constants in  $\gamma$  such that  $c \prec c'$  whenever  $c$  appears as an input and  $c'$  as an output in some  $l \in \gamma$ . ▶

**Definition 2.12 (Feature):** Given a template  $\tau = (\gamma, \mu)$ , a  $\tau$ -pre-feature  $F$  is a finite conjunction of literals containing no constants or functions, such that  $lits(F\theta) \subseteq \gamma$  for some substitution  $\theta$ . The occurrence of variable in the  $i$ -th argument of literal  $l$  in  $F$  is an *input* (*output*) occurrence in  $F$  if the  $i$ -th argument of  $l\theta$  is (is not) in  $\mu$ . A variable is *neutral* in  $F$  if it has [i] at least one input occurrence in  $F$ , and [ii] exactly one output occurrence in  $F$ . A  $\tau$ -pre-feature  $F$  is a  $\tau$ -feature if all variables in  $F$  are neutral. ▶

The definition of templates and the definition of features provided above (Def. 2.11 and Def. 2.12) would not be valid if we did not require existence of a partial irreflexive

order on constants in  $\gamma$ . Indeed, if we wanted to express the following *incorrect template*  $p(+a, -a, -a), p(-a, +a, -a)$  in the theoretical framework given by the definitions,  $\tau$  would look as follows:  $\tau = (\gamma = \{p(a, a, a)\}, \mu = \{(p(a, a, a), 1), (p(a, a, a), 2)\})$ . However, we could not decide which arguments of e.g. feature  $F = p(A, A, A)$  are inputs and which are outputs given a substitution  $\theta$  such that  $F\theta \subseteq \gamma$ . The problem is that the two declared predicates  $p(+a, -a, -a), p(-a, +a, -a)$  give rise to only one literal in  $\gamma$ . On the other hand, if a template satisfies the condition on the existence of a partial irreflexive order on constants, then this cannot happen because any two literals in a template must have differently typed arguments as, otherwise, there would be a cycle in the partial irreflexive order, which contradicts the definition of *partial irreflexive order*.

For a correct  $\tau$ -feature  $F$ , the assignment of inputs and outputs also need not be given uniquely. However, in this case, it is not a problem, because, unlike in the previous case, the assignment of inputs and outputs becomes unique given a substitution  $\theta$  such that  $F\theta \subseteq \gamma$ .

Existing propositionalization systems tackle the problem of feature construction for features (as defined in Def. 2.12) by depth-first search, which brings an exponential factor into propositionalization. As we show in Chapter 5, this super-polynomial runtime for feature construction is unavoidable (unless  $\mathbf{P} = \mathbf{NP}$ ).

## 2.4.2 Extension Computation

Given a set of examples  $E$ , extension of a feature  $F$  ( $ext_E(F)$ ) is the set of examples covered by  $F$ . Particular settings in ILP have different meaning of the notion *covers* [35].

**Definition 2.13 (Learning from Entailment):** If  $H$  (hypothesis) is a clausal theory and  $e$  a clause, then  $H$  covers  $e$  under entailment if and only if  $H \models e$ . ▶

**Definition 2.14 (Learning from Interpretations):** If  $H$  (hypothesis) is a clausal theory and  $e$  is a Herbrand interpretation, then  $H$  covers  $e$  under interpretations if and only if  $e$  is a model for  $H$ . ▶

Since the entailment relation is generally undecidable, it is typically approximated in practice, e.g. by  $\theta$ -subsumption or by limiting depth of resolution trees. If we restrict ourselves to learning a single predicate and if we restrict the hypotheses to be function-free and non-recursive, then we may use a slightly simpler version of the learning from interpretations setting, which is used in Chapter 4.

Feature construction and extension computation do not need to be tackled separately. In fact, a combination of feature construction and extension computation can make the propositionalization process much more efficient as we also show in Chapter 4. Nevertheless, some state-of-the-art systems (e.g. RSD [44] or SINUS [21]) have separate procedures for feature construction and extension computation.

### 2.4.3 Feature Filtering

Once a set of features together with features' extensions is constructed, it is usually necessary to reduce this set because the number of generated features is typically overwhelming. Propositionalization systems (e.g. RSD [44]) cope with this problem by discarding all but one feature from each equivalence class corresponding to features with equal extensions.

Feature filtering can be also seen as feature selection and any of the wide range of feature selection methods can be used. Also, feature filtering need not be done only after both feature construction and extension computation take place. Instead, all these three steps can be combined into one, typically more efficient, step. For example, frequent-pattern discovery systems such as WARMR [9] can be seen as propositionalization systems, which combine feature construction, extension computation and filtering of infrequent features. In Chapter 4, we design propositionalization algorithms for a limited class of features, which also combine these three steps but which can use more sophisticated filtering than is mere filtering of infrequent features.

### 2.4.4 State-of-the-Art Propositionalization Systems

In this section, we describe several state-of-the-art propositionalization systems. One of the systems, RSD, will be used in Chapter 4 for evaluation of our novel propositionalization algorithms.

#### 2.4.4.1 RSD

RSD [44] is a state-of-the-art system for propositionalization and subgroup discovery. RSD is based on syntactically restricting the set of possible features. Its propositionalization procedure has two stages: feature construction and extension computation. In the first stage, it constructs a set of all correct features w.r.t. some given template  $\tau$  and some  $n \in N$ . In the second stage, it computes extensions of the features generated in the

first stage and processes constants.

Unlike templates considered in this thesis, RSD’s templates are not obliged to have the partial order on types. Therefore RSD would allow e.g. template  $vertex(+a), vertex(-a), edge(+a, -a)$ , which would not be allowable in the *feature-template* framework used in this thesis. RSD uses depth-first search to construct features correct w.r.t. some  $\tau$  and  $n$  and resolution in Prolog to compute their extensions. A detailed evaluation of RSD and two other propositionalization systems (SINUS [21] and RELAGGS [22]) can be found in [21].

#### 2.4.4.2 WARMR

WARMR [9] is a prominent frequent query discovery system, whose output can also be used for attribute value learning. WARMR constructs queries in a top-down manner and exploits anti-monotonicity of query frequency. Several performance optimizations have been proposed and implemented to WARMR. Therefore WARMR might seem as a good candidate for comparison with our novel algorithms HiFi and RELF, however, the way WARMR prunes discovered queries introduces a hardly interpretable bias into propositionalization. The problem is that WARMR prunes every query, which is  $\theta$ -equivalent to some already discovered query [33]. For example, if WARMR found feature  $F_1 = \leftarrow car(C) \wedge hasLoad(C, L1) \wedge hasLoad(C, L2)$  and if it had already found feature  $F_2 = \leftarrow car(C) \wedge hasLoad(C, L)$ , it would prune  $F_1$ , because  $F_1 \approx_\theta F_2$ . It follows that e.g. query  $\leftarrow car(C) \wedge hasLoad(C, L1) \wedge hasLoad(C, L2) \wedge triangle(L1) \wedge box(L2)$  would never be found<sup>3</sup>. This would make comparison of WARMR and our algorithms hard to interpret.

#### 2.4.4.3 Feature Description Logics

In [7], a propositionalization method based on feature description logics was introduced. It used so called concept graphs for efficient (polynomial-time) subsumption computation. However, this approach differs in several aspects from ours. First, features correspond to distinct substitutions of variables in feature templates in this approach, e.g. using similar example as given in paper [7], template (*AND name (father name)*) might give rise to the following two features (*AND name(Homer) (father name(Lisa))*) and (*AND name(Homer) (father name(Bart))*) in some appropriate interpretation. While such

---

<sup>3</sup>This particular example could be remedied by representing *load shapes* by constants, however, in more complex settings no such simple remedies exist.



features are also expressible in our framework, templates, used in our algorithms and e.g. in RSD or WARMR, are much more flexible.

#### 2.4.4.4 Horn-SAT Reduction

In [52], a reduction of feature construction problem to enumeration of solutions of Horn-SAT problem was shown. Horn-SAT is a subclass of NP-complete SAT [42] problem, which is solvable in polynomial time. It is possible to find a polynomial-time conversion e.g. for tree-like features. The reduction is a two-phase process. First, a bottom clause  $\perp$  complying to a given template  $\tau$  and respecting maximum allowed size of features is constructed. In the second phase, a set of Horn clauses  $H$  is created, which encodes the set of possible  $\tau$ -features. Each literal  $l \in \perp$  gives rise to a boolean variable  $P_l$ . The set of Horn clauses then encodes all constraints given by  $\tau$ . Each solution of  $H$  represents one feature, where a literal is added if and only if the respective boolean variable has value *false*. Rather than discussing this approach in its full generality, we show here an example from [52].

**Example 2.3:** Let

$$\tau = \text{hasCar}(-c), \text{hasRoof}(+c), \text{hasLoad}(+c, -l), \text{box}(+l), \text{triangle}(+l)$$

be a template and let  $n = 3$ . The corresponding bottom clause is

$$\perp = \leftarrow \text{hasCar}(C) \wedge \text{hasRoof}(C) \wedge \text{hasLoad}(C, L) \wedge \text{box}(L) \wedge \text{triangle}(L).$$

We assign to each literal from  $\perp$  a boolean variable, consecutively:  $P_1 \approx \text{hasCar}(C)$ ,  $P_2 \approx \text{hasRoof}(C)$ ,  $P_3 \approx \text{hasLoad}(C, L)$ ,  $P_4 \approx \text{box}(L)$ ,  $P_5 \approx \text{triangle}(L)$ .

The respective set of Horn clauses will contain the following clauses:  $\neg P_2 \vee \neg P_3 \vee P_1$  (*if car(C) is in a feature F, then either hasRoof(C) or hasLoad(C, L) must be in F*, which corresponds to the condition that there must be an input occurrence of each variable in any correct  $\tau$  feature),  $\neg P_4 \vee \neg P_5 \vee P_3$  (*if hasLoad(C, L) is in a feature F, then either triangle(L) or box(L) must be in F*),  $\neg P_1 \vee P_2$ ,  $\neg P_1 \vee P_3$ ,  $\neg P_3 \vee P_4$ ,  $\neg P_3 \vee P_5$  (*if a child of some literal is in a feature F, then its parent must be in F*),  $\neg P_1 \vee \neg P_2 \vee \neg P_3 \vee \neg P_4 \vee \neg P_5$  (*any feature must have at least one literal*).  $\triangle$

An obvious drawback of this approach is that without some symmetry breaking, which is not considered in [52], the same feature can be generated many times.

## 2.5 Runtime Distributions

The runtime distribution function which expresses the probability that a given algorithm stops before  $t$  time units (or before  $t$  searched nodes depending on the context), is defined as

$$F(t) = P[T \leq t], \quad (2.1)$$

The complementary distribution function (also known as survival function), which is another frequently used characteristics of the behavior of search algorithms, is defined as

$$S(t) = 1 - P[T \leq t]. \quad (2.2)$$

We note that there are basically two distinct things, for which we will use terms runtime distribution and complementary runtime distribution in this thesis. First, assuming that we have a generator of random instances (e.g. pairs of clauses for  $\theta$ -subsumption check), the runtime distribution and survival function will refer to runtimes of a given search algorithm on the instances generated by this generator. Second, assuming that we have a single fixed problem instance and a search algorithm, which possesses some kind of randomness, the probability distribution of the time needed to solve the problem will refer to runtime distribution of the randomized algorithm on the fixed instance.

The quantity characterizing the runtime distribution that is most important for evaluation of the performance of a search algorithm is naturally its mean, as it expresses the average amount of time needed to solve a sufficiently high number of problem instances. The mean sometimes behaves in a rather strange way. As shown in [14], it sometimes does not stabilize in a reasonable number of runs (Fig. 2.1). To explain such an erratic behavior of mean and to gather better insight into behavior of search algorithms, one needs to focus on the whole runtime distribution.

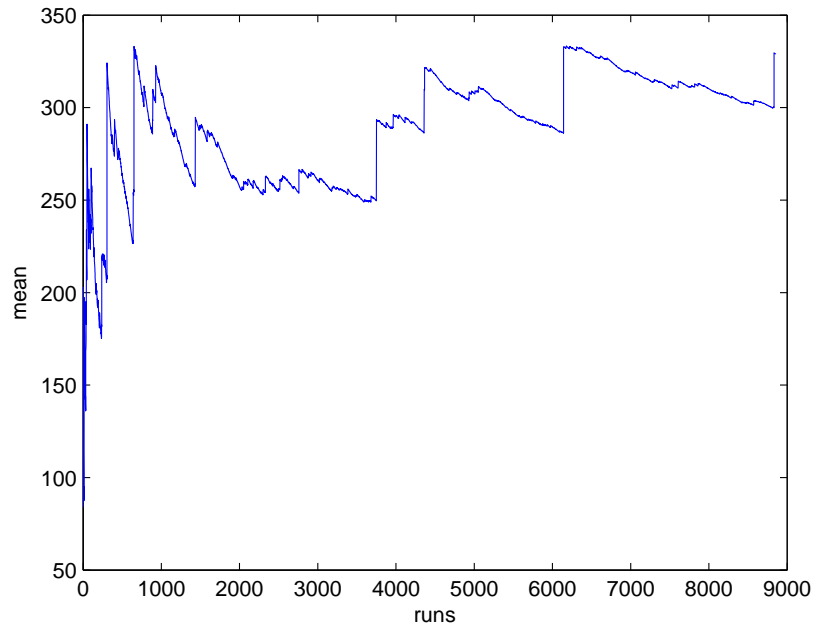


Figure 2.1: Erratic convergence of mean of a heavy-tailed distribution

Runtime distributions of search algorithms have wide range of possible forms [13], but despite such a variability there are basically two main classes of distributions into which they can be divided according to the decay of their tails: exponential and heavy tailed distributions.

The former ones are quite common in physics and engineering. The tails of their survival functions are bounded by exponential functions. A typical example is the well-known normal distribution, whose probability density function is given by

$$f(t) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(t-\mu)^2}{2\sigma^2}}, \quad (2.3)$$

and whose tail for  $\mu = 0$  is approximately [53]

$$1 - F(t) \sim \frac{1}{t\sqrt{2\pi}} e^{-\frac{t^2}{2\sigma^2}}. \quad (2.4)$$

The distributions with exponentially decaying tails cannot explain the encountered erratic behavior of the expected runtime in Fig. 2.1. Such a phenomenon, however, naturally appears for the latter class of runtime distributions, the heavy tailed distributions. Using the definition from [49], a distribution function  $F(t)$  is said to be heavy tailed if

$$P[T > t] \sim C \cdot t^{-\alpha}, \text{ where } t \rightarrow \infty, 0 < \alpha < 2, C > 0 \quad (2.5)$$

The heavy tailed distributions have some intriguing properties. The heavy tailed distributions with  $\alpha < 1$  have infinite mean and all higher moments. For  $\alpha \in (1, 2)$ , the distributions have finite mean but variance and all higher moments are again infinite. This is the reason for the encountered erratic behavior of the expected runtime. However, in practice, the runtime distributions are bounded, because the search spaces have finite size, and therefore also finite mean and all higher moments. On the other hand, the size of the search space might be often so high that it is no difference from the point of view of someone who waits for the solution whether the runtime really has infinite mean or whether it has mean as high as the age of universe.

The simplest heavy tailed distribution is the Pareto distribution defined as

$$F(t) = 1 - t^{-\alpha}, \quad \alpha > 0, \quad t \geq 1 \quad (2.6)$$

Another example is the Weibull distribution defined as

$$f(x, k, \lambda) = \frac{k}{\lambda} \left(\frac{x}{\lambda}\right)^{k-1} e^{-(x/\lambda)^k}, \quad (2.7)$$

which is heavy-tailed for  $c < 1$  and  $a > 0$ .

There is a simple visual method for quickly estimating whether a sample of values belongs to a heavy tailed distribution. It is based on the fact that the tail of a heavy tailed distribution looks approximately linearly when plotted in log-log scale, whereas distribution with the exponential tail exhibits a sharp decay and looks like an exponential curve. The difference between log-log plots of a heavy tailed distribution and a distribution with exponentially decaying tail is shown in Fig. 2.2

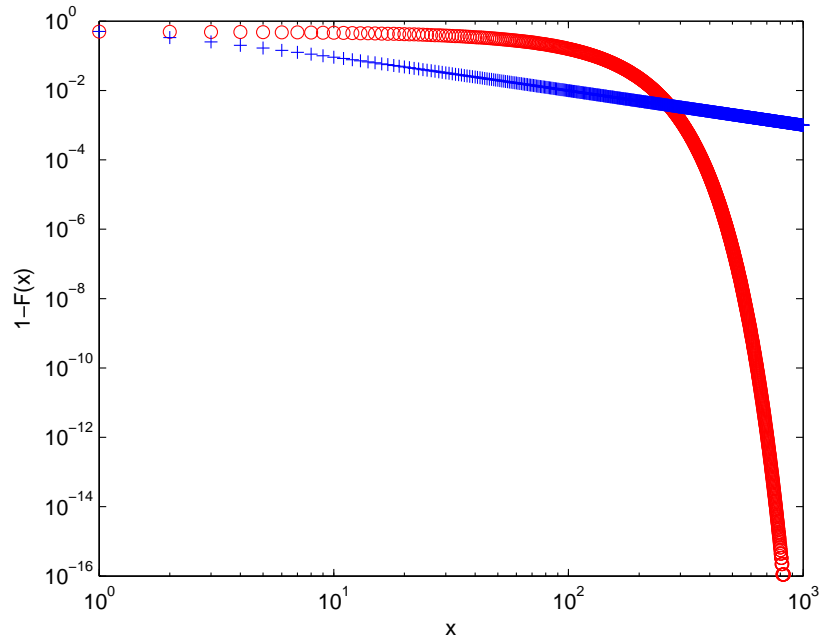


Figure 2.2: Pareto (+) and normal ( $\circ$ ) complementary probability distributions

Heavy tailed distributions have been identified in various domains, for example in economy, statistical physics, in characterization of hyperlinks on the web, in sociology, biology etc.

## 2.6 Restart strategies

When the runtime distribution of a search algorithm is heavy tailed, the mean runtime of the search process may become extremely high. Fortunately, there is a way to overcome this deficiency by so called rapid randomized restart strategies (RRRS) first used by Harvey [17], which consists of randomization of the value and variable ordering heuristics and repeated restarting of the search after a given cutoff time. The cutoff time can be either constant (fixed cutoff strategy) or might be given by a function  $f : N \rightarrow R$ .

The feasibility of RRRS is due to their ability to eliminate heavy tails from arbitrary runtime distributions [29], which has been empirically proven on many real-world and randomly generated problems [49]. The theoretical justification of the fixed cutoff strategy is given by Theorem 2.1 [14].

---

**Algorithm 3** A restarted search algorithm
 

---

**Input:** A problem instance *instance*.

$n \leftarrow 1$

**repeat**

    Answer  $\leftarrow$  Run *SearchAlgorithm*(*instance*) with number of searched nodes limited to  $R(n)$

$n \leftarrow n + 1$

**until** Solution is found

**return** Solution

---

**Theorem 2.1:** *For an arbitrary distribution  $F(t)$ , the strategy with a fixed cutoff  $t_c$  guarantees that the survival function  $S(t)$  of the restarted search algorithm with statistically independent runs decays exponentially.*

**Proof:** Lets assume that the problem instance, we are going to solve, has at least one solution and that the probability of finding one of them in time  $t \leq t_c$  is  $p_{succ} = F(t_c)$ . Then the probability that the restarted search procedure finds a solution in its  $r$ -th run follows the geometric distribution and is given by

$$P_{restarts}(Runs = r) = p_{succ} \cdot (1 - p_{succ})^{r-1} \quad (2.8)$$

The survival function is then

$$S_{restarts}(r) = P_{restarts}(Runs > r) = (1 - p_{succ})^r, \quad (2.9)$$

which is already exponential with respect to number of restarts  $r$ . To show that the survival function  $S(t)$  also decays exponentially it suffices to show that

$$\begin{aligned} S(t) &= P(Time > t) \leq P(Time > \lfloor t/t_c \rfloor \cdot t_c) = \\ &= P_{restarts}(Runs > \lfloor t/t_c \rfloor) = (1 - p_{succ})^{\lfloor t/t_c \rfloor} \leq (1 - p_{succ})^{t/t_c - 1}, \end{aligned} \quad (2.10)$$

which finishes the proof, as it shows that the survival function  $S(t)$  is bounded by exponential 2.10.  $\square$

Luby et al [29] proved that there is a cutoff value  $t_c^*$  such that the restart strategy with this cutoff is optimal over all static restart strategies. Even though the derivation of the optimal value for the cutoff is quite straightforward, we will not present it here and instead show only the main result.

**Theorem 2.2:** *The cutoff  $t_c^*$  is optimal, i.e. the mean runtime of the restarted search  $E[T_{restarted}]$  is minimal, under assumption that*

$$E[T_{restarted}] = \frac{1 - F(t_c^*)}{f(t_c^*)}, \quad (2.11)$$

where  $F(t)$  is the original continuous runtime distribution of the non-restarted search algorithm and  $f(t)$  its respective probability density function.

Despite its optimality, the restart strategy with a fixed cutoff has two serious drawbacks. The first one is concerned with proving that no solution to a search problem actually exists, because without storing the information about already explored nodes of the search tree, the search might run forever. On the other hand, if such information is stored, the runs of the algorithm are no longer independent and the analysis of the restarted strategy becomes more complicated.

The second drawback lies in the fact, that we cannot determine the optimal cutoff without prior knowledge about the distribution. An incorrectly chosen cutoff can then decrease the performance of the restarted algorithm. Furthermore, solutions to certain problems cannot in principle be found without searching a minimum number of nodes. If the cutoff happens to be smaller than this number, the expected runtime will obviously be infinite. Although it is unlikely that one would choose such a small cutoff that would result in the infinite expected runtime, it is generally less harmful to choose the cutoff too high rather than too small.

A way to get rid of the above-mentioned drawbacks, is to use a varying cutoff instead of the fixed one. Luby [29] proposed a universal strategy with varying cutoff given by the sequence

$$(t_0, t_0, 2t_0, t_0, t_0, 2t_0, 4t_0, t_0, t_0, 2t_0, t_0, t_0, 2t_0, 4t_0, 8t_0, t_0, \dots) \quad (2.12)$$

He has proven a bound on the expected runtime  $E[T_{universal}]$  of this strategy with respect to the expected runtime  $l^*$  of an optimal restart strategy

$$E[T_{universal}] \leq 192 \cdot l^*(\log_2(l^*) + 5) \quad (2.13)$$

and showed that this strategy is optimal in the sense that no other restart strategy can do any better up to a multiplicative constant if no information about the distribution is known. Note that to make statements about performance of Strategy 2.12 sensible, one must work with  $t_0$  equal to the smallest amount of time being considered. In an extreme setting, when  $t_0$  is too high, Luby's strategy might even reduce to a non-restarted search.

In practice, the choice of  $t_0$  can greatly affect the performance of Luby's universal strategy. By incorrectly choosing  $t_0$  too high, the restart strategy could even lose its optimality and the bound 2.13 would be no longer valid. On the other hand, by making it too small, the overall search time could be penalized by costs needed for the initialization of new runs of the algorithm, which is not considered in Luby's analysis.

Another restart strategy called geometric restart strategy was proposed by Walsh [47] and has the form

$$(t_0, r \cdot t_0, r^2 \cdot t_0, r^3 \cdot t_0, r^4 \cdot t_0, \dots) \quad (2.14)$$

There is no theoretical bound on the performance of this strategy, so it might be arbitrarily worse than the optimal fixed cutoff strategy on some (hopefully pathological) runtime distributions. It, however, exhibits good average runtime on many real problems [49] on which it usually outperforms Luby's universal strategy. This is also the reason why we prefer the geometric restart strategy to Luby's strategy in our experiments in Chapter 3.



# Chapter 3

## Two $\theta$ -subsumption Algorithms

In this chapter, we develop two algorithms for the  $\theta$ -subsumption problem - a complete randomized restarted  $\theta$ -subsumption algorithm RESUMER2 [26] and a heuristic coverage estimation algorithm RECOVER [24]. In Section 2.3 we have shown that  $\theta$ -subsumption is equivalent to finite-domain constraint satisfaction problems. It might therefore seem that there is no need for dedicated  $\theta$ -subsumption algorithms and that it would suffice to use state-of-the-art methods from the field of constraint satisfaction. This is true only to some extent, as there are several important differences between general constraint satisfaction problems and  $\theta$ -subsumption problems encountered in real applications. First, CSP algorithms are typically designed for solving as large (and hard) problems as possible in as short time as possible. A typical CSP task is to solve a hard problem within a given time limit. On the other hand, in ILP, the respective  $\theta$ -subsumption problems are usually not particularly hard and we are not interested in solving the hardest instances in as short time as possible, rather, we are interested in solving hundreds of thousands (typically small) problems as fast as possible. Therefore, in ILP, we also care about the easy problems, as we try to solve as many problems as possible in a given time limit. Second, a specific feature of CSP problems in ILP (corresponding to  $\theta$ -subsumption problems) is that we typically solve a number of problems with a fixed constraint structure, but with different constraint languages, i.e. we try to solve  $\theta$ -subsumption problems for a fixed hypothesis and different examples. In this chapter, we try to exploit these special properties of  $\theta$ -subsumption at least to some extent.

---

**Algorithm 4** *SubsumptionCheck*( $C, e$ ): A simple subsumption test algorithm
 

---

```

1: Input: Clause  $C$ , example  $e$ ;

2: if  $C \subseteq e$  then
3:   return YES
4: else
5:   Choose variable  $V$  from  $C$  using a heuristic function /* see main text */
6:   for  $\forall S \in PossibleSubstitutions(V, C, e)$  /* see main text */ do
7:      $SearchedNodes \leftarrow SearchedNodes + 1$ 
8:     /* SearchedNodes is a global variable initiated to 0 prior to executing this algo-
9:       rithm. */
9:      $C' \leftarrow$  Substitute  $V$  with  $S$ 
10:    /* Method LookAhead is described in main text */
11:    if  $LookAhead(C', V, S) = YES$  then
12:      if  $SubsumptionCheck(C', e) = YES$  then
13:        return YES
14:      end if
15:    end if
16:  end for
17:  return NO
18: end if

```

---

## 3.1 Runtime Distributions

### 3.1.1 Basic Algorithm

In this subsection, we describe a simple heuristic algorithm (Algorithm 4) for verifying whether a clause  $C$   $\theta$ -subsumes an example  $e$ . This simple algorithm will be used in subsequent sections to develop restarted randomized algorithms RESUMER and RESUMER2. Similarly to Django [30] this algorithm is inspired by the CSP framework. It is a backtracking search algorithm with a variable selection heuristic and randomization. The heuristic function aims at choosing variables whose substitution makes it likely that an inconsistency, if one exists, is detected soon. If no variable has been selected yet (i.e. we are on the start of the search), the first variable to be substituted is selected randomly with probability proportional to number of its occurrences in the given clause. Other-

---

**Algorithm 5** *SubstitutionPossible*( $V, T, C, e$ ): Returns NO if  $C$  cannot subsume  $e$  when  $V$  is grounded to  $T$ . (The reverse implication may not hold, see main text.)

---

```

1: Input: Variable  $V$ , constant  $T$ , clause  $C$ , example  $e$ ;
2: for  $\forall A \in C$  such that atom  $A$  contains variable  $V$  do
3:    $A' \leftarrow$  replace all occurrences of variable  $V$  in atom  $A$  by  $T$ .
4:   if  $\forall \theta. A'\theta \not\subseteq e$  /* easy to check for a single atom  $A$  */ then
5:     return NO
6:   end if
7: end for
8: return YES

```

---



---

**Algorithm 6** *LookAheadFC*( $C, V, S, e$ ): Returns NO if  $C$  cannot subsume  $e$  when  $V$  is grounded to  $S$ . (The reverse implication may not hold.)

---

```

1: Input: Clause  $C$ , variable  $V$ , constant  $S$ , example  $e$ ;
2: if  $\forall W \in \text{Adjacency}(V) : \text{PossibleSubstitutions}(W, P', e) \neq \emptyset$  then
3:   return YES
4: end if
5: return NO

```

---

wise, for a variable  $V$ , the heuristic function computes weighted sum of occurrences of variables in clause  $C$  that have already been grounded and that share at least one literal with  $V$ . The weights are proportional to rareness of the particular literals in which the variable is contained. This sum is then multiplied by  $1 + \frac{1}{D}$ , where  $D$  is an upper bound on the size of the domain of  $V$  computed in the initialisation phase of the algorithm's run. The variable which maximizes this function is selected; in case of a tie, a random choice is made with uniform probability among the highest scoring variables. Function *PossibleSubstitutions*( $V, C, e$ ) returns all constants  $T$  (in random order) for which *SubstitutionPossible*( $V, T, C, e$ ) (Algorithm 5) returns YES. The function prunes away a subset of possible groundings for  $V$  whose inclusion in  $\theta$  would imply  $C\theta \not\subseteq e$ . In general though, not all such groundings are detected by the function. Function *LookAhead* used on line 11 of Algorithm 4 may be one of the look-ahead methods from CSP framework: nothing or forward checking (Algorithm 6). Arc-consistency can be also applied to reduce domains of variables prior to search, which is not shown in Algorithm 4 for simplicity.

### 3.1.2 Subsumption Test Runtime Distributions

To obtain runtime distributions of the algorithm, we tested it on randomly generated hypotheses and examples. We devised two different graph generators for this purpose. The first generates Erdos-Rényi random graphs where any two vertices are connected with a probability  $p$  (by an edge of a random orientation). The second produces scale-free graphs; here, an edge is attached to a vertex with probability increasing with the number of edges already connected to the vertex. In both algorithms, all vertices are colored as black with probability 0.5 and red otherwise. We will refer to parameter  $p$  ( $k$ , respectively) of a random graph as the *connectivity* of the class of random graphs. Instances of  $\theta$ -subsumption problem will be created by converting the graph representations to first order logic representations. In what follows, we will speak about numbers of variables (for hypotheses) and numbers of constants (for examples) instead of numbers of vertices. On the other hand, we will use the term *connectivity* even in first order logic representations of the generated data.

We performed experiments with random sets of hypotheses and examples generated by both of the random graph generators, under various settings of  $n$  and  $p$  ( $n$  and  $k$ , respectively). Algorithm 4 was used with forward checking as its look-ahead method for the experiments. Our objective was to verify the presence of *heavy tails* in the runtime distributions  $F(t)$ . For a  $t > 0$ ,  $F(t)$  is the probability that the tested algorithm resolves a random subsumption instance with at most  $t$  explored search nodes. Recall that, informally, a heavy-tailed distribution indicates the non-negligible probability of subsumption instances with extremely long runtime.

Recall from Section 2.5, that the presence of a heavy tail in an empirical runtime distribution  $F(t)$  may be approximately checked graphically, by plotting  $1 - F(t)$  against  $t$  on a log-log scale. In the case of a power-law distribution, this plot acquires a linear shape [14]. Even though more sophisticated statistical methods for checking heavy-tailedness of a distributions exist, we will not need them in this thesis, as we are mainly interested in speeding-up the search and not in the distributions itself. Interested reader may find a short discussion of these issues in [23], where parameters of generalized Pareto distributions for subsumption test runtimes were estimated.

We have performed a series of experiments in the phase transition framework, which was imported to relational learning by Giordana et. Saitta [12]. Phase transition framework studies the correspondence between parameters of hypotheses and probability  $p_{subs}$  that a randomly selected hypothesis  $C$  with these parameters  $\theta$ -subsumes an example  $e$ .

A phase transition spectrum is divided into three disjoint regions: the YES region, the NO region and the PT (phase transition) region. For hypotheses corresponding to the YES region the probability  $p_{subs}$  is close to 1, as hypotheses are mostly underconstrained in this region, while it is nearly 0 for hypotheses corresponding to the NO region, as these are mostly overconstrained. The PT region then corresponds to the region where  $p_{subs}$  usually quickly drops from being close to 1 to being almost 0 and subsumption checking is usually computationally most expensive for hypotheses in this region. The computational difficulty in the transition region is due to hypotheses being neither overconstrained nor underconstrained. For an extensive general introduction to phase transitions in computational complexity, see the seminal paper [20].

Our experiments in phase transition framework revealed a systematic progression from heavy-tailed regimes corresponding to configurations located in the YES region (low connectivity in hypotheses) of the phase transition spectrum to non-heavy-tailed regimes corresponding to configurations located in the NO region (high connectivity in hypotheses). This observation agrees with the previous study [6]. This progression is shown in Fig. 3.1 for the Erdos-Renyi graph data (Algorithm 15). The same trends were observed for the small-world graph data. The complementary runtime distributions plotted refer to subsumption checks between examples with fixed numbers of variables and with fixed connectivity and hypotheses with fixed numbers of constants and connectivity changing among particular distributions.

The complementary runtime distributions plotted in the top left panel of Fig. 3.1 refer to satisfiable problem instances, i.e. those where the hypotheses  $\theta$ -subsume the examples. The hypotheses had  $n = 15$  variables (vertices in the underlying random graphs) and connectivity consecutively  $p = 0.1$ ,  $p = 0.15$ ,  $p = 0.2$ ,  $p = 0.25$ . The examples had  $n = 50$  constants and connectivity  $p = 0.3$ . The complementary distributions in the top right panel of Fig. 3.1 refer to unsatisfiable problem instances with hypotheses with  $n = 15$  variables and connectivity consecutively  $p = 0.15$ ,  $p = 0.2$ ,  $p = 0.25$ ,  $p = 0.3$ ,  $p = 0.35$ . The examples had  $n = 50$  constants and connectivity  $p = 0.3$ . The bottom left panel displays the phase transition spectrum corresponding to the complementary runtime distributions displayed in the top panels. Finally, the bottom right panel displays complementary runtime distributions for basic restarted algorithm on satisfiable and unsatisfiable instances. The runtime distributions for both satisfiable and unsatisfiable instances have heaviest tails when the problem instances are located in the YES region and they decay faster as the problem instances get closer to the NO region. A difference between runtime distributions for satisfiable and unsatisfiable problem instances

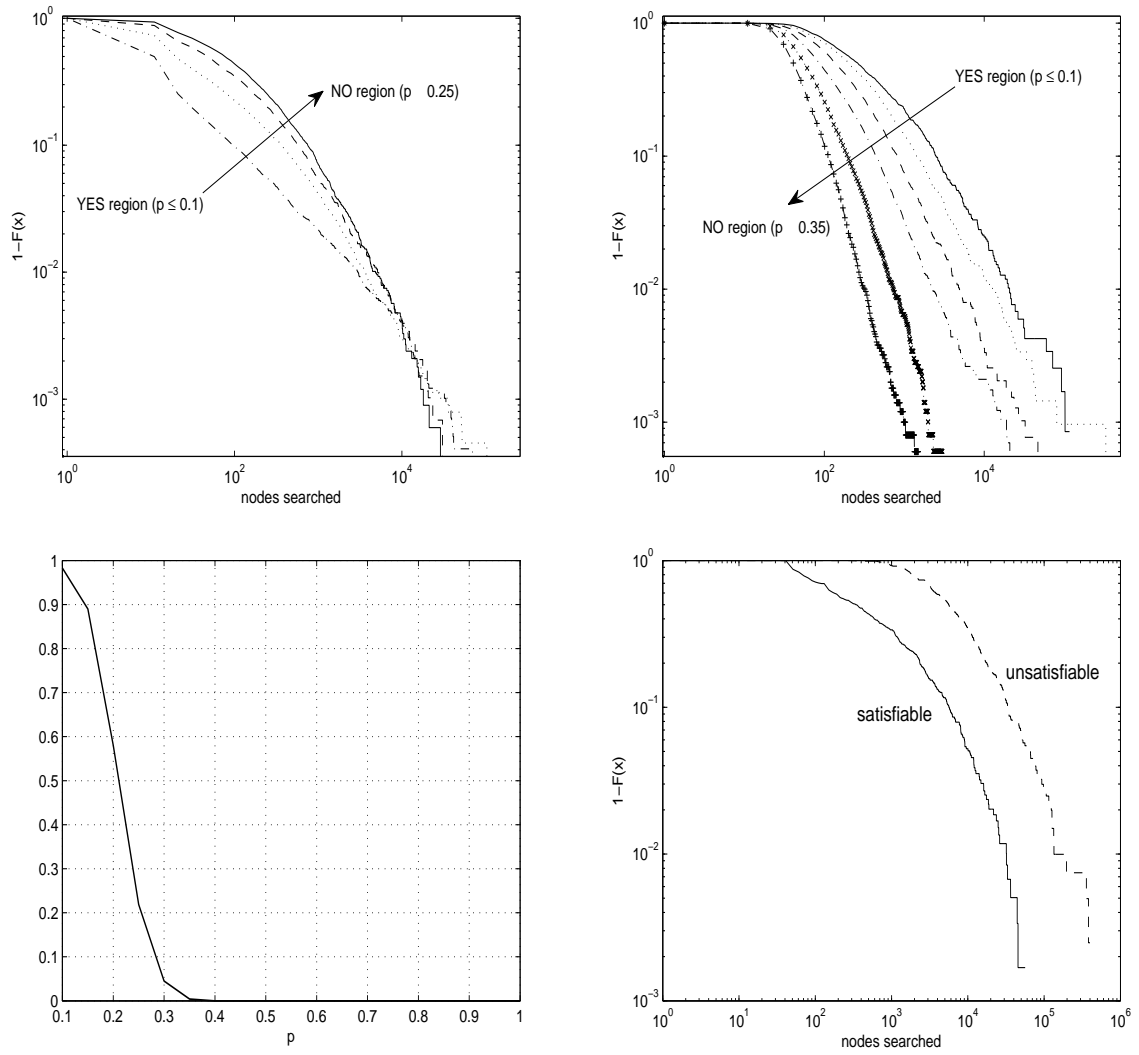


Figure 3.1: Complementary runtime distributions for random graph data

is observed in the probability of very short runs, which is high for the satisfiable instances located in the YES region, but which is negligible for the unsatisfiable instances in that region. As a consequence, individual distributions cross each other in the top left panel of Fig. 3.1 whereas this effect is not observed in the top right panel of Fig. 3.1.

## 3.2 RESUMER: A Restarted $\theta$ -subsumption Algorithm

In this section, we develop a randomized restarted  $\theta$ -subsumption algorithms RESUMER and RESUMER2. Restarting search process of a backtracking-style algorithm has been shown very useful empirically (e.g. [14]) and theoretically [13] when runtime distributions exhibited heavy-tailed behaviour. We show that restarts can significantly decrease runtime of the basic  $\theta$ -subsumption algorithm (Algorithm 4). We also propose and test several heuristic techniques to speed-up the search even more. We allow a limited communication between subsequent restarts. We make the strength of constraint propagation increase with the number of unsuccessful restarts. Finally, we also allow the algorithm to exploit information about successful variable orders in previous subsumption checks with the same hypothesis but with a different example.

### 3.2.1 Designing a Restarted Subsumption Test Algorithm

While the presence of heavy tails for some classes of subsumption instances indicates possible large runtime benefits achievable by a restarting strategy [14], its effect on the non-heavy-tailed classes may not be necessarily very detrimental. We thus decided to assess the overall impact of restarting empirically. For this sake we designed a complete restarted randomized subsumption algorithm RESUMER (Algorithm 7). Its completeness is guaranteed by the assumption that for the cutoff sequence  $R(n)$ ,  $R(n) \rightarrow \infty$  as  $n \rightarrow \infty$ , because then there is always such  $n_0$ , for which the cutoff  $R(n_0)$  enables Algorithm 7 to explore the whole search tree. Recall that variable selection heuristic on line 5 of Algorithm 4 is randomized and that randomization of the ordering of possible groundings is used.

The complementary runtime distributions for RESUMER, with an ad-hoc chosen

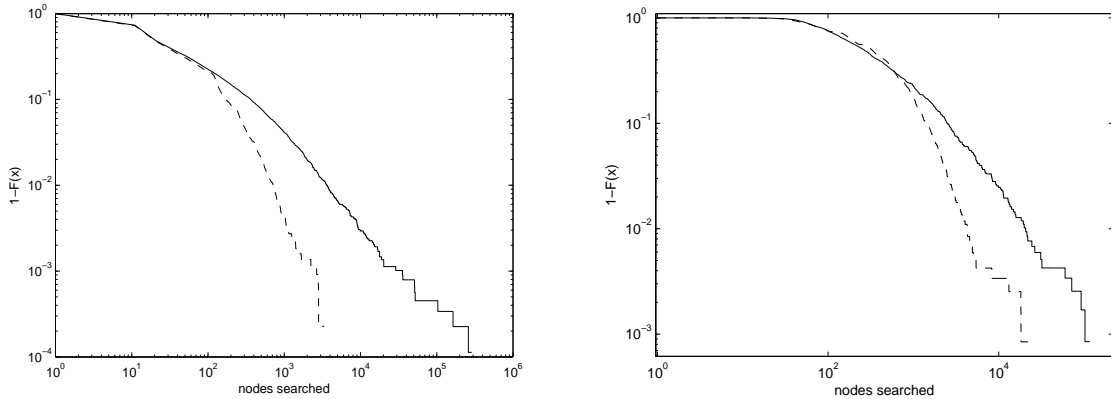


Figure 3.2: Effect of restarts for satisfiable (left) and unsatisfiable (right) instances

restart sequence

$$R(n) = \lfloor 10 \cdot e^n + 100 \rfloor$$

are plotted in Fig. 3.2 for two of the cases exemplified also in Fig. 3.1. The hypotheses have  $n = 15$  variables and connectivity  $p = 0.15$ . In both cases, examples had  $n = 50$  constants and connectivity  $p = 0.3$ . Both hypotheses and examples were randomly generated by a generator of Erdos-Rényi random graphs. In both of these cases, restarts reduce runtime, although the difference is much more significant in the satisfiable case. Of relevance, the times taken by RESUMER on the unsatisfiable instances were in some of our experiments, where examples were significantly larger than hypotheses, about  $10^2$  times higher than on the satisfiable ones, which will be further exploited in Section 3.3.1 where algorithm RECOVER is developed. This has two main reasons. First, the ‘iterative’ character of RESUMER has to be taken into account; while satisfiability can in principle be shown in any single restart, unsatisfiability can only be shown after  $n$  restarts making  $R(n)$  sufficiently high. Second, when dealing with satisfiable and unsatisfiable problems generated with the same parameters, satisfiable instances seem to be easier even for the non-restarted basic algorithm.

It is further possible to increase the performance of RESUMER by a heuristic modification of the basic restarted strategy (Algorithm 8), which repeatedly calls Algorithm 4. The idea is to allow a certain transfer of knowledge between individual restarts by guiding the initial selection of a variable to be substituted in Algorithm 4 (line 4). In particular, when this algorithm is called from RESUMER, we choose the variable which caused the last backtrack in the previous restart, i.e. the last variable  $V$  which yielded



---

**Algorithm 7** *ReSumEr*( $C, e, R$ ): A restarted subsumption algorithm

---

```

1: Input: Hypothesis  $C$ , example  $e$ , cutoff sequence  $R$ ;
2:  $n \leftarrow 1$ 
3: repeat
4:    $Answer \leftarrow$  Run SubsumptionCheck( $C, e$ ) with number of searched nodes limited
     to  $R(n)$ 
5:    $n \leftarrow n + 1$ 
6: until  $Answer$  YES or NO is returned
7: return  $Answer$ 

```

---

an empty set  $PossibleSubstitutions(V, P, e)$ . The rationale for this modification is that the variable which caused backtracking is more likely to be highly constrained than a randomly chosen variable. As such it is a good candidate to start the search with.

This straightforward modification (called *restart strategy with dependent runs*) has the consequence that pairs of restarts are no longer statistically independent trials. In general, this might represent a problem. It is known that a restarted strategy exhibits the desirable property of exponentially decaying runtime if individual restarts are independent. For this reason we can allow the above described transfer of knowledge only from odd restarts into the subsequent even restarts, resulting in a series of restart pairs, which are mutually independent. Thus we maintain the exponential decay guarantee.<sup>1</sup> Although we prefer to use this outlined *odd/even* strategy, here, we also show that the first strategy, where the first variable picked by the algorithm in a restart is equal to the variable on which the previous restart failed, also guarantees exponential decay of the complementary runtime distribution under some natural assumptions.

We will analyze the problem in the idealized setting where the respective probability density of runtime may be non-zero even for values higher than is the size of the whole search space, because this will allow us to work in the asymptotical setting. While this assumption typically holds for randomized restarted algorithms with cutoffs that do not allow the algorithm to search through the whole search space, it usually does not hold in other cases. However, if we did not stick to this idealized model, claims such as  $1 - R(t)$  *decays exponentially fast* would not be very informative because any  $1 - R(t)$ , which is zero for  $t > t_0$  for some  $t_0$ , is bounded by  $c \cdot e^{-\alpha t}$  for some  $\alpha, c \in R^+$ .

---

<sup>1</sup>Not all sequences of cutoffs lead to an exponential decay guarantee, but the strategy with knowledge transfer between each two consecutive restarts does not necessarily possess this property even if the respective cutoff sequence guarantees it for the case of independent restarts.

---

**Algorithm 8** *ReSumEr2*( $C, e, R$ ): A modified restarted subsumption algorithm

---

```

1: Input: Hypothesis  $C$ , example  $e$ , cutoff sequence  $R$ ;
2:  $n \leftarrow 1$ 
3: repeat
4:   if  $n$  is odd then
5:      $Answer \leftarrow$  Run SubsumptionCheck( $C, e$ ) with number of searched nodes limited
       to  $R(n)$  and record  $LastVariable \leftarrow$  the last variable that caused backtracking
       there in.
6:   else
7:      $Answer \leftarrow$  Run SubsumptionCheck( $C, e$ ) with number of searched nodes limited
       to  $R(n)$  and the first checked variable set to  $LastVariable$ 
8:   end if
9:    $n \leftarrow n + 1$ 
10: until  $Answer$  YES or NO is returned
11: return  $Answer$ 

```

---

The next theorem does not refer directly to our  $\theta$ -subsumption algorithms. It rather treats the general case when we have an algorithm with some runtime distribution. We will suppose that the runtime distribution of the algorithm is a mixture of at most  $k$  runtime distributions, where each component of the mixture corresponds to some fixed *initial conditions*. Note that in the case of our  $\theta$ -subsumption algorithms initial conditions refer to selection of the first variable to be assigned a value.

**Theorem 3.1:** *Let  $g(t)$  be a real valued function and let  $A$  be the set of all  $\alpha \in \mathbb{R}$  such that  $g(\alpha \cdot t)$  is a decreasing function of  $t$ . Let  $C(n) : \mathbb{N} \rightarrow \mathbb{N}$  be a sequence of cutoffs having the property that, for any runtime distribution with positive probability of finding a solution within  $C(n)$  time units for all  $n \in \mathbb{N}$ ,  $C(n)$  guarantees that there is  $\alpha \in A$  such that the restarted distribution decays as  $O(g(\alpha \cdot t))$  (e.g. exponentially). Then a strategy, which uses  $C(n)$  as its restart sequence and which may select initial conditions of each restart deterministically using information about previous restarts, also guarantees that, for any runtime distribution, which has  $k \in \mathbb{N}$  possible initial conditions each leading to a run with positive probability of finding a solution in  $C(n)$  time units for all  $n \in \mathbb{N}$ , there is  $\beta \in A$  such that decay of the restarted runtime distribution is of order  $O(g(\beta \cdot t))$ .*

**Proof:** Let  $F(t) = \sum_{i=1}^k p_i f_i(t)$  be the probability that a non-restarted algorithm fails to find solution within  $t$  time units (where  $f_i(t)$  is the probability that the algorithm fails

to find solution within  $t$  time units conditioned that it started with initial conditions  $i$ ) and let  $f_i(t) > 0$  for all  $t > 0$ . First, we bound the probability  $R_{dep}(t)$  that the restarted algorithm with dependent runs fails to find solution within  $t$  time units.

$$R_{dep}(t) \leq \max_j f_j(t - \sum_{l=1}^m C(l)) \cdot \prod_{i=1}^m \max_j f_j(C(i)), \quad m = \max_i \{i : \sum_{j=1}^i C(j) \leq t\} \quad (3.1)$$

Now, we will proceed indirectly. Instead of playing with Eq. 3.1, we will construct such a runtime distribution (non-restarted) whose complementary runtime distribution after application of independent restarts will be equal to the right-most expression in Eq. 3.1. The next complementary distribution<sup>2</sup> has the desired properties

$$R_{synth}(t) = \max_j f_j(t)$$

Indeed, the probability that the restarted algorithm with independent runs does not find solution within  $t$  time units is

$$\begin{aligned} R_{rest}(t) &= R_{synth}(t - \sum_{l=1}^m C(l)) \cdot \prod_{i=1}^m R_{synth}(C(i)) = \\ &= \max_j f_j(t - \sum_{l=1}^m C(l)) \cdot \prod_{i=1}^m \max_j f_j(C(i)) \geq R_{dep}(t), \quad m = \max_i \{i : \sum_{j=1}^i C(j) \leq t\} \end{aligned} \quad (3.2)$$

Since we have assumed that  $C(i)$  guarantees to make runtime distribution of any restarted algorithm with independent restarts to be of order  $O(g(\alpha \cdot t))$  for some  $\alpha \in A$ , we also have that  $R_{dep}(t)$  decays like  $O(g(\beta \cdot t))$ , which finishes the proof.  $\square$

A key assumption of the above theorem is that there is always non-negligible probability of finding a solution for any initial conditions. It is easy to see that this is satisfied for our randomized  $\theta$ -subsumption algorithm (Algorithm 4) as the value ordering heuristic returns values in random order (selecting permutations with uniform probability). If the algorithm is given enough time (i.e. if it can search more than  $|vars(C)|$  nodes of the search tree) then for probability  $p_{succ}$  of finding a solution of  $C \preceq_{\theta} e$  it holds  $p_{succ} \geq |vars(e)|^{-|vars(C)|}$ .

It might seem from the discussion above that allowing communication between subsequent restarts does not harm theoretical guarantees. However, if we weaken the assumptions and assume that there may be countably many distinct initial conditions then the strategy with dependent restarts may give rise to a heavy-tailed distribution even for

<sup>2</sup>It is easy to check that  $R_{synth}(t)$  is indeed a complementary distribution.

fixed-cutoff strategy. On the other hand, the strategy with independent restarts will still guarantee exponential decay if there is a positive probability of finding a solution in a single restart. This is illustrated by the next example.

**Example 3.1:** Let there be a countable set  $C$  of possible initial conditions of a randomized algorithm  $A$  and let  $F = \{1, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \dots\}$  be probabilities of  $A$  failing to find solution within  $t$  time units when started with initial conditions  $i$ . Further, we assume that the randomized algorithm  $A$  selects initial conditions  $i$  with probability  $(1-p)^{i-1} \cdot p$ . We now compute the probability that  $A$  does not find any solution within  $t$  time units:

$$\begin{aligned} p_{fail} &= \sum_{i=1}^{\infty} \frac{i}{i+1} (1-p)^{i-1} \cdot p = -p \cdot \sum_{i=1}^{\infty} \frac{d}{dp} \frac{(1-p)^i}{i+1} = -p \cdot \frac{d}{dp} \left( \frac{1}{1-p} \sum_{i=2}^{\infty} \frac{(1-p)^i}{i} \right) = \\ &= -p \cdot \frac{d}{dp} \left( \frac{p-1-\ln p}{1-p} \right) = \frac{1+p \cdot \ln p - p}{(1-p)^2} \end{aligned}$$

Note that the derivation shown above is correct for  $p \in (0, 1)$  as the power series  $\sum_{i=1}^{\infty} \frac{i}{i+1} (1-p)^{i-1}$  converges uniformly on every closed interval  $[\epsilon, 1]$  where  $\epsilon \in (0, 1)$ . The reason why we performed this tiresome derivation was to show that  $0 < p_{fail} < 1$  for  $p \in (0, 1)$ , i.e. that the algorithm behaves well. From this we can deduce that the complementary runtime distribution of the restarted strategy with fixed cutoff  $t$  and independent runs decays exponentially (cf. Theorem 2.1).

Now, suppose that the restarted algorithm with dependent runs has the property that if solution is not found in  $i$ -th restart, the next run starts with initial conditions  $i+1$ . The probability that no solution is found within first  $k$  restarts is<sup>3</sup>

$$R_{dep}(k) = \frac{1}{2} \cdot \frac{2}{3} \cdot \frac{3}{4} \cdot \dots \cdot \frac{k}{k+1} = \frac{1}{k+1}$$

So, we see that in spite of the fact that the series of cutoffs  $t, t, t, \dots$  leads to exponentially decaying tail for statistically independent runs, it leads to a heavy-tailed distribution for the restart strategy with dependent runs.

In order to better understand the differences between the setting assumed in Theorem 3.1 and the setting used in this example, it is illustrative to see where the proof of Theorem 3.1 fails for the latter setting. It is the fact that since  $F$  is infinite, there need not exist  $\max_j f_j(t)$ , which is also the case in this example.  $\triangle$

---

<sup>3</sup>For the restart strategy with fixed cutoff it suffices to show that the complementary runtime distribution decays exponentially (has a heavy tail, respectively) as a function of number of restarts because the case for complementary runtime distributions as a function of time follows from it.

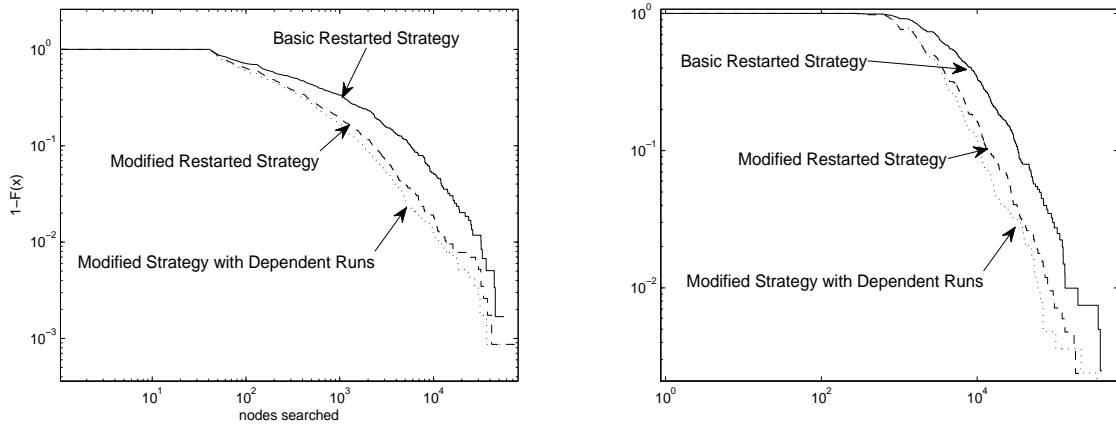


Figure 3.3: Effect of individual restart strategies

The examples above showed that the strategy with dependent restarts may be in some sense weaker than the strategy with independent restarts in situations when there are infinitely (countably) many initial conditions from which the algorithm may start its restarts. In spite of the fact that this setting does not seem to emerge in any practical situation, it suggests that we should be cautious when designing restart strategies with dependent restarts. The reason is that if there is a sufficient number of possible initial conditions, there could be a  $t_0$  making the distribution decay at a power-law rate for  $t < t_0$ . Furthermore, no real distribution considered in this thesis is heavy-tailed in the strong sense, as discussed in Section 2.5, because runtime of the basic complete non-restarted  $\theta$ -subsumption algorithm is always bounded by size of the particular search tree. Nevertheless, it is still very helpful to use restarts for the distributions with tails decaying at power-law rate only for a limited range of  $t$ . Hence, it might be the case that an algorithm using the strategy with dependent runs would have a runtime distribution whose mean could be decreased by restarting this restarted strategy. And in fact, this is a possible way to view the *odd/even* strategy - as a restarted version of the strategy with dependent runs.

The effect of using the basic restarted strategy, the strategy with dependent runs and the *odd/even* strategy is shown in Fig. 3.3 for satisfiable (left panel) and unsatisfiable instances (right panel). The hypotheses have  $n = 20$  variables and connectivity  $p = 0.2$ . In both cases, examples had  $n = 100$  constants and connectivity  $p = 0.3$ . The restart sequence was  $C(n) = \lfloor 10 \cdot e^n + 100 \rfloor$ . Both hypotheses and examples were randomly generated by Algorithm 15. It seems clear that the additional gain of the strategy with

dependent runs over *odd/even* strategy is marginal and does not seem to be significant enough to face its undesirable properties discussed in the paragraphs above.

We can speed up the restarted algorithm even more by considering two additional heuristic improvements. First, notice that since the variable ordering heuristic is randomized, it produces variable orderings of different quality, i.e. producing bigger or smaller search trees. We might assume that typically if a solution was found in some restart (or it was proven that there is no solution), the variable ordering used in this restart was not very bad. Therefore we might transfer this ordering to the next  $\theta$ -subsumption check between the same hypothesis and the next example. Thus we exploit the fact that in ILP, we usually need to decide subsumption relation for a hypothesis and a set of examples. This is typical for ILP and is not typically encountered in the field of constraint satisfaction. Second, not all instances of  $\theta$ -subsumption problems encountered in ILP are generally very hard (compared to problems typical in constraint satisfaction), so it can be undesirable to use immediately the strongest propagation methods as these methods have a non-negligible overhead and become favorable only for sufficiently hard problem instances. Therefore we associate with each implemented propagation method (nothing, forward-checking and arc-consistency on binary projection of constraints) a natural number denoting the first restart<sup>4</sup>, in which the particular propagation should be applied. Thus, we avoid using overly strong methods for easy problems. On the other hand, we do not face the burden caused by having a big search tree, which could likely be pruned by sufficiently strong propagation methods, for harder problem instances.

We denote the version of the algorithm with the described knowledge transfer between odd and even restarts and enriched with other heuristic improvements described above as RESUMER2. In the next section, we evaluate RESUMER2 on artificial and real-world data and compare it with a state-of-the-art  $\theta$ -subsumption algorithm Django.

### 3.2.2 Experimental Evaluation

We subjected RESUMER2 to a comparative empirical evaluation with a state-of-the-art  $\theta$ -subsumption algorithm Django [30]. All experiments were conducted on the same computer. Django was implemented in C while RESUMER2 was implemented in Java. In [26], we performed experiments with a version of RESUMER2 implemented in C. Since C is generally faster than Java, the C version may outperform the Java version for small

---

<sup>4</sup>These are parameters that the user can tune for a particular application.

problem instances, where the implementation actually matters, even though it does not use all the heuristics, which are present in the Java version. Therefore, we evaluate also the C version of RESUMER2 in Section 3.2.2.2 where we show that RESUMER2 can outperform Django even for small hypotheses, which are in reach of current ILP systems. We used Django version 11.

### 3.2.2.1 Generated Data

In this section, we evaluate RESUMER2 and Django on generated random graph data. We aim at measuring the runtime distributions both for the case when there is a big difference between sizes of hypotheses and examples and for the case when there is not a very big difference.

Figure 3.4 displays the results for hypotheses and examples generated as Erdos-Rényi random graphs. The comparative runtimes (top panels) are accompanied by the corresponding phase transition diagrams (bottom panels). The left (right, respectively) panels correspond to a smaller (larger, respectively) size difference between the hypotheses and the examples. Size is understood as the number of contained vertices in the underlying graphs (variables in the respective hypotheses or constants in the respective examples). The connectivity parameter of random graphs was  $p = 0.3$  for examples and it varied for hypotheses (shown on the horizontal axis). In the top left panel, the hypotheses have 30 variables and examples have 100 constants. In the top right panel, hypotheses have 10 variables and examples have 200 constants. All shown points are averages of 50 computations of subsumption of a random hypothesis against 20 random examples. In the bottom, the phase transition landscapes (i.e. the probability that a random hypothesis with connectivity  $p$  subsumes a random example) for the respective settings from the top panels are shown.

We now note on the principled trends apparent from the results. First, RESUMER2 outperformed Django in the YES region of the phase transition spectrum. This region corresponds to the left parts of all diagrams in Figure 3.4. Although the observed absolute difference is larger in the NO region, in relative terms it is much smaller than the difference in the YES region. Second, in the experiments with a larger size-difference between the hypotheses and the examples (examples much larger), RESUMER2 was faster across the entire phase transition domain. Third, heavy-tailed behavior of Django was observed: in spite of its typical measured runtimes in the order of milliseconds to seconds, occasional runs in satisfiable instances took up to tens of minutes and had to be curtailed. This

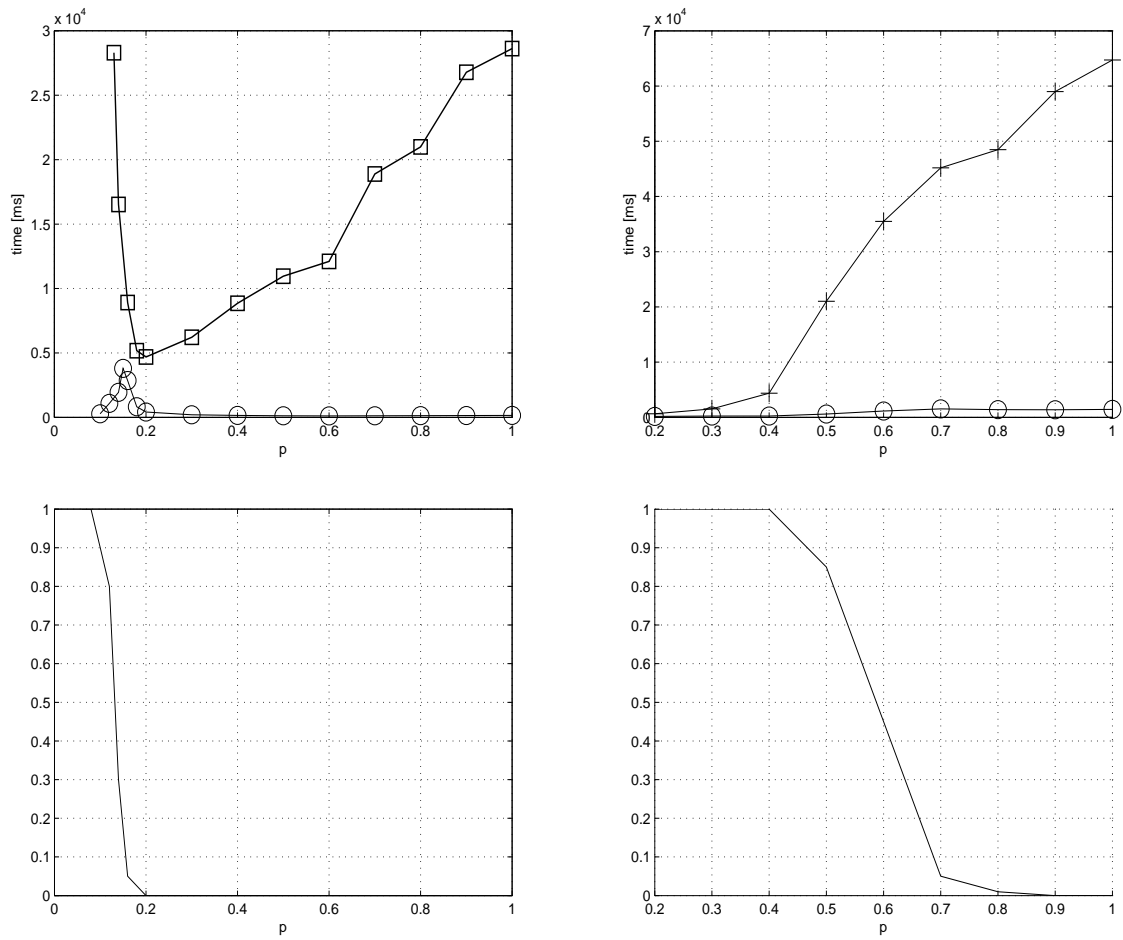


Figure 3.4: Comparison of RESUMER2 ( $\circ$ ) and Django ( $\square$ ) on Erdos-Rényi random graphs.



resulted in Django’s excessive runtimes in the top-left panel of Fig. 3.4 for  $p \leq 0.1$ . Heavy-tailed behavior is prevented by RESUMER2 resulting in its vast superiority in the  $p \leq 0.1$  region of Fig. 3.4, top-left panel. In some of our experiments, however, RESUMER2’s averaged runtimes were also excessive and Django outperformed it. Unlike for Django, here the reason was not in occasional excessive runs, but rather in the systematic increase of runtime required to complete the unsatisfiable subsumption instances.

### 3.2.2.2 Predictive Toxicology Challenge Data

Next, we studied performance of RESUMER2 on a real-life dataset from the Predictive Toxicology Challenge (PTC) [18]. The PTC dataset consists of 344 organic molecules marked according to their carcinogenicity on male and female mice and rats. Our relational-logic representation of these molecules consisted of ternary literals for atomic bonds  $bond(at1, at2, bondName)$ , unary literals representing types of particular bonds  $singleBond(bondName)$ ,  $doubleBond(bondName)$ ,  $tripleBond(bondName)$  and  $resonantBond(bondName)$  and unary literals for atom types  $c(atom)$ ,  $h(atom)$ ,  $n(atom)$  etc.

In [26], we performed experiments in this domain, in which we generated hypotheses by random walking through the subsumption lattice. These experiments demonstrated that RESUMER2 was significantly faster for long hypotheses in this real-world domain. This was possible due to the fact that random walks enabled us to encounter long hypotheses during the search. On the other hand, here we also want to demonstrate that RESUMER2 can significantly outperform Django even for relatively short hypotheses. Therefore we performed a series of experiments with a naive exhaustive best first search algorithm with limit on size of hypotheses  $|lits(C) \setminus \{l \in lits(C) : pred(l) = different\}| \leq 10$  and  $|lits(C) \setminus \{l \in lits(C) : pred(l) = different\}| \leq 100$ . We let the search algorithm expand 50 search nodes in each run and let it run 152 times each time with different bottom clause (i.e. once for each positive example). The runtimes in Table 4.5 refer to average time of a single run of the search algorithm with the particular subsumption algorithm.

The results of the PTC dataset experiments, summarized in Table 4.5, differ according to the chosen relational logic representation of the molecules. The first version **PTC-v1** uses a rather naive representation. Here, each molecular bond is represented by two literals  $bond(at1, at2, bondName)$ ,  $bond(at2, at1, bondName)$ . A source of imprecision of this representation is that two variables in a hypothesis may represent the same atom, which does not make intuitive sense. For this setting, the difference in performance

Algorithm	PTC-v1 [s]	PTC-v2 [s]	PTC-v3 [s]
RESUMER2 (JAVA)	12.9	318	28.6
RESUMER2 (C)	5.9	1580	11.8
Django (C)	8.2	n.a.	120.3

Table 3.1: Average runtimes for hypothesis search for the PTC dataset and max. hypothesis size 10.

Algorithm	PTC-v1 [s]	PTC-v2 [s]	PTC-v3 [s]
RESUMER2 (JAVA)	19.3	494	42.5
RESUMER2 (C)	8.8	2712	18.8
Django (C)	11.2	n.a.	297.2

Table 3.2: Average runtimes for hypothesis search for the PTC dataset and max. hypothesis size 100.

of RESUMER2 and Django was not significant. The **PTC-v2** version deals with the deficiencies of the **PTC-v1** data representation. Here, every bond is again represented by two literals  $bond(at1, at2, bondName)$  and  $bond(at2, at1, bondName)$ . Furthermore, we added literals  $different(a, b)$  for all pairs of atom-representing constants  $a$  and  $b$  in examples. In this setting, the representation size of examples grew to thousands of atoms. In this case, RESUMER2 was significantly faster than Django, whose runtime exceeded time limit 5 hours. In the **PTC-v3** experiment we reduce the size of the representation of examples by inserting the  $different(a, b)$  literal only for atoms  $a, b$  that both have a bond with a common atom. In this last case, RESUMER2 was again significantly faster than Django (about one order of magnitude for the C version).

The results of the PTC dataset experiments, summarized in Table 3.2 refer to the setting with sizes of hypotheses bounded by 100. It is important to note that despite the fact that relatively long hypotheses are allowed, the actual lengths of these hypotheses are governed by the search algorithm. This explains why the speed-up achieved by RESUMER2 is not as high as in the *random walk* setting used in [26]. On the other hand, the speed-up is still very significant (orders of magnitude for the more complex settings).

While results achieved in Section 3.2.2.1 on generated graph data indicated that RESUMER2's performance superiority over Django grows with increasing size of examples,

the PTC domain experiments clearly confirm this observation.

### 3.3 RECOVER: A Restarted $\theta$ -subsumption Estimator

One line of research in developing efficient  $\theta$ -subsumption algorithms focuses on complete algorithms such as Django or RESUMER presented in the previous section. Another line of research concentrates on approximation of  $\theta$ -subsumption. Sebag et al. [41] presented a tractable approximation of  $\theta$ -subsumption called stochastic matching. Arias et al. [1] developed a randomized table-based approximation method. The algorithm, termed RECOVER, introduced in this section is an example of the latter class of algorithms. It utilizes randomized restarted strategies to estimate coverage of a hypothesis with respect to a set of examples.

Algorithm RECOVER cannot be used in propositionalization for computing extensions because it returns only a single number - estimate of coverage. On the other hand, whenever there is e.g. a constraint on minimum frequency of generated features, it can be immediately used to estimate this frequency. Furthermore, we develop a modification of RECOVER, which may be used for estimating extensions of features. This modification is more suitable for propositionalization than RECOVER.

#### 3.3.1 Derivation of Coverage Estimator

Let us first explain the intuition underlying RECOVER. Recall that the times taken by RESUMER on the unsatisfiable instances were in some of our experiments much higher than on the satisfiable ones. One of the reasons for this behaviour is the ‘iterative’ character of RESUMER2; unsatisfiability can only be shown after  $n$  restarts making  $R(n)$  sufficiently high. It would be thus advantageous if we could avoid proving non-existence of a solution. While this does not seem possible in general, we will develop a heuristic estimation method that will at least try to achieve this.

We are given a clause  $C$ , and example set  $E$  and we would like to estimate the coverage  $cov(C, E) = |\{e \in E \mid C \preceq_{\theta} e\}|$ . Let us run Algorithm 4 on  $C$  and  $e$ , successively for all  $e \in E$ . For each  $e$ , we however stop the algorithm if no decision has been made in  $R$  steps. Let  $\mathcal{E} \subseteq E$  be the subset of examples proven to be subsumed by  $C$  in this

experiment. Denote  $s_1 = |\mathcal{E}|$ . We now remove all examples in  $\mathcal{E}$  from  $E$  and repeat this experiment, obtaining analogical number  $s_2$ . Further such iterations generate numbers  $s_3, s_4$ , etc. Clearly, for the desired value  $\text{cov}(C, E)$ , we have that  $\text{cov}(C, E) = \lim_{j \rightarrow \infty} S_j$  where  $S_j = \sum_{i=1}^j s_i$ .

The main idea of RECOVER is that the limit of  $S_j$  for  $j \rightarrow \infty$  should be estimated by extrapolating the series from its first few elements  $S_1, S_2, \dots$ . Thus we achieve a coverage estimate without excessive effort to prove non-existence of a solution for the examples not subsumed by  $C$ . In order to precisely derive an estimation algorithm following the above idea, we first need to make the following assumption<sup>5</sup>.

**Assumption 3.1:** Given a clause  $C$  and a set of examples  $E$ , the probability  $p > 0$  that Algorithm 4 finds a solution (i.e. returns YES as its answer) before it explores more than  $R$  nodes of the search tree, is the same for all  $e \in E$  such that  $C$  subsumes  $e$ .  $\blacktriangleright$

In other words, we assume that properties of particular examples such as their size are not dramatically different. The assumption will be empirically validated in the next section. We assume a given clause  $C$  and we fix a constant cutoff value  $R$ . In the first step, for each  $e \in E$  we run *SubsumptionCheck*( $P, e$ ) (Algorithm 4), stopping it as soon as the number of searched nodes has reached  $R$ . Then, after  $|E|$  restarts (each time with a different  $e \in E$ ), we can derive the probability that the algorithm has produced exactly  $m_1$  ‘YES’ responses in this first step. In particular, this probability  $P(m_1)$  is

$$P(m_1) = \binom{A}{m_1} p^{m_1} (1-p)^{A-m_1} \quad (3.3)$$

where  $A = |\{e \in E | C \preceq_\theta e\}|$ . In the next step, all  $m_1$  examples shown to be subsumed in the first step are removed from  $E$  and the procedure is repeated with the remaining examples. In general, we can derive the probability that exactly  $m_i$  YES answers are generated in the  $i$ -th step. Thus for  $i = 2$ , we obtain

$$P(m_2 | m_1) = \binom{A - m_1}{m_2} p^{m_2} (1-p)^{A - m_1 - m_2} \quad (3.4)$$

and similarly for an arbitrary  $i \geq 1$ , we have

$$P(m_i | m_{i-1}, \dots, m_1) = \binom{A - \sum_{j=1}^{i-1} m_j}{m_i} p^{m_i} (1-p)^{A - \sum_{j=1}^i m_j} \quad (3.5)$$

---

<sup>5</sup>Naturally, the derived estimator will be correct only for situations where this assumption holds, but this does not necessarily mean that it will not work in practice for other situations.

---

**Algorithm 9** *ReCovEr*( $C, E, R, M, \Delta$ ): Algorithm for coverage estimation
 

---

```

1: Input: Clause  $C$  and set of examples  $E$ , Integers  $R$  ('cutoff'),  $M, \Delta$ ;

2:  $tries \leftarrow 0$ 
3:  $Unknown \leftarrow Examples$ 
4:  $CoveredInIthTry \leftarrow []$ 
5: repeat
6:    $tries \leftarrow tries + 1$ 
7:    $CoveredInThisTry \leftarrow 0$ 
8:   for  $\forall E \in Unknown$  do
9:      $Answer \leftarrow \text{Run } SubsumptionCheck(C, E)$  with number of searched nodes limited to  $R$ 
10:    if  $Answer = PositiveMatching$  then
11:       $CoveredInThisTry \leftarrow CoveredInThisTry + 1$ 
12:       $Unknown \leftarrow Unknown \setminus E$ 
13:    end if
14:  end for
15:   $CoveredInIthTry[tries] \leftarrow CoveredInThisTry$ 
16: until TerminationCondition
17: return  $LikelihoodEstimate(tries)$ 

```

---

The probability of a sequence  $(m_1, \dots, m_k)$ , where  $m_i$  is the number of examples for which YES was produced in the  $i$ -th step, is given by

$$P(m_1, \dots, m_k) = \prod_{i=1}^k P(m_i | m_{i-1}, \dots, m_1) \quad (3.6)$$

Substituting for  $P(m_i | m_{i-1}, \dots, m_1)$  from Eq. 3.5 and taking the logarithm Eq. 3.6 results in

$$\ln(P(m_1, \dots, m_k)) = \sum_{i=1}^k (\alpha + m_i \ln p + \beta) \quad (3.7)$$

where

$$\alpha = \ln \left( \frac{A - \sum_{j=1}^{i-1} m_j}{m_i} \right)$$

and

$$\beta = \left( A - \sum_{j=1}^i m_j \right) \ln(1 - p)$$

To find the parameters  $A$  and  $p$  for which  $P(m_1, \dots, m_k)$  is maximized, we take the partial derivative of Eq. 3.7 with respect to  $p$  and then find its roots, yielding

$$p = \frac{\sum_{i=1}^k m_i}{\sum_{i=1}^k m_i + \sum_{i=1}^k \left( A - \sum_{j=1}^i m_j \right)} \quad (3.8)$$

Finding the global maximum of  $P(m_1, \dots, m_k)$ , if it exists, from Eq. 3.6 on the set

$$D = \{(A, p) | A \in \{1, 2, \dots, |E|\} \wedge p \in (0; 1)\} \quad (3.9)$$

is now straightforward, since using (3.8) we can find the maximum on every line

$$L_i = \{(i, p) | p \in (0; 1)\} \quad (3.10)$$

If there is a maximum on line  $L_i$ , it is located at the value of  $p$  given by (3.8). If there is not a maximum on  $L_i$ , we need to inspect the limits for  $p \rightarrow 0+$  and  $p \rightarrow 1-$ . We can exclude  $p \rightarrow 0+$  because if the algorithm finds no solutions in first  $k$  restarts, it cannot decide whether no example is covered by the hypothesis or whether the probability of finding a solution is just too close to zero (and if some examples were covered,  $p$  could not be zero). Therefore if no example is covered in first  $k$  restarts, the algorithm chooses to answer that there is no example covered. On the other hand, if some examples are covered, it suffices to evaluate (3.6) at the points given by (3.8) and the limit for  $p \rightarrow 1$  on  $L_i$  for every  $i$  ( $1 \leq i \leq |E|$ ). The estimate of  $A$  then equals the index  $i$  of the  $L_i$  for which a maximum value was obtained (in any of the two points).

The described estimator is used in RECOVER (Algorithm 9). The question how to choose  $k$ , i.e. how long a sequence  $(m_1, \dots, m_k)$  should be generated as the input to the estimator, is tackled iteratively: the sequence is being extended until a termination condition is met. We have considered several termination conditions, of which two turned out to be quite useful. The first termination condition stops generating the sequence when two subsequent estimates differ by less than some  $\Delta_e$ , specified as a parameter. The second termination condition stops generating the sequence when estimate and number of examples already shown to be covered by the clause differ again by less than some  $\Delta_c$ , which ensures that the estimator will never overestimate the actual coverage by more

than  $\Delta_c$ . A minimum length  $M$  of the sequence is however imposed in both previous cases, to avoid premature estimates coinciding by chance.

Another degree of freedom in Algorithm 9 is the cutoff  $R$ , which may significantly affect the performance of the restarted algorithm. A heuristic method suggests itself that first tries to find a suitable cutoff. Unlike Algorithm 9 it starts with a base cutoff value, and then doubles it after every single restart. If at any restart Algorithm 4 with cutoff set to  $R$  does not cover more examples than the same algorithm at previous restart with cutoff set to  $\frac{R}{2}$ , then we can accept cutoff  $\frac{R}{2}$ .

An obvious drawback of algorithm RECOVER is that a fixed cutoff strategy must be used. This drawback is partially overcome by the heuristic method for selecting a good cutoff described above. Nevertheless, if we restrict ourselves to estimating extensions, which is in fact of higher importance for propositionalization than estimating coverage, we can implement a modification of RECOVER, which can be used with increasing cutoff sequences. This is good news especially because increasing cutoff sequences are much more robust than those with a fixed cutoff.

The restarted extension estimator is, in fact, just a slight modification of RECOVER algorithm. It is RECOVER with stopping condition, which stops restarts when the estimated coverage is equal to number of already covered examples, and with an increasing cutoff sequence. We do not provide full theoretical justification for this algorithm. We will only show an indirect evidence based on Theorem 3.2 and on simulations in Matlab. Theorem 3.2 says basically that if the estimated probability  $\hat{p}$ , that in any restart subsumption is decided for a given example, is smaller than the average probability  $\bar{p} = \frac{1}{k} \sum_{i=1}^k p_i$ , where  $p_i$  is probability that subsumption is decided in  $i$ -th restart (with cutoff  $c_i$ ), then the estimate will be equally good as if we knew exact values of  $p_i$ .

Of course, the question remains whether really  $\hat{p} \leq \bar{p}$ . To answer this question we performed experiments with simulation of RECOVER in Matlab. We generated  $k$  random values uniformly distributed on  $(0, 0.1)$  and we sorted them so that they would form an increasing sequence  $(p_i)_{i=1}^k$ . Then we simulated RECOVER with probability of finding a solution in  $i$ -th restart equal to  $p_i$ . Then we computed estimate  $\hat{p}$  and we checked whether  $\hat{p} \leq \frac{1}{k} \sum_{i=1}^k p_i$ . Figure 3.5 displays results of this simulation. The dashed line corresponds to average fraction of examples covered in  $k$  restarts. The solid line corresponds to probability that  $\hat{p} \leq \frac{1}{k} \sum_{i=1}^k p_i$ . It seems encouraging that the assumption is valid, for example, in about 80% cases for  $k$  (number of restarts), which is otherwise sufficient to cover only about 50% of examples on average.

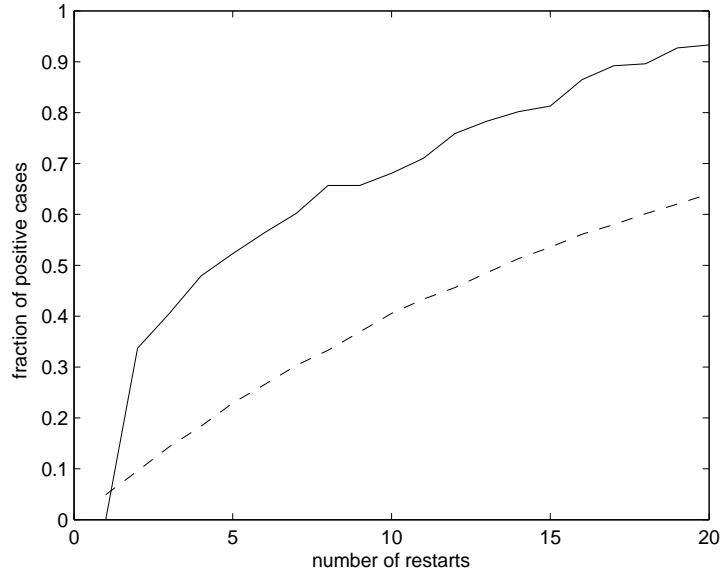


Figure 3.5: Simulation of RECOVER with increasing cutoff sequences

**Lemma 3.1 (Jensen's inequality):** *Let  $f$  be a real concave function defined on  $D_f \subseteq \mathbb{R}$ , let  $x_i \in D_f$  be real numbers and let  $a_i \in \mathbb{R}$  be weights. Then following holds*

$$f\left(\frac{\sum_i a_i \cdot x_i}{\sum_i a_i}\right) \geq \frac{\sum_i a_i \cdot f(x_i)}{\sum_i a_i}$$

**Proof:** Proof can be found in [37]. □

**Theorem 3.2:** *Let  $(p_i)_{i=1}^{\infty}$  be unknown probabilities that a solution is found for an example in a restart with cutoff  $c_i$ . Let us assume that  $p_i$  is the same for all examples (cf. Assumption 3.1). Further, let an instance of the RECOVER algorithm be used with cutoff sequence  $(c_i)_{i=1}^{\infty}$  in the setting where new restarts are performed until the number of covered examples is equal to the estimated coverage. Let  $(m_i)_{i=1}^k$  be a sequence of counts of examples covered by this algorithm. Let  $e$ ,  $(\hat{p}$ , respectively) be a maximum-likelihood estimate of coverage (of probability of finding a solution in a single restart, respectively) given the sequence  $(m_i)_{i=1}^k$  under assumption of use of a fixed-cutoff strategy with cutoff  $c_1$ . If for the estimated probability  $\hat{p}$  it is true that  $\hat{p} \leq \frac{1}{k} \cdot \sum_{i=1}^k p_i$ , then  $e$  is also a maximum-likelihood estimate of coverage under assumption that the increasing cutoff sequence  $(c_i)_{i=1}^{\infty}$  is used and the probabilities  $(p_i)_{i=1}^{\infty}$  are known.*

**Proof:** Let  $L(A, p_1, \dots, p_k)$  be logarithm of probability that sequence  $(m_1, \dots, m_k)$  was generated by algorithm  $R$  with probabilities of finding a solution in



$i$ -th restart equal to  $p_i$ . For  $A = \sum_{i=1}^k m_i$ , we have

$$L\left(\sum_{i=1}^k m_i, p_1, \dots, p_k\right) = \sum_{i=1}^k \left[ \ln \binom{\sum_{j=i}^k m_j}{m_i} + m_i \ln p_i + \ln(1 - p_i) \sum_{j=i+1}^k m_j \right],$$

and for general  $A$ , we have

$$L(A, p_1, \dots, p_k) = \sum_{i=1}^k \left[ \ln \binom{A - \sum_{j=1}^{i-1} m_j}{m_i} + m_i \ln p_i + \ln(1 - p_i) \left( A - \sum_{j=1}^i m_j \right) \right].$$

The assumption that  $A = \sum_{i=1}^k m_i$  is a maximum likelihood estimate for  $p_i = \hat{p}$  gives us

$$L\left(\sum_{i=1}^k m_i, \hat{p}, \dots, \hat{p}\right) - L(A, \hat{p}, \dots, \hat{p}) = \sum_{i=1}^k \left( \alpha_i^{\hat{p}} + \beta_i^{\hat{p}} \right) \geq 0,$$

$$\alpha_i^{\hat{p}} = \ln \frac{\binom{\sum_{j=i}^k m_j}{m_i}}{\binom{A - \sum_{j=1}^{i-1} m_j}{m_i}}, \quad \beta_i^{\hat{p}} = \ln(1 - \hat{p}) \left( \sum_{j=1}^k m_j - A \right)$$

where  $\beta_i^{\hat{p}} \geq 0$  for  $A \geq \sum_{i=1}^k m_i$ . Note that it would not make sense to consider  $A < \sum_{i=1}^k m_i$  as this would mean estimating lower coverage than is the number of already covered examples. What we need to prove is that  $L(\sum_{i=1}^k m_i, p_1, \dots, p_k) - L(A, p_1, \dots, p_k) \geq 0$  if  $L(\sum_{i=1}^k m_i, \hat{p}, \dots, \hat{p}) - L(A, \hat{p}, \dots, \hat{p}) \geq 0$ , i.e. that

$$L\left(\sum_{i=1}^k m_i, p_1, \dots, p_k\right) - L(A, p_1, \dots, p_k) = \sum_{i=1}^k (\alpha_i + \beta_i) \geq 0,$$

$$\alpha_i = \ln \frac{\binom{\sum_{j=i}^k m_j}{m_i}}{\binom{A - \sum_{j=1}^{i-1} m_j}{m_i}}, \quad \beta_i = \ln(1 - p_i) \left( \sum_{j=1}^k m_j - A \right)$$

Since  $\alpha_i$  does not depend on  $p_i$ , it will suffice to show that  $\sum_{i=1}^k \beta_i^{\hat{p}} \leq \sum_{i=1}^k \beta_i$  for  $\hat{p} \leq \bar{p}$ . The next derivation uses Jensen's inequality.

$$\sum_{i=1}^k \ln(1 - \hat{p}) \geq \sum_{i=1}^k \ln \left( 1 - \frac{1}{k} \sum_{j=1}^k p_j \right) \geq \sum_{i=1}^k \sum_{j=1}^k \frac{1}{k} \ln(1 - p_j) = \sum_{i=1}^k \ln(1 - p_i)$$

It follows (assuming  $\sum_{i=1}^k m_i \leq A$ , i.e. that  $\sum_{i=1}^k m_i - A \leq 0$ ) that

$$\sum_{i=1}^k \beta_i^{\hat{p}} = \sum_{i=1}^k \left( \sum_{j=1}^k m_j - A \right) \ln(1 - \hat{p}) \leq \sum_{i=1}^k \left( \sum_{j=1}^k m_j - A \right) \ln \left( 1 - \frac{1}{k} \sum_{j=1}^k p_j \right) \leq$$

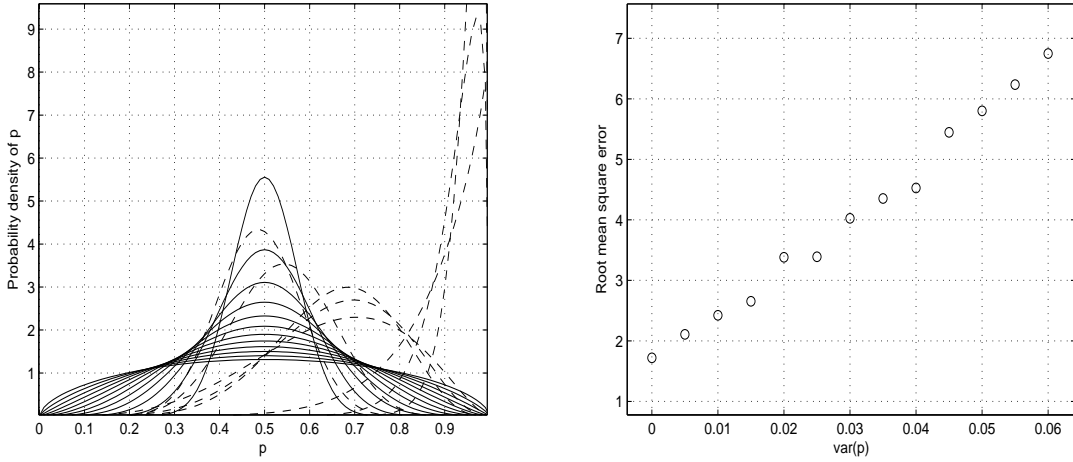


Figure 3.6: Sensitivity of RECOVER to violations of Assumption 3.1

$$\leq \sum_{i=1}^k \left( \sum_{j=1}^k m_j - A \right) \sum_{j=1}^k \frac{1}{k} \ln(1 - p_j) = \sum_{i=1}^k \left( \sum_{j=1}^k m_j - A \right) \ln(1 - p_i) = \sum_{i=1}^k \beta_i$$

This finishes the proof because  $0 \leq \sum_{i=1}^k (\alpha_i^{\hat{p}} + \beta_i^{\hat{p}}) \leq \sum_{i=1}^k (\alpha_i + \beta_i) = L(\sum_{i=1}^k m_i, p_1, \dots, p_k) - L(A, p_1, \dots, p_k)$ .  $\square$

The above theorem justifies (at least to some extent) the use of an increasing cutoff sequence in RECOVER in the setting where no extrapolation is performed and estimation is limited to deciding whether the current number of covered examples equals the overall number of covered examples. This *no extrapolation* setting is basically extension estimation and it will be called RECOVER-E in what follows.

### 3.3.2 Experiments

In this section, we first investigate the sensitivity of RECOVER to a violation of Assumption 3.1. Then we evaluate its performance and precision on random graph data and on real-world data from organic chemistry and from engineering. We compare performance of RECOVER with that of the state-of-the-art  $\theta$ -subsumption algorithm Django. We have chosen Django, and not any of the existing approximate algorithms, for comparisons because in [1] it has been shown to perform similarly well or even better than one of the approximate methods: the table-based incomplete heuristic method.

### 3.3.2.1 Sensitivity Analysis

Here we address Assumption 3.1. We first want to verify to what extent the assumption is satisfied, and subsequently, how much the deviations from the assumption influence RECOVER's precision. To this end we performed a series of experiments, in which we measured empirical distributions of probabilities  $p$  and we run simulations of RECOVER.

According to Assumption 3.1, probability  $p \in (0; 1]$  would be a constant. Now, we assume that  $p$  is a random variable with some distribution on  $[0; 1]$ , which we would like to estimate. A standard approach to this task is to parameterize a Beta distribution on  $[0; 1]$  from empirical data. To obtain the data, we experimented with the random graph data with parameters of generated hypotheses  $p = 0.35$ ,  $n = 10$ , parameters of generated examples  $p = 0.3$ ,  $n = 50$  and RECOVER's cutoff  $R = 75$ . This resulted in Beta distributions plotted in dashed lines in Fig. 3.6. For comparisons, we plotted also Beta distributions with mean  $\mu = 0.5$  and variance consecutively  $0, 0.005, \dots, 0.06$  (solid lines).

Then we investigated RECOVER's sensitivity to the modeled deviations. We assumed to have  $n = 100$  examples, of which 50 were covered by a clause  $C$ . Further, probabilities  $p_i$  that Algorithm 4 finds a solution for a covered example  $e_i$  in time less than  $R$  were sampled from the Beta distribution with given mean  $\mu = 0.5$  and variance consecutively  $0, 0.005, \dots, 0.06$  (corresponding to those displayed in Fig. 3.6). Then, we simulated RECOVER's estimation procedure on these data in Matlab. We used the stopping condition based on difference of estimate and lower bound, the parameters were  $M = 3$ ,  $\Delta = 1$ . The right panel of Fig. 3.6 displays the dependence of root mean square error on the variance of the beta distributions. It is encouraging to see that the root mean square error grows roughly *linearly* with growing variance in  $p$ 's distribution, indicating RECOVER's robustness towards this variance.

### 3.3.2.2 Experiments with Generated Graph Data

Figure 3.7 demonstrates the precision of RECOVER and RECOVER-E on the Erdos-Rényi graph data by showing 1000 pairs (estimated coverage, actual coverage). Hypotheses and examples were generated with  $p = 0.2$  for hypotheses and  $p = 0.3$  for examples. The graphs corresponding to hypotheses had 20 vertices, and the graphs corresponding to examples had 100 vertices. The left panel refers to estimates obtained by RECOVER enhanced by cutoff selection with base cutoff  $R = 100$  ( $\circ$ ) and by RECOVER-E with the same base cutoff  $R = 100$ , while the right panel refers to estimates obtained by the same

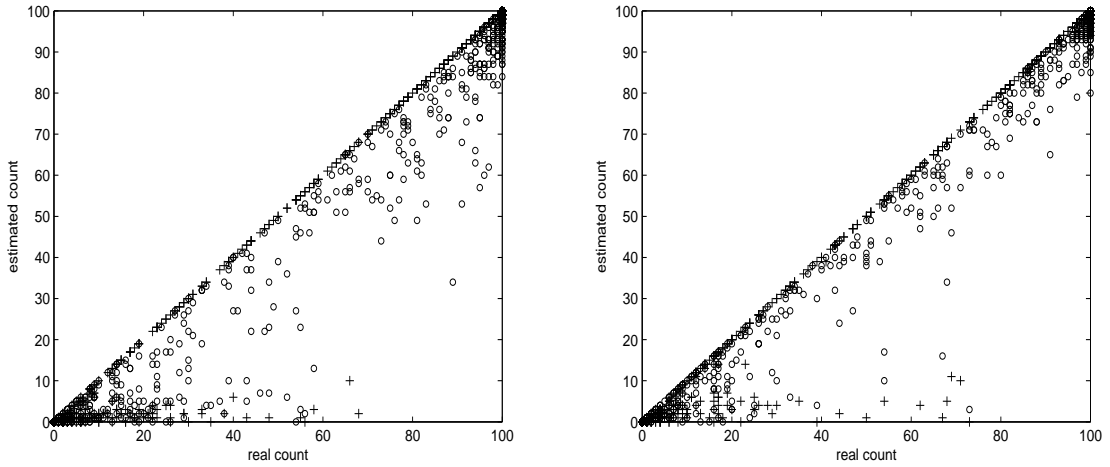


Figure 3.7: Precision of RECOVER (o) and RECOVER-E (+)

algorithms with base cutoff  $R = 200$ . A bias towards coverage under-estimation can be observed, as well as a positive effect of the higher base cutoff on estimation precision. The under-estimation bias is caused by the used stopping condition. Table 3.3 shows average runtimes of RECOVER, RECOVER-E and Django for these settings.

It is interesting to note that while root mean squared error of RECOVER-E was lower for base cutoff 100 ( $err = 6.4$ ) than root mean squared error of RECOVER ( $err = 8.7$ ), it was higher for the setting with base cutoff 200 ( $err = 6.4$  for RECOVER-E and  $err = 5.9$  for RECOVER for  $R = 200$ ). On the other hand, RECOVER-E estimated correct coverage (extension) in significantly larger fraction of cases (81% for  $R = 100$  and 81% for  $R = 200$ ) than RECOVER (38% for  $R = 100$  and 48% for  $R = 200$ ). This discrepancy may be explained by the fact that while RECOVER-E usually estimates correct coverage, sometimes, if it underestimates the correct value, it underestimates it more than RECOVER. This is mostly because the cutoff selection method used in RECOVER causes, for first few restarts, more rapid growth of cutoff than is the growth of the cutoff sequence in RECOVER-E for the base cutoff  $R = 200$  ( $2 \cdot 200 > 200 + 10 \cdot e^2$ ,  $4 \cdot 200 > 200 + 10 \cdot e^3$ ). Therefore RECOVER-E *does not always catch* as fast as RECOVER in situations where probability of finding a solution is very small for low values of cutoff. While this could definitely be fixed by using a more fine-tuned cutoff sequence, we do not consider it necessary to do this fine tuning here.

Whether the observed estimation variance is tolerable for the task of clause *ranking* usual in inductive logic programming is the subject of the experiments in the next section.

Algorithm	Avg. Time [s]
RECOVER, $R = 100$	13.5
RECOVER, $R = 200$	14.7
RECOVER-E, $R = \lfloor 100 + 10 \cdot e^r \rfloor$	18.2
RECOVER-E, $R = \lfloor 200 + 10 \cdot e^r \rfloor$	19.4
Django	1230

Table 3.3: Avg. estimation runtimes for the configurations from Fig. 3.7.

### 3.3.2.3 Experiments with Real-World Data

In this section, we describe experiments with algorithm RECOVER in two real-world domains. We do not evaluate performance of RECOVER-E, since it is intended to be an algorithm suitable for propositionalization, while here we want to test the coverage estimation algorithm in hypothesis search. We use the C version of RECOVER described in [24].

In order to assess performance in conditions of a real-life learning setting, we decided not to generate clauses entirely randomly. Our intention was to simulate general principles of clause production in an inductive logic programming system, while avoiding an overfit to a specific clause search strategy (which would e.g. be a result of adhering to a specific heuristic function for selecting literals). Thus we developed a simple relational learner, which we use for further experiments with RECOVER. The learner (Algorithm 10) is a randomized variation of a specific-to-general beam search. It starts with the most specific clause  $\perp$  and at each search step, it generates at least  $n \cdot |Beam|$  new hypotheses by removing random subsets of literals from the hypotheses already present in *Beam*. The output of the algorithm is one best clause, which is why we assess its quality through precision and recall.

The first set of experiments, which we have conducted with Algorithm 10, deals with the Mutagenesis dataset [43]. This dataset consists of descriptions of 188 organic molecules, which are marked according to their mutagenicity. In our experiments, we used only the information about atom-bond relationships and about types of atoms. We did not consider numerical parameters such as *lumo* or *logp*. Our relational-logic representation of these molecules consisted of ternary literals for atomic bonds  $bond(at1, at2, bondType)$ , unary literals representing types of particular bonds and unary literals for atom types. As in our experiments with algorithm RESUMER in the PTC

---

**Algorithm 10** *Learner*( $\perp, p, BeamSize, Tries$ ): A Clause Learner
 

---

```

1: Input: Most specific clause  $\perp$ , Real numbers  $p$ , Integers  $BeamSize, MaxSearched$ 

2:  $Beam \leftarrow \{\perp\}$ 
3:  $BestClause \leftarrow \perp$ 
4: repeat
5:    $Candidates \leftarrow Beam$ 
6:   for  $\forall h_i \in Beam$  do
7:     for  $i = 1 \dots BeamSize$  do
8:        $GenerateClause(h_i)$ 
9:        $C \leftarrow$  connected components of  $c$ 
10:      Evaluate each  $c_i \in C$ 
11:       $Candidates \leftarrow candidates \cup C$ 
12:    end for
13:  end for
14:  for  $\forall h \in Candidates$  such that  $h$  is estimated to be better than  $BestClause$  do
15:    if  $h$  is shown to be better than  $BestClause$  by a deterministic subsumption
        algorithm then
16:       $BestClause \leftarrow h$ 
17:    end if
18:  end for
19:  Choose  $BeamSize$  best hypotheses from  $Candidates$  and add them to  $Beam$ 
20:   $Explored \leftarrow Explored + 1$ 
21: until  $Beam = \{\}$  or  $Explored = Tries$ 

```

---

domain described in Section 3.2.2.2<sup>6</sup>, we have considered three variants of relational logic description of the molecules, with growing complexity (size of examples). The first version **Muta-v1** uses a naive representation. Here, each molecular bond is represented by a single literal  $bond(at1, at2, bondType)$ , thus imposing a bond orientation (atom order) chosen at random. The second source of imprecision of this representation is that two variables in a clause may represent the same (chemical) atom, which does not make intuitive sense. The second version **Muta-v2** deals with the first source of imprecision, as it represents every atomic bond with a pair of literals  $bond(at1, at2, bondType)$  and

---

<sup>6</sup>The settings described in this section are slightly different from those shown in Section 3.2.2.2.

$bond(at2, at1, bondType)$ . The third version **Muta-v3** solves the second source of imprecision by adding literals  $different(a, b)$  for all pairs of atom-representing constants  $a, b$ .

The second set of experiments pertains to class-labeled CAD data (product structures) described in [51], consisting of 96 CAD examples each containing several hundreds of first-order literals.

The main observation provided by the experiments is that RECOVER becomes quickly superior to Django as the example size grows, whereas the two algorithms do not significantly differ in terms of the training-set<sup>7</sup> accuracy of the discovered clauses. It is interesting to note that Django’s poor runtime performance on the learning tasks with large examples (**CAD** data and **Muta-v2**) was often due to occasional subsumption cases. Clearly, this is a manifestation of heavy tails present in Django’s runtime distribution. Unlike Django, RECOVER was exhibiting steady performance.

Dataset	RECOVER [s]	Django [s]
Muta-v1	42	29
Muta-v2	513	1627
Muta-v3	1695	>5h
CAD	121	>2h

Table 3.4: Average runtimes of the learner (Algorithm 10,  $p = 0.75$ ,  $Tries = 10$ ) for real-world datasets.

Dataset	Avg. Precision	Avg. Recall
Muta-v1	0.84	0.61
Muta-v2	0.81	0.65
Muta-v3	0.83	0.84
CAD	0.92	0.7

Table 3.5: Quality of learned hypotheses for RECOVER

---

<sup>7</sup>As this paper is not concerned with improving generalization performance, we did not measure accuracies on hold-out test sets.

Dataset	Avg. Precision	Avg. Recall
Muta-v1	0.86	0.6
Muta-v2	0.82	0.65
Muta-v3	n.a.	n.a.
CAD	n.a.	n.a.

Table 3.6: Quality of learned hypotheses for Django

### 3.4 Discussion of Experiments with RESUMER and RECOVER

The algorithms presented in this chapter were designed to exploit special properties of  $\theta$ -subsumption. Experiments on both artificially generated and real-life data showed that both RESUMER2 and RECOVER are competitive with state-of-the-art  $\theta$ -subsumption algorithm Django, which is considered the fastest  $\theta$ -subsumption engine.

The experiments showed wide ranges of speed-up achieved by our novel algorithm RESUMER2 in comparison to Django. On real-life data, the speed-up ranged from actually a slow-down of tens of percent to a speed-up of several orders of magnitude for large examples. While the results are encouraging, we expect that there is also a vast number of datasets where Django could outperform our algorithms. The reason is that Django uses a CSP encoding of  $\theta$ -subsumption, which is dual to the encoding used by our algorithms. Order parameters, which determine worst-case time complexity of the algorithms, are different for the two encodings [1]. Therefore empirical performance could also differ significantly depending on a given dataset. Based on experiments, which showed that RESUMER2 outperformed Django especially in the YES region, we conjecture that RESUMER2 or some similar algorithm could be beneficial in a relational learner, which would construct hypotheses in a general-to-specific manner.

Evaluation of our second algorithm, RECOVER, is a bit harder because we need to take into account both runtime and accuracy of estimation. Nevertheless, the experiments, which we have performed, indicate that RECOVER or some similar algorithm could be beneficial in the situations when very complex hypotheses and examples are encountered. It is question whether existing publicly-available datasets are challenging enough from this point of view. For example, in order to test the ability of RECOVER to cope with complex hypotheses from a real-life dataset, we had to devise a (naive) specific-



to-general-style algorithm because such an algorithm typically produces long hypotheses already in early stages of its run.

An interesting property of RECOVER, which we have not tested, is that the core algorithm need not be complete but that some incomplete algorithm could be used as well. For example, it would be interesting to see how RECOVER would work with the randomized table-based method described in [1].

# Chapter 4

## Two Propositionalization Algorithms

As we have already explained in Section 2.4, propositionalization aims at converting structured descriptions of examples<sup>1</sup> into attribute-value descriptions which can be then processed by most established machine learning algorithms. In this chapter, we work in the *learning from interpretations setting* [35]. We assume that examples are represented as first-order logic interpretations, features are first-order conjunctions and some Horn background theory  $B$  (possibly empty) is available. Feature  $F$  acquires value *true* for example  $I$  (is *covered* by the example) if  $m(B) \cup I \models F$  where  $m(B)$  is the minimal model of  $B$ , otherwise it has the *false* value. With slight abuse of notation, we will write this relation simply as  $B \wedge I \models F$ .

Current systems (e.g. RSD [44], SINUS [21]) construct conjunctions iteratively in a level-wise manner by adding one literal at time. Monotonicity of frequency (if  $F$  is not frequent then  $F \wedge l$  is not frequent for any literal  $l$ ) is exploited for pruning by numerous algorithms, e.g. by a well-known frequent query discovery system WARMR [9]. Unfortunately, *relevancy* and *irreducibility* which are another properties of interest in propositionalization are in general not monotone in this level-wise approach. In this chapter we introduce two novel propositionalization algorithms for construction of *hierarchical* features, which construct feature sets by composition of smaller conjunctions of literals ('building blocks'). The two algorithms exploit a form of monotonicity of *relevancy* and *irreducibility*, which is preserved in the block-wise approach. This enables them to vastly outperform existing state-of-the-art systems by orders of magnitude in runtime even in situations where these systems work also only with *hierarchical* features.

This chapter is organized as follows. First, we study properties of hierarchical features

---

<sup>1</sup>The structured representation of examples may have different forms depending on used learning setting - e.g. learning from interpretations as used here or learning from entailment.

regarding extension computation, reducibility and relevancy. The obtained theoretical results are then used to develop the two propositionalization algorithms. The first of these algorithms (RELF) exploits mostly the properties of relevancy and reducibility of features, while the second algorithm (HiFi) focuses more on restricting the feature space syntactically.

## 4.1 Analysis of Hierarchical Features

### 4.1.1 Hierarchical Features

The set of literals in a conjunction  $C$  is written as  $lits(C)$ ,  $|C| = |lits(C)|$  is size of  $C$ . A set of conjunctions is said to be *standardized-apart* if no two conjunctions in the set share a variable. Given a set  $A$  of atoms, we denote  $Args(A) = \{(a, n) | a \in A, 1 \leq n \leq \text{arity}(a)\}$ , i.e.  $Args(A)$  is the set of all argument places in  $A$ . We will assume that no conjunction contains two equal literals, i.e.  $p(X) \wedge p(Y)$  will be allowed, but  $p(a) \wedge p(a)$  will not. For an atom  $a$ ,  $arg_i(a)$  is its  $i$ -th argument. The admissible features syntax will be constrained by means of *hierarchical templates*, which will be a slight modification<sup>2</sup> of templates described in Section 2.4.1.

**Definition 4.1 (Hierarchical Template):** A pre-template is a pair  $(\gamma, \mu)$  where  $\gamma$  is a finite set of ground atoms and  $\mu \subseteq Args(\gamma)$ . Elements of  $\mu$  ( $Args(\gamma) \setminus \mu$ ) are called *inputs* (*outputs*) in  $\gamma$ . A pre-template  $(\gamma, \mu)$  is a *hierarchical template* if every atom in  $\gamma$  has at most one input argument and there is a partial irreflexive order  $\prec$  on constants in  $\gamma$  such that  $c \prec c'$  whenever  $c$  appears as an input and  $c'$  as an output in some  $l \in \gamma$ . ►

A hierarchical template  $\tau = (\gamma, \mu)$  is conveniently shown by writing  $\gamma$  with input (output) arguments marked with the sign  $+$  ( $-$ ), such as  $\tau \approx \{hasCar(-c), hasLoad(+c, -l), box(+l), large(+l), triangle(+l)\}$ .

**Definition 4.2 (Hierarchical Feature):** Given a hierarchical template  $\tau = (\gamma, \mu)$ , a  $\tau$ -pre-feature  $F$  is a finite conjunction of literals containing no constants or functions, such that  $lits(F\theta) \subseteq \gamma$  for some substitution  $\theta$ . The occurrence of variable in the  $i$ -th argument of literal  $l$  in  $F$  is an *input* (*output*) occurrence in  $F$  if the  $i$ -th argument of  $l\theta$  is (is not) in  $\mu$ . A variable is *neutral* in  $F$  if it has [i] at least one input occurrence in  $F$ , and

<sup>2</sup>This modified definition is not work of the author of this thesis but of author's advisor.

[ii] exactly one output occurrence in  $F$ . A variable is *pos* (*neg*) in  $F$  if it complies only with [i] ([ii]). A  $\tau$ -pre-feature is a hierarchical  $\tau$ -feature  $F$  if all its variables are neutral in  $F$ ; it is a *pos* (*neg*)  $\tau$ -feature  $F$  if it has exactly one *pos* (*neg*) variable, denoted  $\mathbf{p}(F)$  ( $\mathbf{n}(F)$ ), and the remaining variables are neutral; it is a *pos-neg*  $\tau$ -feature if it has exactly one *pos* variable and exactly one *neg* variable and the remaining variables are neutral. ►

For example, using the  $\tau$  defined under Def. 4.1 the following conjunction is a hierarchical  $\tau$ -feature

$$\begin{aligned} & hasCar(C) \wedge hasLoad(C, L1) \wedge hasLoad(C, L2) \wedge \\ & \wedge box(L1) \wedge large(L1) \wedge triangle(L2) \end{aligned}$$

$hasCar(C)$  is a *neg*  $\tau$ -feature,  $hasLoad(C, L1) \wedge box(L1)$  is a *pos*  $\tau$ -feature and  $hasLoad(C, L1) \wedge box(L1) \wedge large(L1) \wedge triangle(L2)$  is a *pos-neg*  $\tau$ -feature. It is important to note that the only *pos* (*neg*) variable  $\mathbf{p}(F^+)$  ( $\mathbf{n}(F^-)$ ) in a *pos* (*neg*) feature is uniquely given despite the fact that inputs and outputs are not given uniquely in general.

Wherever we shall deal with a single fixed template  $\tau$ , we will simply speak of (hierarchical, *pos*, *neg*, *pos-neg*) features instead of (hierarchical, *pos*, *neg*, *pos-neg*)  $\tau$ -features. Given the assumed partial order in templates, a hierarchical feature  $F$  corresponds to a tree graph  $T_F$ , here called the  $F$ -tree, with vertices  $v_i$  corresponding to literals  $l_i$ . There is an edge between  $v_i$  and  $v_j$  if there is a variable which has an output occurrence in  $l_i$  and an input occurrence in  $l_j$ . We say that a (hierarchical, *pos*, *neg*) feature has depth  $d$  if the corresponding  $F$ -tree has depth  $d$ . Analogously, we say that a literal  $l$  is in depth  $d$  in (hierarchical, *pos*, *neg*) feature if it is in depth  $d$  in the corresponding  $F$ -tree (if  $d = 0$ , we call  $l$  *root* of  $F$ ).

**Definition 4.3 (Graft):** Let  $F^-$  be a *neg* (*pos-neg*) feature and  $\phi^+ = \{F_i^+\}$  be a standardized-apart (possibly with exception  $\mathbf{n}(F^-) = \mathbf{p}(F_i^+)$ ) set of *pos* features. We define the *graft*  $F^- \oplus_{\mathbf{n}(F^-)} \phi^+ = F^- \wedge_i F_i \theta_i$ , where each substitution  $\theta_i = \{\mathbf{p}(F_i^+) / \mathbf{n}(F^-)\}$ . A *pos* feature  $F^+$  is said to be contained in hierarchical (*pos*) feature  $F$  if and only if  $F^+ \subset F$  and  $F \setminus F^+$  is a *neg* (*pos-neg*) feature. ►

In what follows, the variable in the subscript of the graft operator will be uniquely determined by context. Hence we will mostly drop the subscript for simplicity.

**Example 4.1 (Running Example):** Let us have a set of two positive examples  $E^+$  and a set of two negative examples  $E^-$

$$E^+ = \{\{hasCar(c1), hasLoad(c1, l1), circ(l1), box(l1), hasLoad(c1, l2), tri(l2)\},$$

$$\begin{aligned} & \{hasCar(c2), hasLoad(c2, l3), box(l3), tri(l3)\} \\ E^+ = & \{\{hasCar(c3), hasLoad(c3, l4), box(l4), circ(l4)\}, \\ & \{hasCar(c4), hasLoad(c4, l5), tri(l5), circ(l5)\}\} \end{aligned}$$

We define a very simple hierarchical template  $\tau \approx \{hasCar(-c), hasLoad(+c, -l), box(+l), tri(+l), circ(+l)\}$  to constrain the features for this data. Although there is an infinite number of features correct w.r.t.  $\tau$ , there are only finitely many features, which are not *H-reducible*, as we will see in the next section.  $\triangle$

### 4.1.2 Irreducibility

To define the reducibility property of conjunctive features, we will use the notion of  $\theta$ -subsumption.

**Definition 4.4 (Reducibility):** If  $D \preceq_{\theta} C$ , we call  $C$  and  $D$   $\theta$ -equivalent (written  $C \approx_{\theta} D$ ). We say that  $C$  is *reducible* if there exists a clause  $C'$  such that  $C \approx_{\theta} C'$  and  $|C| > |C'|$ . A clause  $C'$  is said to be a *reduction* of  $C$  if  $C \approx_{\theta} C'$  and  $C'$  is not reducible.  $\blacktriangleright$

An example of a  $\theta$ -reducible conjunction of literals is  $C = hasCar(C) \wedge hasLoad(C, L1) \wedge hasLoad(C, L2) \wedge box(L1)$ . Let  $D = hasCar(C) \wedge hasLoad(C, L1) \wedge box(L1)$ , then  $C\theta \subseteq D$ , where  $\theta = \{L2/L1\}$ , and  $|D| < |C|$ .

In this thesis we cannot rely directly on the established notion of reducibility as defined above. This is because a reduction of a hierarchical  $\tau$ -feature may not be a  $\tau$ -feature itself. For example, for  $\tau \approx \{car(-c_1), hasLoad(+c_1, -l), hasLoad(-c_2, +l), hasCar(+c_2)\}$ , the conjunction  $car(C_1) \wedge hasLoad(C_1, L_1) \wedge hasLoad(C_2, L_1) \wedge car(C_2)$  is a correct  $\tau$ -feature but its reduction  $hasCar(C_1) \wedge hasLoad(C_1, L_1)$  is not.

The fact that reduction does not preserve correctness of feature syntax may represent a problem because, to avoid redundancy, we would like to work only with reduced features. The next definition introduces *H-reduction*, which has the property that *H-reduction* of a hierarchical  $\tau$ -feature is always a hierarchical  $\tau$ -feature. While there is an infinite number of hierarchical  $\tau$ -features for a sufficiently rich template  $\tau$ , there is always only a finite number of non-*H-reducible* ones.

**Definition 4.5 (H-reduction):** We say that hierarchical (pos)  $\tau$ -feature  $f$  *H-subsumes* hierarchical (pos)  $\tau$ -feature  $g$  (written  $f \preceq_H g$ ) if and only if there is a substitution (called *H-substitution*)  $\theta$  such that  $f\theta \subseteq g$  and for every literal  $l \in lits(f)$  it holds

$depth_f(l) = depth_g(l\theta)$  for some correct assignment of inputs and outputs of  $f$  and  $g$ . If further  $g \preceq_H f$ , we call  $f$  and  $g$   $H$ -equivalent (written  $f \approx_H g$ ). We say that hierarchical (pos)  $\tau$ -feature  $f$  is  $H$ -reducible if there is a hierarchical (pos)  $\tau$ -feature  $f'$  such that  $f \approx_H f'$  and  $|f| > |f'|$ . Hierarchical feature  $f'$  is said to be an  $H$ -reduction of  $f$  if  $f \approx_H f'$  and  $f$  is not  $H$ -reducible.  $\blacktriangleright$

In the proof of Theorem 4.1 we will speak interchangeably about substitution  $\theta$  from variables to terms and about the induced substitution from literals to literals.

**Theorem 4.1:** *Let  $F^+$  be a pos feature and let  $F^-$  be a neg feature. Then following holds: (i)  $F^+$  is  $H$ -reducible if and only if  $F^+$  contains pos features  $F_1^+, F_2^+$  such that  $F_1^+ \neq F_2^+$ ,  $\mathbf{p}(F_1^+) = \mathbf{p}(F_2^+)$  and  $F_1^+ \preceq_H F_2^+$ . (ii) If  $F^+$  is  $H$ -reducible, then  $F^- \oplus F^+$  is also  $H$ -reducible. (iii) Whether a hierarchical (pos) feature  $F$  is  $H$ -reducible can be computed in polynomial time (in  $|F|$ ).*

**Proof:** In this proof we will use the following observation. Let  $A, B$  be pos features and let  $\theta$  be a  $H$ -substitution such that  $A\theta \subseteq B$ . Observe that if  $l \in B \setminus A\theta$ , then also  $l' \in B \setminus A\theta$  for any literal  $l'$  contained in pos feature  $F_l^+ \subseteq B$ , where  $F_l^+$  has  $l$  as its root. (i  $\Rightarrow$ ) Let  $F_r^+$  be  $H$ -reduction of  $F^+$  and let  $\theta_1, \theta_2$  be  $H$ -substitutions such that  $F_r^+\theta_1 \subseteq F^+$  and  $F^+\theta_2 \subseteq F_r^+$ . Substitution  $\theta_3 = \theta_2\theta_1$  is a mapping  $\theta_3 : lits(F^+) \rightarrow lits(F^+)$ . Since  $F^+\theta_2 \subseteq F_r^+$ ,  $|F^+\theta_2| \leq |F_r^+|$  and consequently  $|F^+\theta_3| \leq |F_r^+| < |F^+|$ , because applying a substitution to a feature cannot increase its size. Therefore there is a literal  $l \in F^+ \setminus F^+\theta_3$  and, as we have observed, also a whole pos feature  $F_1^+ \subseteq F^+ \setminus F^+\theta_3$ . Thus, there is a pos feature  $F_2^+$  ( $F_2^+ \neq F_1^+$ ) such that  $F_1^+\theta_3 \subseteq F_2^+$ . It remains to show that for some such  $F_1^+, F_2^+$ ,  $\mathbf{p}(F_1^+) = \mathbf{p}(F_2^+)$ . If  $\mathbf{p}(F_1^+) \neq \mathbf{p}(F_2^+)$ , then there must be pos features  $F_1^{+'}, F_2^{+'}$  such that  $F_1^+ \subset F_1^{+'}$ ,  $F_2^+ \subset F_2^{+'}$  and  $F_1^{+'}\theta_3 \subseteq F_2^{+'}$ . For such  $F_1^{+'}, F_2^{+'}$  with maximum size,  $\mathbf{p}(F_1^+) = \mathbf{p}(F_2^+)$ . (i  $\Leftarrow$ ) Let  $\theta$  be a  $H$ -substitution such that  $F^+ \setminus F^+\theta = F_1^+$ , then  $F^+\theta \approx_H F^+$  and  $|F^+\theta| = |F^+| - |F_1^+| < |F^+|$ . (ii) This follows from (i). (iii) An approach, which tests whether  $F_1^+ \preceq_H F_2^+$  for all pairs of pos features  $F_1^+, F_2^+$  contained in  $F^+$  with equal pos variables, runs in polynomial time in  $|F|$  (cf. Algorithm 11).  $\square$

The second property of  $H$ -reduction stated in Theorem 4.1 allows us to filter  $H$ -reducible pos features during propositionalization process.

**Example 4.2 (Running Example):** Let us return to our running example from Section 4.1.1. As we have already mentioned, there is an infinite number of hierarchical features, but only a finite number of non- $H$ -reducible ones. An example of an  $H$ -reducible

**Algorithm 11**  $\text{dom}_{I,B}(S)$ 

- 
- 1: **Input:** Pos feature  $F^+$ , Interpretation  $I$ , Background theory  $B$ ;
  - 2:  $\text{litsDom} \leftarrow \{l \mid \text{pred}(l) = \text{pred}(\text{root}(F^+)) \wedge ((I \wedge B) \models l)\}$
  - 3: **for**  $\forall$  output variables  $\text{out}_i$  of  $F^+$ 's root **do**
  - 4:    $\text{argDom}_i \leftarrow \bigcap_{c \in \text{children}_{\text{out}_i}(l)} \text{dom}_I(c)$
  - 5:    $\text{litsDom} \leftarrow \text{litsDom} \cap \{l \mid \text{arg}_i(l) \in \text{argDom}_i\}$
  - 6: **end for**
  - 7: **return**  $\{t \mid t \text{ is term at input of } l \wedge l \in \text{litsDom}\}$
- 

hierarchical feature for template  $\tau$  is

$$F_{\text{reducible}} = \text{hasCar}(C) \wedge \text{hasLoad}(C, L) \wedge \text{box}(L) \wedge \\ \wedge \text{hasLoad}(C, L2) \wedge \text{box}(L2) \wedge \text{circ}(L2) \wedge \text{tri}(L2).$$

This feature is indeed  $H$ -reducible as may be seen from the following fact

$$\text{hasLoad}(C, L) \wedge \text{box}(L) \preceq_H \\ \preceq_H \text{hasLoad}(C, L2) \wedge \text{box}(L2) \wedge \text{circ}(L2) \wedge \text{tri}(L2)$$

When all  $H$ -reducible hierarchical features are removed, there remain only 18 correct  $\tau$ -features for  $\tau$  from our running example.  $\triangle$

### 4.1.3 Relevancy

**Definition 4.6 (Domain):** Let  $I$  be an interpretation,  $B$  be a background theory,  $\tau$  be a hierarchical template and  $T$  be a set of terms. Let  $S$  be a standardized-apart set of pos  $\tau$ -features. Then, domain w.r.t.  $I$  and  $B$  ( $\text{dom}_{I,B}$ ) is a mapping  $\text{dom}_{I,B} : S \rightarrow 2^T$  such that  $\text{dom}_{I,B}(F^+)$  contains all terms  $t$  such that  $(I \wedge B) \models F^+\theta$ , where  $\theta = \{\mathbf{p}(F^+)/t\}$ .  $\blacktriangleright$

Domain assigns to each pos feature a set of terms  $\{t_i\}$ , for which there is a grounding of  $S$  such that  $p(S) = t_i$  and the grounding of  $S$  is true in  $I \wedge B$ . In order to allow efficient computation of domains of pos features, we make the assumption that for every literal  $l$  with a subset of output arguments  $\text{out}_1, \dots, \text{out}_i$  grounded, it is possible to find the set of all possible groundings of this literal efficiently. If this assumption holds, then an algorithm exists, which correctly computes domain and works in time polynomial in the size of  $F^+$  (Algorithm 11). This algorithm is not novel, for  $B = \emptyset$  it corresponds

to an algorithm known as directed-arc-consistency algorithm in the field of CSP and as conjunctive acyclic query algorithm in database theory [50].

Once we have established how domain of a pos feature is computed, it is easy to use this method to decide whether  $(I \wedge B) \models F$ , where  $F$  is a hierarchical feature. If  $l = \text{pred}(X_1, \dots, X_n)$  is root of  $F$ , it suffices to replace  $l$  by  $l' = \text{pred}(X_0, X_1, \dots, X_n)$  in  $F$  and extend background theory to  $B' = B \cup \{\text{pred}(\text{yes}, X_1, \dots, X_n) \leftarrow \text{pred}(X_1, \dots, X_n)\}$ . If domain of this newly created pos feature  $\text{feat}_+(F)$  is non-empty, then  $(I \wedge B) \models F$ . This simplifies notation because we do not need to treat neutral features separately.

**Example 4.3:** Let us have hierarchical feature  $F$  and interpretation  $I$

$$F = \text{hasCar}(C) \wedge \text{hasLoad}(C, L) \wedge \text{tri}(L) \wedge \text{box}(L),$$

$$I = \{\text{hasCar}(c), \text{hasLoad}(c, l1), \text{hasLoad}(c, l2), \text{tri}(l1), \\ \text{circ}(l1), \text{tri}(l2), \text{box}(l2), \text{hasLoad}(c, l3), \text{box}(l4)\},$$

$$B = \emptyset.$$

First, we modify  $F$  so that we could use Algorithm 11 to decide whether  $(I \wedge B) \models F$ , i.e. we replace  $\text{hasCar}(C)$  by  $\text{hasCar}(X, C)$  and extend the background theory  $B = \{\text{hasCar}(\text{yes}, X) \leftarrow \text{hasCar}(X)\}$ .<sup>3</sup> Then we may proceed as follows: (i) We compute domains of pos features  $\text{box}(L)$  and  $\text{tri}(L)$ , i.e.  $\text{dom}_I(\text{box}(L)) = \{l2, l4\}$ ,  $\text{dom}_I(\text{tri}(L)) = \{l1, l2\}$ . (ii) Then we compute domain of pos feature with root  $\text{hasLoad}(C, L)$ , i.e.  $\text{dom}_I(\text{hasLoad}(C, L)) = \{l1, l2, l3\} \cap \{l2\} \cap \{l1, l2\} = \{l2\}$ . (iii) Since no domain is empty so far, we proceed further and compute domain of pos feature with root  $\text{hasCar}(X, C)$ , which becomes  $\text{dom}_I(\text{hasCar}(X, C)) = \{\text{yes}\} \cap \{\text{yes}\}$ . So, we see that  $(I \wedge B) \models F$ .  $\triangle$

The next definition introduces irrelevancy of boolean attribute [28], which enables one to filter such *irrelevant* attributes from dataset.

**Definition 4.7 (Irrelevant Attribute):** Let  $A$  be a set of boolean attributes, let  $\text{cov}(a)$  denote subset of examples covered by  $a \in A$  and let  $\text{pos}(a)$  ( $\text{neg}(a)$ ) denote subset of positive (negative) examples covered by  $a \in A$ . A boolean attribute  $a \in A$  is said to be  $E^+$ -irrelevant ( $E^-$ -irrelevant) if and only if there is  $a' \in A$ ,  $a \neq a'$  such that  $\text{pos}(a) \subseteq \text{pos}(a')$  and  $\text{neg}(a') \subseteq \text{neg}(a)$  ( $\text{neg}(a) \subseteq \text{neg}(a')$  and  $\text{pos}(a') \subseteq \text{pos}(a)$ ). If one of the inclusions, for at least one example, is strict,  $a$  is said to be strictly  $E^+$ -irrelevant

---

<sup>3</sup>We assume that there had been no literals  $\text{hasCar}(X, C)$  before we added them to the original feature and to background theory.



( $E^-$ -irrelevant). A boolean attribute  $a$  is called irrelevant if it is both  $E^+$ -irrelevant and  $E^-$ -irrelevant. It is called strictly irrelevant if it is irrelevant and strictly  $E^+$ -irrelevant or strictly  $E^-$ -irrelevant.  $\blacktriangleright$

In this section, we develop methods for detection of pos features, which give rise to irrelevant features when grafted with neg features.

**Lemma 4.1:** *Let  $I$  be an interpretation,  $B$  background theory and let  $S_1 = \{F_1^+, F_2^+, \dots, F_m^+\}$  and  $S_2 = \{G_1^+, G_2^+, \dots, G_n^+\}$  be standardized-apart sets of pos features such that  $\bigcap_{i=1}^m \text{dom}_{I,B}(F_i^+) \subseteq \bigcap_{i=1}^n \text{dom}_{I,B}(G_i^+)$ , then for any pos-neg feature  $F^-$   $\text{dom}_{I,B}(F^- \oplus_V S_1) \subseteq \text{dom}_{I,B}(F^- \oplus_V S_2)$*

**Proof:** Let us first consider the case when  $\text{depth}_{F^-}(V) = 0$ . The only place, where domains of  $F_i \in S_1$  ( $G_i \in S_2$  respectively) are used, is line 4 in Algorithm 11. Clearly  $\text{argDom}_{S_1} = \bigcap_{i=1}^m \text{dom}_{I,B}(F_i^+) \subseteq \text{argDom}_{S_2} = \bigcap_{i=1}^n \text{dom}_{I,B}(G_i^+)$  and consequently  $\text{litsDom}_{S_1} \subseteq \text{litsDom}_{S_2}$  and therefore also  $\text{dom}_{I,B}(F \oplus_V S_1) \subseteq \text{dom}_{I,B}(F \oplus_V S_2)$ . The general case of lemma may be proved by induction on depth of  $V$ . (i) The case for depth 0 has been already proved. (ii) Let us suppose that lemma holds for depth  $d$ . Now, we suppose that  $\text{depth}_{F^-}(V) = d + 1$ . We may take the pos-neg feature  $T \subset F^-$  which contains  $V$  as output such that  $\text{depth}_T(V) = 0$ ,  $W = \mathbf{p}(T)$  (respecting inputs/outputs of  $F^-$  and  $\mathbf{n}(F^-) = V$ ) and graft it with  $S_1$  ( $S_2$ , respectively). We have  $\text{dom}_{I,B}(T \oplus_V S_1) \subseteq \text{dom}_{I,B}(T \oplus_V S_2)$  and by induction argument finally also  $\text{dom}_{I,B}(F^- \oplus_V S_1) = \text{dom}_{I,B}((F^- \setminus T) \oplus_W \{T \oplus_V S_1\}) \subseteq \text{dom}_{I,B}((F^- \setminus T) \oplus_W \{T \oplus_V S_2\}) = \text{dom}_{I,B}(F^- \oplus_V S_2)$ , which finishes the proof.  $\square$

**Theorem 4.2:** *Let  $E^+$  be a set of positive examples, let  $E^-$  be a set of negative examples and let  $B$  be background theory. Let  $S = \{F_i^+\}_{i=1}^n$  be a set of distinct pos features with equal types of input arguments such that for all  $i \neq j$  there is an example  $I \in E^+ \cup E^-$  with  $\text{dom}_{I,B}(F_i^+) \neq \text{dom}_{I,B}(F_j^+)$ . Let  $\text{dom}_{I,B}(F_1^+) \subseteq \bigcap_{i=2}^n \text{dom}_{I,B}(F_i^+)$  be true for all  $I \in E^+$  ( $I \in E^-$ ) and let  $\bigcap_{i=2}^n \text{dom}_{I,B}(F_i^+) \subseteq \text{dom}_{I,B}(F_1^+)$  be true for all  $I \in E^-$  ( $I \in E^+$ ). (i) For any neg feature  $F^-$ ,  $F^- \oplus \{F_1^+\}$  is  $E^+$ -irrelevant ( $E^-$ -irrelevant). (ii) Let  $S'$  be a set of pos features obtained from  $S$  by repeatedly removing irrelevant pos features. Set  $S'$  is unique.*

**Proof:** We will prove only the case for  $E^+$ -irrelevancy because the proof for  $E^-$ -irrelevancy is analogous.

(i) Let  $F^-$  be a neg feature and let  $F^\pm = \text{feat}_+(F^-)$  be the corresponding pos-neg feature, which has unique  $\mathfrak{p}(F^\pm)$ . By application of Lemma 4.1, if  $\text{dom}_{I,B}(F_1^+) \subseteq \bigcap_{i=2}^n \text{dom}_{I,B}(F_i^+)$  for all  $I \in E^+$ , then  $\text{dom}_{I,B}(F^\pm \oplus \{F_1^+\}) \subseteq \text{dom}_{I,B}(F^\pm \oplus \{F_2^+, \dots, F_n^+\})$  for all  $I \in E^+$ . Similarly, if  $\bigcap_{i=2}^n \text{dom}_{I,B}(F_i^+) \subseteq \text{dom}_{I,B}(F_1^+)$  for all  $I \in E^-$ , then  $\text{dom}_{I,B}(F^\pm \oplus \{F_2^+, \dots, F_n^+\}) \subseteq \text{dom}_{I,B}(F^\pm \oplus \{F_1^+\})$  for all  $I \in E^-$ . Therefore if  $(I \wedge B) \models F^- \oplus \{F_1^+\}$ , then  $(I \wedge B) \models F^- \oplus \{F_2^+, \dots, F_n^+\}$  for all  $I \in E^+$  and similarly if  $(I \wedge B) \models F^- \oplus \{F_2^+, \dots, F_n^+\}$ , then  $(I \wedge B) \models F^- \oplus \{F_1^+\}$  for all  $I \in E^-$ . This means that  $F_1^+$  is  $E^+$ -irrelevant.

(ii) It could happen that by removing some irrelevant pos features from  $S$ ,  $F_1^+$  could become relevant. We need to show that this is not the case. Let  $G^+$  be a graph with vertices corresponding to  $S_i$ . Let there be an edge  $(v_i, v_j)$  if and only if  $\text{dom}_{I,B}(F_i^+) \subseteq \bigcap_{k \in A} \text{dom}_{I,B}(F_k^+)$  for all  $I \in E^+$  and  $\bigcap_{k \in A} \text{dom}_{I,B}(F_k^+) \subseteq \text{dom}_{I,B}(F_i^+)$  for all  $I \in E^-$  and  $j \in A$ ,  $i \notin A$ . Let  $v_i$  be a vertex corresponding to pos feature  $F_i^+$ . Any vertex with non-zero in-degree corresponds to an irrelevant pos feature. If we show that  $G$  is acyclic, then it is easy to find the unique set of relevant pos features as the set of pos features corresponding to vertices with zero in-degree. To show that  $G$  is acyclic, we first notice that if two distinct vertices  $v_i, v_j \in G$  are connected by a directed path, then there is also the edge  $(v_i, v_j)$ . Therefore if there was a directed cycle containing  $(v_1, v_2)$ , there would have to be also two edges  $(v_1, v_2)$  and  $(v_2, v_1)$ . This would mean that for each positive example it would be true that  $\text{dom}_{I,B}(F_2^+) \subseteq \bigcap_{i=3}^n \text{dom}_{I,B}(F_i^+) \cap \text{dom}_{I,B}(F_1^+)$  and  $\text{dom}_{I,B}(F_1^+) \subseteq \bigcap_{i=3}^n \text{dom}_{I,B}(F_i^+) \cap \text{dom}_{I,B}(F_2^+)$ , but then  $\text{dom}_{I,B}(F_1^+) = \text{dom}_{I,B}(F_2^+)$  and the same would be true for negative examples implying  $\text{dom}_{I,B}(F_1^+) = \text{dom}_{I,B}(F_2^+)$  for all examples, which contradicts assumption that no two distinct pos features have equal domains for all examples. Therefore  $G$  must be acyclic and the set of irrelevant pos features (in a given set) must be unique.  $\square$

## 4.2 RELF

### 4.2.1 Algorithm

In this section, we design a propositionalization algorithm RELF (**R**elevant **F**eature **C**onstruction). RELF (Algorithm 12) merges the two usual phases of propositionalization, i.e. feature construction and extension computation. Specifically, the core algorithm accepts a set of learning examples and a hierarchical feature template. The hierarchical

features are obtained by combinatorial composition of pos features, which act as the primitive building blocks. The advantage of this assembly approach is that  $H$ -redundant or irrelevant pos features may be removed from the set of pos features while guaranteeing that all relevant features will be found. In the filtering phase, the algorithm first filters pos features, which have equal domains for all examples, and only after that it also removes irrelevant pos features (thus satisfying conditions of Theorem 4.2). The rules used for detection of  $E^+$ -irrelevant ( $E^-$ -irrelevant) pos features are based on Theorem 4.2. That means  $S_1$  is  $E^+$ -irrelevant if there is set of pos features  $\{S_i\}$  and  $Ind \subset N$  such that  $\text{dom}_{I,B}(S_1) \subseteq \bigcap_{i \in Ind} \text{dom}_{I,B}(S_i)$  on positive examples and  $\bigcap_{i \in Ind} \text{dom}_{I,B}(S_i) \subseteq \text{dom}_{I,B}(S_1)$  on negative examples.

The algorithm exploits the partial irreflexive order, which is imposed on types of arguments by Def. 4.2. Due to existence of this order it is possible to sort all declared predicates  $l \in \gamma$  topologically with respect to a graph induced by the partial order. When the topological order is found, it is possible to organize generation of features in such a way that pos features are built iteratively by combining smaller pos features into larger ones. With input arguments  $\tau$  and  $E$ , where  $\tau$  is a template and  $E$  is a set of examples, it returns a set of hierarchical features  $T_{Alg} \subseteq T_\tau$ , where  $T_\tau$  is set of all correct  $\tau$ -features. An important property of Algorithm 12 is that for any hierarchical feature  $F \in T_\tau$ , which is not strictly irrelevant, there is a hierarchical feature  $F' \in T_{Alg}$  such that  $F$  and  $F'$  cover the same set of examples. In other words, RELF correctly outputs all relevant boolean attributes, which means that it is complete in a well-specified sense.

**Example 4.4 (Running Example):** In Section 4.1.2, we have made the set of correct hierarchical  $\tau$ -features finite. We have shown that, for our particular template  $\tau$ , there are only 18 features. We will now demonstrate how RELF constructs the  $E^+$ -relevant features. We first generate and filter the following three pos features:  $box(L)$ ,  $tri(L)$ ,  $circ(L)$ . We may check that  $circ(L)$  is  $E^+$ -irrelevant (due to  $box(L)$ ), so we may safely throw it away. The next pos features created by grafting with  $box(L)$  and  $tri(L)$  are pos features  $F_1^+ = hasLoad(C, L) \wedge box(L)$ ,  $F_2^+ = hasLoad(C, L) \wedge box(L) \wedge tri(L)$  and  $F_3^+ = hasLoad(C, L) \wedge tri(L)$ . Notice that if we had not removed  $circ(L)$ , there would have been seven such pos features. We can now filter also  $F_1^+, F_2^+, F_3^+$  in the exactly same manner as we filtered  $box(L), tri(L), circ(L)$ . In this case,  $F_2^+$  is  $E^+$ -irrelevant because

$$\begin{aligned} \text{dom}_{I_1}(F_2^+) &= \emptyset \subseteq \text{dom}_{I_1}(F_1^+) \cap \text{dom}_{I_1}(F_3^+) = \{c1\}, \\ \text{dom}_{I_2}(F_2^+) &= \{c2\} \subseteq \text{dom}_{I_2}(F_1^+) \cap \text{dom}_{I_2}(F_3^+) = \{c2\}, \end{aligned}$$

---

**Algorithm 12** RELF (Sketch of Algorithm): Given a template and a set of examples, RELF computes the propositionalized table.

---

```

1: Input: template  $\tau$ , examples  $E$ ;
2:  $PosFeats \leftarrow \{\}$ 
3:  $OrderedDefs \leftarrow$  topologically ordered predicate definitions computed from  $\tau$ 
4: for  $\forall pred \in OrderedDefs$  do
5:    $NewPosFeats \leftarrow \{\}$ 
6:    $NewPosFeats \leftarrow \text{Combine}(pred, PosFeats)$ 
7:   Filter  $H$ -reducible pos features
8:   Filter pos features with equal domains for all examples
9:   Filter irrelevant pos features
10:  Add  $NewPosFeats$  to  $PosFeats$ 
11: end for
12: Save all correct hierarchical features from  $PosFeats$ 

```

---

**Algorithm 13** Combine (Procedure used by RELF): Given a predicate symbol and a set of pos features, **Combine** constructs pos features.

---

```

1: Input:  $predicate$ ,  $PosFeats$ ;
2:  $Constructed \leftarrow \{\}$ 
3:  $ArgCombinations \leftarrow []$ 
4: for  $\forall$  output arguments  $out_i$  of  $predicate$  do
5:    $Smaller \leftarrow$  pos features from  $PosFeats$  with  $type$  of input equal to  $type$  of  $out_i$ 
6:    $ArgCombinations[out_i] \leftarrow$  all combinations without repetition of  $F^+ \in Smaller$ 
7: end for
8:  $Constructed \leftarrow$  all possible graftings of  $predicate(X_1, \dots, X_k)$  with the respective combinations from  $ArgCombinations$ 
9: return  $Constructed$ 

```

---

$$\text{dom}_{I_3}(F_1^+) \cap \text{dom}_{I_3}(F_3^+) = \emptyset \subseteq \text{dom}_{I_3}(F_2^+) = \emptyset,$$

$$\text{dom}_{I_4}(F_1^+) \cap \text{dom}_{I_4}(F_3^+) = \emptyset \subseteq \text{dom}_{I_4}(F_2^+) = \emptyset.$$

Finally, we may graft these pos features with  $car(C')$  to obtain the resulting set of hierarchical features. △

Generating features in this manner can be conveniently combined with computation

of domains. Brief inspection of Algorithm 11 reveals that in order to compute domain of a pos feature we only need to know the domains of its children. However, the domain of any pos feature  $F^+$  can be computed when  $F^+$  is added to the set of already generated pos features and then it can be reused many times. This is exemplified in Fig. 4.1. In this particular example, when pos features are reused (top panel), Algorithm 11 needs to compute domains of only 9 literals as opposed to domains of 15 literals when pos features are not reused (bottom). Naturally, this is magnified for larger feature spaces. Note that for simplicity we assume that subsumption is not rejected for any of the four features.

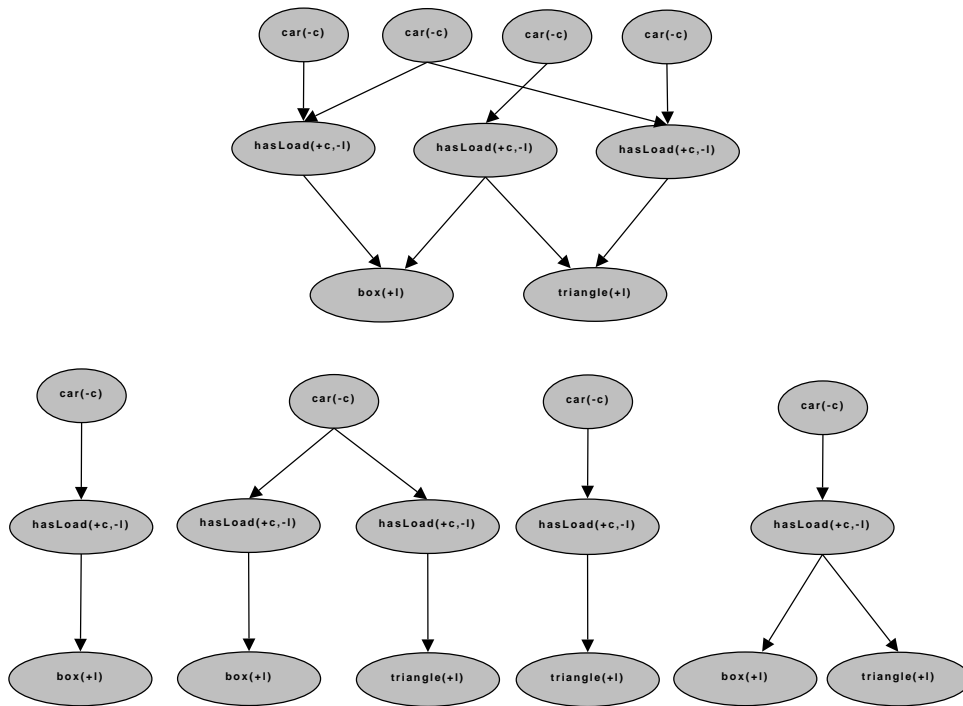


Figure 4.1: Illustration of reuse of pos features for computation of domains.

## 4.2.2 Experiments

In this section we evaluate performance and accuracy of RELF. We evaluate RELF in three relational learning domains in comparison to RSD [44] and Progol [31]. Our intention is to demonstrate (i) that RELF can propositionalize relational data orders of magnitude

faster than standard algorithms and (ii) that classifiers built using features generated by RELF are competitive with those built using more general feature declarations.

In [21] extensive experiments were conducted to compare three state-of-the-art propositionalization systems: RSD, SINUS and RELAGGS. In this study each of the systems obtained best predictive accuracy on exactly two out of six domains. RELAGGS proved itself superior especially in domains where numerical attributes were essential. On the other hand SINUS and RSD performed well in typical ILP tasks such as predicting mutagenicity or learning legal positions of chess-end-games. However, in all experiments RSD was several times faster than SINUS. That is why we chose RSD for comparisons. We also conduct experiments comparing RELF to state-of-the-art ILP system Progol and we also compare our results with those presented in literature.

In all experiments described in this section random forest classifiers<sup>4</sup> are used [5]. We follow suggestions given in [39] to obtain unbiased estimate of quality of learned classifiers. We perform experiments both with RSD having the same feature declaration bias as RELF and with RSD allowing cyclic features. While for RELF the only necessary restriction on hierarchical features is given by the templates (which implicitly restrict their depth), for RSD we also need to bound maximum size of features. We use stratified 10 fold cross-validation. In the experiments with molecular data (Mutagenesis and PTC), we generate propositionalized representation for various maximum depths of features. For each fold, we determine an optimal feature depth by cross-validation on remaining nine folds and then record accuracy obtained on the test fold.

#### 4.2.2.1 Mutagenesis

Our first set of experiments was done on the well-known Mutagenesis dataset [16], which consists of 188 organic molecules marked according to their mutagenicity. We used atom-bond descriptions and numerical attributes *lumo* and *logP*. The longest features found by RELF had over 20 *bond*-literals and were found in 116 seconds. The longest features found by RSD had 3 *bond*-literals and RSD needed 272 seconds. Bigger features could not be found in 15000s by RSD. RSD without acyclic feature bias was not able to find more features than with the acyclic bias.

Table 4.7 displays predictive accuracies obtained on the Mutagenesis dataset. The third column refers to Progol with maximum number of searched nodes set to 10000 and maximum clause length set to 4. Theory construction runtime was 398s. All clauses

---

<sup>4</sup>We used the random forest classifier from Weka package [48].

found by Progol were acyclic. The third column displays accuracy obtained by an ensemble method with a set of theories found by Progol reported in [16], which is to date the highest predictive accuracy for this dataset. However, in this last experiment more information about molecules was used (functional groups and indicator variables). Therefore we repeated our experiments, but we added also the indicator variables and functional groups and obtained accuracy 87.4%. Adding functional groups was not very useful in this case, because RELF was already able to capture the functional groups due to its ability to construct long features.

Algorithm	RELF	RSD	Progol	Progol Ens.
Accuracy [%]	89.8	87.8	82.0	95.8

Table 4.1: Accuracies on Mutagenesis dataset.

#### 4.2.2.2 CAD Documents

The second set of experiments was conducted in a domain describing CAD documents (product structures) [51]. The dataset consists of 96 class-labeled examples. This dataset is interesting because long features are needed to obtain reasonable classification accuracy. For RSD we used both the same template (acyclic bias) and a slightly more general template. We needed to significantly constrain size of RSD’s features to 12 literals for acyclic case (resulting in runtime 12324s) and 11 literals for cyclic case (resulting in runtime 2198s). This is in contrast with RELF, whose longest features had over 50 literals (with runtime 108s). Importantly, the single feature that separated the dataset best was discovered only by RELF. The accuracy of Progol is low due to the fact that Progol is unable to find clauses with sufficient lengths within 15000s limit, which agrees with findings reported in [51].

Algorithm	RELF	RSD	RSD (cyclic)	Progol
Accuracy [%]	96.7	95.5	91.2	81.1

Table 4.2: Accuracies on CAD dataset.

We have performed an additional experiment on CAD dataset to make clear to what extent the speedup achieved by RELF compared to RSD is due to filtering of irrelevant pos

features. With enabled irrelevancy filtering RELF ran 108 seconds, whereas with disabled irrelevancy filtering it crashed after running for several minutes because of lack of free memory. This shows that the key concept, which makes RELF efficient, is irrelevant pos feature filtering.

### 4.2.2.3 Predictive Toxicology Challenge

The last set of experiments was done with data from the Predictive Toxicology Challenge [18]. The PTC dataset consists of 344 organic molecules marked according to their carcinogenicity on male and female mice and rats. Our experiments were done for male rats. Longest features constructed by RELF had over 20 *bond*-literals and were constructed in 762 seconds, while longest features constructed by RSD had only 4 *bond*-literals in 2971 seconds.

Table 4.5 refers to predictive accuracies obtained on the PTC dataset. With Progol, we were unable to obtain any theory compression. The third column in Table 4.5 refers to approach based on *optimal assignment kernel* [11]. The fourth column also refers to approach based on kernels [36]. Predictive accuracy reported for this last approach is the highest presented in literature, however, it is a leave-one-out estimate as opposed to 10-fold cross-validation estimates of the other discussed results.

Algorithm	RELF	RSD	Kernel1	Kernel2
Accuracy [%]	62.5	64.0	63.0	65.7

Table 4.3: Accuracies on PTC dataset for male rats.

## 4.3 HiFi

In this section, we describe propositionalization algorithm HiFi (**H**ierarchical **F**eature construction), which relies more on syntactical restrictions of the set of correct features than on the irrelevancy filtering. Although HiFi is not able to reach feature lengths achievable by RELF, it does not need the information about class labels of examples, therefore, unlike RELF it can be used in unsupervised and semisupervised learning settings.



### 4.3.1 Propositionalization Setting of HiFi

Here, we formally define the propositionalization setting of HiFi because we will pose more constraints on the output set of features. First, we need to state what properties any *canonical ordering*  $\prec_c$  on pos features must have. We describe a simple canonical ordering complying to this definition in the Appendix. The reason why we need an ordering on features is the fact that, if we did not have one, the output of HiFi could be arbitrary to some extent<sup>5</sup>.

**Definition 4.8 (Canonical ordering):** Canonical ordering  $\prec_c$  is a total order on hierarchical (pos) features satisfying the following two conditions:

1. If  $|F_1^+| < |F_2^+|$ , then  $F_1^+ \prec_c F_2^+$ .
2. If  $F_1^+ \prec_c F_2^+$ , then  $F^- \oplus \{F_1^+\} \prec_c F^- \oplus \{F_2^+\}$ .

Here,  $F_1^+$ ,  $F_2^+$  are hierarchical (pos) features and  $F^-$  is a neg feature. ▶

**Definition 4.9 (HiFi Propositionalization):** Let  $\tau = (\gamma, \mu)$  be a hierarchical template, let  $n \in \mathbb{N}$  and  $E$  be an arbitrarily ordered set of examples. Let  $\Phi$  be the set of all hierarchical  $\tau$ -features with size not greater than  $n$ . Hierarchical feature  $F_1 \in \Phi$  is said to be redundant w.r.t.  $E$  and  $\Phi$  if  $F_1$  is irrelevant and there is  $F_2 \in \Phi$ , which covers the same set of examples, and  $F_2 \prec_c F_1$ . The set  $P = \{(f, ext(f)) | f \in F \wedge f \text{ is not redundant w.r.t. } E \text{ and } \Phi\}$  is then called *HiFi propositionalization*. ▶

Note that demanding non-redundancy of features is a weaker assumption than demanding relevancy features, which is demanded in RELF.

### 4.3.2 The Propositionalization Algorithm

In this section, we design a propositionalization algorithm HiFi (Algorithm 14 and more detailed description in Algorithm 17). Similarly as RELF, HiFi merges the two usual phases of propositionalization, i.e. feature construction and extension computation. It produces all features complying to a given template and subsuming some examples. These features are obtained by combinatorial composition of pos features, which act as the primitive building blocks.

---

<sup>5</sup>Another reason is based on an yet unimplemented modification of HiFi, which we do not discuss in this thesis.

The algorithm again exploits the partial irreflexive order, which is imposed on types of arguments by Def. 4.2. First, it sorts all declared predicates  $l \in \gamma$  topologically with respect to a graph induced by the partial order. When the topological order is found, it is possible to organize generation of features in such a way that hierarchical (pos) features are built iteratively by combining smaller pos features into larger ones. Redundant pos features are filtered in a manner analogical to the way RELF filters irrelevant pos features.

**Theorem 4.3:** *Let  $F_1^+, F_2^+$  be pos features with equal domains and let  $F_2^+ \prec_c F_1^+$ . Then any hierarchical feature  $G_1 = F^- \oplus \{F_1^+\}$  is redundant.*

**Proof:** First, notice that any two hierarchical features  $G_1 = F^- \oplus \{F_1^+\}$  and  $G_2 = F^- \oplus \{F_2^+\}$  will necessarily have equal domains, which follows directly from Lemma 4.1. From definition of canonical ordering  $\prec_c$ , it follows that  $G_2 \prec_c G_1$  and therefore any such  $G_1$  must be redundant.  $\square$

Now, we may turn our attention to the problem of bounding the maximum size of a feature. Despite the fact that this may seem as a trivial problem, it is not as trivial. On the other hand, it is not very hard either. In Chapter 5, we show that for general features, bounding features' size is one of the factors that makes it **NP**-hard even to decide whether at least one feature exists. This problem is no longer **NP**-hard if we constrain ourselves to hierarchical features as shown in [46]. However, it is important to stress that an obvious approach, which would discard pos features with size greater than the maximum size limit  $n$ , would not be very efficient because it could often leave many pos features untouched, which could not be extended to sufficiently small hierarchical features. Here, we develop methods to check whether a pos feature may be extended to a hierarchical feature with size not greater than some  $n \in N$  efficiently. The methods will be summarized through the next two lemmas.

Before we step towards stating and proving the lemmas, let us first make some terminology conventions, which will be needed only in the rest of this section. In the next lemmas and in Theorem 4.4, we will talk about *declared predicates from template*  $\tau$ . If  $\tau = (\gamma, \mu)$  is a template, term *declared predicate* will refer to a ground atom  $l \in \gamma$  together with the respective argument places  $\mu_l \subseteq \mu$ , e.g. *hasLoad(+c, -l)* will be a declared predicate. We do this mainly to make the rest of this section more readable since the theoretical framework does not fit here (at least when it comes to readability) very well because we need to talk not only about pos or neg features, but also about the declared predicates alone.

---

**Algorithm 14** *HiFi* (Sketch of Algorithm): Given a hierarchical template and a set of examples, *HiFi* computes the propositionalized table.

---

- 1: **Input:** template  $\tau$ , Integer  $n$  (maximum feature size), examples  $E$ ;
  - 2:  $PosFeatures \leftarrow \{\}$
  - 3:  $OrderedDefs \leftarrow$  topologically ordered predicate definitions computed from  $\tau$
  - 4: **for**  $\forall predicate \in OrderedDefs$  **do**
  - 5:    $NewPosFeatures \leftarrow \{\}$
  - 6:   **for**  $\forall e \in E$  **do**
  - 7:     Add all pos features having *predicate* as its root and covering  $e$ , which were built using pos features from  $PosFeatures$  to  $NewPosFeatures$ .
  - 8:     Record pos eatures with equal domains w.r.t.  $e$
  - 9:   **end for**
  - 10:   Filter redundant pos features (Lemma 4.3) from  $NewPosFeatures$  based on information, which pos features have equal domains for every  $e \in E$
  - 11:   Add pos features from  $NewPosFeatures$  to  $PosFeatures$
  - 12: **end for**
  - 13: Output all hierarchical  $\tau$ -features from  $PosFeatures$
- 

**Lemma 4.2:** *Let  $m$  denote the number of predicates declared in a template  $\tau$  and let  $a$  denote maximum arity of the predicates. Then for all types  $t$ , we can find sizes of the smallest pos features  $F_{min}^+$  such that  $F_{min}^+$  has  $t$  as input type of its root in time  $O(m^2 + m \cdot a)$ .*

**Proof:** We start by finding a topological order of the graph induced by the template. As this graph surely has less than  $m^2$  edges, it is possible to find its topological ordering in time  $O(m^2)$ . When the topological ordering is found, we can take the declared predicates starting with input-only predicates and for every such predicate, we can compute size of the corresponding smallest pos feature. To accomplish this, we sum sizes of smallest pos features whose subroots have input types equal to output types of the processed predicate declaration. These sizes must have been already computed due to the topological order assumed. Since this is done for all  $m$  declared predicates, it follows that computing sizes of the desired smallest pos features takes time  $O(m^2 + m \cdot a)$ .  $\square$

**Lemma 4.3:** *Let  $m$  denote the number of predicates declared in a template  $\tau$  and let  $a$  denote maximum arity of the predicates. Then, for all predicates  $p$ , we can find sizes of*

the smallest hierarchical features  $F_{min}$  containing  $p$  in time  $O(m^2 + m \cdot a)$ .

**Proof:** Using Lemma 4.2, we can translate this problem to a problem of finding shortest paths in an acyclic graph  $G$ . Let  $V$  be a set of vertices of  $G$  and let each vertex correspond to a predicate declared in  $\tau$ . Let there be an oriented edge  $e$  between predicates  $p_1$  and  $p_2$  if and only if type  $t$  of the input argument of  $p_2$  equals to the type of some of the output arguments of  $p_1$ . Further, let the weight of each edge be defined according to Eq. 4.1, where  $MinSize(t)$  refers to minimum size of a feature with input-type of its root equal to  $t$ .

$$w(p_2, p_1) = 1 - MinSize(Input(p_2)) + \sum_{t \in Outputs(p_1)} MinSize(t) \quad (4.1)$$

The size of the smallest feature, which contains  $p$ , is then given as the length of the shortest path from  $p$  to the only vertex with no inputs plus the size of the smallest subfeature having  $p$  as its subroot. The number of edges of  $G$  is bounded by  $m^2$  and the graph is acyclic, thus we can compute all the shortest paths in time  $O(m^2)$ . Combining this with Lemma 4.2, we see that it is possible to compute smallest sizes of features containing some declared predicate in time  $O(m^2 + m \cdot a)$ .  $\square$

Lemma 4.3 enables us to decide whether a pos feature  $F^+$  can be extended to a correct hierarchical feature with size less than some  $n$ . This is because the minimum possible size of a hierarchical feature containing  $F^+$  can be computed as the size of the smallest feature containing root of  $F^+$ , from which we subtract the size of the smallest pos feature having  $F^+$  as its subroot and to which we add the size of  $F^+$ .

**Example 4.5:** Let us compute sizes of the smallest features, which contain some declared predicate, which are defined by the following template.

$$\begin{aligned} \leftarrow car(-c), has1Load(+c, -l), has2Loads(+c, -l, -l), \\ box(+l), triangle(+l) \end{aligned}$$

First we find the topological ordering on types, which is  $(L, C)$ . Then we can compute sizes of smallest pos features with given roots:

$$\begin{aligned} box(+l) \rightarrow 1, triangle(+l) \rightarrow 1, has1Load(+c, -l) \rightarrow 2, \\ has2Loads(+c, -l, -l) \rightarrow 3, car(-c) \rightarrow 3 \end{aligned}$$

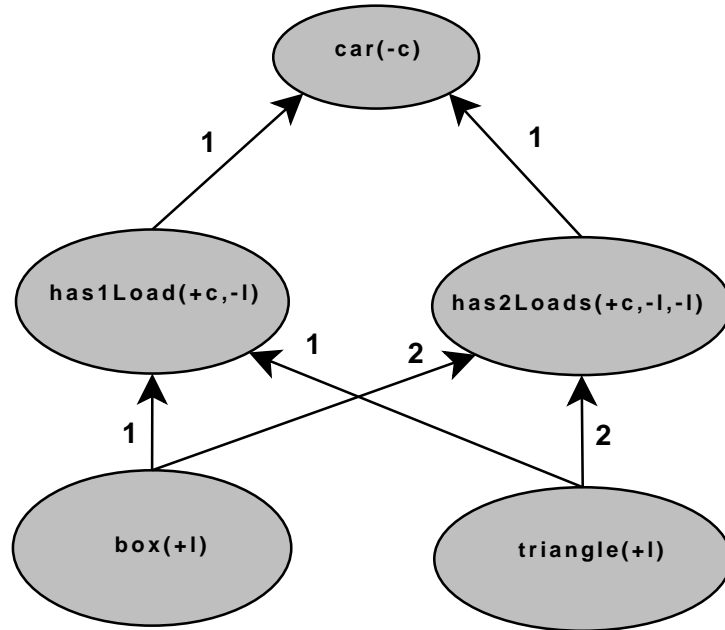


Figure 4.2: An example graph corresponding to template used in Example 4.5. Edge labels are computed from Eq. 4.1.

Now, we need to compute sizes of hierarchical features which contain given declared predicates. In order to do so, we build the corresponding graph (Fig. 4.2) and compute shortest paths. It follows that e.g. the smallest feature containing predicate  $has2Loads(+c, -l, -l)$  has size  $1 + 3 = 4$  (1 is the length of the path from  $has2loads$  to  $car$  and 3 is the size of the smallest subfeature having  $has2loads$  as subroot) and e.g. the smallest feature containing predicate  $box(+l)$  has size  $2 + 1 = 3$ .

Being able to decide whether a pos feature  $F^+$  can be extended to a correct feature  $F$  with size less than  $n$  is an important factor that enables us to prove Theorem 4.4, which states that HiFi runs in time polynomial in  $n$ , whenever the number of generated features is polynomial in  $n$ .

**Theorem 4.4:** *Let  $\tau$  be a template and  $n \in \mathbb{N}$ . Let  $\tau(n) = (\gamma(n), \mu(n))$  be a template such that the number of features correct w.r.t  $n$  and  $\tau(n)$  is polynomial in  $n$ , then Algorithm 17 generates the set of non-redundant features in time polynomial in  $n$ .*

**Proof:** Let us first ignore computation of domains on line 18 of Algorithm 17 and on line 18 of Algorithm 18. Any pos feature, which is stored in the set  $PosFeatures$ , and any combination of pos features, which is generated by procedure  $BuildCombinations(\cdot)$ , is used at least once in the resulting set of generated features. Thus at any time, we

can bound the number of pos features in both of these sets by  $n \cdot P(n)$ , where  $P(n)$  is number of correct features, because no feature can contain more than  $n$  pos features. Brief combinatorial reasoning, which is sketched in proof of Theorem 5.3, implies that generating pos features with root equal to a given predicate  $p$  using already generated pos features takes time polynomial in the number of pos features just being generated. As there are only  $m$  declared predicates, time complexity of Algorithm 17 is polynomial in  $n$ . Now let us consider also computation of domains, which we have ignored so far. We have a guarantee that computation of domains of pos features takes time polynomial in size of the tested features. Lastly, it is also possible to efficiently (in time polynomial in  $n$ ) decide  $H$ -reducibility on line 19 of Algorithm 18 and also to efficiently filter redundant pos features.  $\square$

### 4.3.3 Experimental Evaluation

In this section we evaluate performance and accuracy of HiFi. We evaluate HiFi in three relational learning domains in comparison to RSD [44] and Progol [31]. Similarly as for RELF, our intention is to demonstrate (i) that HiFi can propositionalize relational data orders of magnitude faster than standard algorithms for exhaustive propositionalization and (ii) that classifiers built using features generated by HiFi are not much worse than those built using more general feature declarations.

In all experiments described in this section random forest classifiers [5] were used. We follow suggestions given in [39] to obtain unbiased estimate of quality of learned classifiers. We use stratified 10 fold cross-validation. For each pair of training and test sets we generate propositionalized representation for various lengths (detailed in each subsection). For each fold, we determine an optimal feature length by cross-validation on remaining nine folds and then record accuracy obtained on the test fold. For each dataset we also present runtimes and accuracies as functions of feature lengths.

We perform experiments both with RSD having the same feature declaration bias as HiFi (constraining features to hierarchical ones) and both with RSD allowing non-hierarchical features. In the case when HiFi and RSD have the same bias, their predictive accuracies are nearly identical, because with the same bias they output equivalent propositionalized tables. This is true despite the fact that HiFi filters redundant pos features, because these redundant pos features (respectively features containing these pos features) are also filtered from RSD’s propositionalized tables in its post-processing step. The only difference between propositionalized tables generated by HiFi and RSD

is the order in which features are listed and the canonical ordering used for filtering of redundant features, which causes minor differences between their accuracies. We now note on the meaning of the *feature length* parameter in presence of constants. To allow comparability with RSD, we adopted RSD’s convention to count every constant as one literal, therefore we also report the highest number of atomic bonds in chemical domains or some similar measure of complexity in other domains to provide more intuition about feature sizes.

#### 4.3.3.1 Predictive Toxicology Challenge

The first set of experiments was done with data from the Predictive Toxicology Challenge [18]. The PTC dataset consists of 344 organic molecules marked according to their carcinogenicity on male and female mice and rats. Our experiments were done for male rats. Table 4.4 displays runtimes of HiFi and RSD and their predictive accuracies as functions of the feature lengths (right panel). Despite the fact that we have also performed experiments with RSD without hierarchical feature bias (RSD NH), feature length achievable in reasonable times (under 15000s) with RSD was not enough to find more relevant features than in the case with hierarchical bias (i.e. no cyclic molecular substructures were found), thus we do not report separate predictive accuracies for RSD without hierarchical feature bias. Longest features generated by HiFi had six *bond*-literals.

<b>Length</b>	<b>5</b>	<b>10</b>	<b>15</b>	<b>20</b>	<b>25</b>
<b>HiFi [s]</b>	43	44	56	93	347
<b>RSD (H) [s]</b>	1.6	4.6	73	2971	n.a.
<b>RSD (NH) [s]</b>	1.2	35.5	n.a.	n.a.	n.a.
<b>HiFi/RSD [%]</b>	62.2	64.6.	65.5	64.6	63.4

Table 4.4: Propositionalization runtimes and accuracies for the PTC dataset

Table 4.5 refers to predictive accuracies obtained on the PTC dataset. The first column refers to HiFi’s predictive accuracy using feature length selection described in previous subsection. The second column refers to Progol with which we were unable to obtain any theory compression. The third column refers to approach based on *optimal assignment kernel* [11]. The fourth column also refers to approach based on kernels [36]. Predictive accuracy reported for this last approach is the highest presented in literature,

however, it is a leave-one-out estimate as opposed to 10-fold cross-validation estimates of the other discussed results.

Algorithm	HiFi	RSD	Progol	Kernel1	Kernel2
Accuracy [%]	64.5	64.0	n.a.	63.0	65.7

Table 4.5: Propositionalization results on PTC dataset for male rats.

### 4.3.3.2 Mutagenesis

Our second set of experiments was done on the well-known Mutagenesis dataset [43], which consists of 188 organic molecules<sup>6</sup> marked according to their mutagenicity. We used atom-bond descriptions and numerical attributes *lumo* and *logP*. We used neither functional group descriptions nor indicator attributes *inda* and *ind1*. The predicates, which we used for Mutagenesis dataset, were different from those used for PTC dataset, which is a reason why features with the same *feature length* in Mutagenesis domain have less *bond*-literals than features from PTC domain with the same *feature length*. However, the longest features in the Mutagenesis domain had nine *bond*-literals, which is more than for the PTC dataset. Table 4.6 displays runtimes and predictive accuracies for Mutagenesis dataset. As in the experiments with the PTC dataset, RSD without hierarchical feature bias (RSD NH) was not able to find more features than with the hierarchical bias.

Length	5	10	15	20	25	30	35	40	45	50	55	60	65
HiFi [s]	15	16	17	19	22	33	47	55	85	146	179	230	377
RSD (H) [s]	1.5	2	8	272	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
RSD (NH) [s]	1	5	5303	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.	n.a.
HiFi/RSD [%]	87.2	88.3	87.2	89.3	89.9	90.9	88.3	87.7	89.2	88.1	88.7	87.0	86.5

Table 4.6: Propositionalization runtimes and accuracies for the Mutagenesis dataset

Table 4.7 displays predictive accuracies obtained on the Mutagenesis dataset. The first column refers to HiFi’s predictive accuracy. The third column refers to Progol. The displayed accuracies refer to maximum number of searched nodes set to 10000 and maximum clause length set to 4. Theory construction runtime then was 398s. It is interesting to note that: (i) all clauses found by Progol were hierarchical and (ii) unlike in the case

<sup>6</sup>We work with the regression friendly part of the Mutagenesis dataset.



of the PTC dataset, Progol was able to find a theory with reasonable accuracy within time limit of 15000s. The success of Progol on the Mutagenesis dataset is mainly due to the presence of *lumo* and *logP* parameters and also due to finer description of atom types (e.g. *aromatic carbon atom*), which enables short clauses to describe more complex molecular structures than in the case where only simple atom-bond information is available. The third column displays accuracy obtained by an ensemble method with a set of theories found by Progol [16], which is to date the highest predictive accuracy reported for this dataset. However, in this last experiment more information about molecules was used (namely functional groups and indicator variables). Therefore we repeated our experiments, but we added also the indicator variables and obtained predictive accuracy 89.3% (HiFi+IND), which is, however, only slightly higher than accuracy obtained without the indicator attributes. So, finally we added also functional groups and obtained accuracy 87.3%. We may conclude that adding functional groups was not very useful in this case, which might have been caused by the fact that HiFi was already able to capture these functional groups due to its ability to construct long features.

Algorithm	HiFi	HiFi+IND	RSD	Progol	Progol Ensemble
Accuracy [%]	88.1	89.3	87.1	82.0	95.8

Table 4.7: Propositionalization results on Mutagenesis dataset.

#### 4.3.3.3 CAD Documents

The third set of experiments was conducted in a domain describing CAD documents (product structures) [51]. The dataset consists of 96 class-labeled examples each containing several hundreds of literals. Table 4.8 displays propositionalization runtimes and obtained accuracies for this dataset. The differences between runtimes of RSD with and without the hierarchical feature bias are not so dramatic as in the other experiments, which is mainly due to the fact that the examples are nearly hierarchical.

Table 4.9 displays predictive accuracies obtained for the CAD dataset. The first column refers to accuracies obtained by using features found by HiFi. The second column refers to accuracy obtained by Progol. The accuracy is low due to the fact that Progol is unable to find clauses with sufficient lengths within 15000s limit, which agrees with findings reported in [51].

<b>Length</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>	<b>11</b>	<b>12</b>	<b>13</b>	<b>14</b>
<b>HiFi [s]</b>	9	9	10	10	12	15	23	39	70
<b>RSD (H) [s]</b>	0.5	2	8	45	310	1749	12324	n.a.	n.a.
<b>RSD (NH) [s]</b>	0.5	2	8	48	324	2198	n.a.	n.a.	n.a.
<b>HiFi/RSD (H) [%]</b>	88.4	85.2	96.8	96.8	97.8	95.6	95.6	94.5	95.6
<b>RSD (NH) [%]</b>	85.4	92.7	92.7	96.8	94.8	96.8	n.a.	n.a.	n.a.

Table 4.8: Propositionalization runtimes and accuracies for the CAD dataset

<b>Algorithm</b>	HiFi	RSD	Progol
<b>Accuracy [%]</b>	94.5	95.5	81.1

Table 4.9: Propositionalization results on CAD dataset.

#### 4.3.3.4 Evaluation of HiFi for Feature Construction

In this section, we do not evaluate HiFi on another dataset, but instead we evaluate ability of HiFi to construct relational features without extension computation<sup>7</sup>. For this experiment, we stick to Michalski’s train domain. Let the template  $\tau$  be defined as follows

$$\tau = \text{car}(-c), \text{hasRoof}(+c), \text{twoWheels}(+c), \text{threeWheels}(+c),$$

$$\text{hasLoad}(+c[3], -load), \text{triangle}(+load), \text{box}(+load), \text{circle}(+load)$$

where number 3 denotes the fact that only three *hasLoad*-literals can be used. Fig. 4.3 displays runtimes of HiFi and RSD needed to generate all correct features. HiFi is about two orders of magnitude faster than RSD. Fig. 4.3 indicates that runtime of both evaluated algorithms is bounded. However, while for RSD this is due to the limit of maximum three *loads* and without this limit runtime of RSD would be unbounded, for HiFi runtime would remain bounded even if no limit on number of *loads* was set. The reason is pruning of *H*-reducible (pos) features done by HiFi, which causes that the number of generated features is bounded.

<sup>7</sup>In HiFi we make all subsumption checks between (pos) features and examples return *true*.

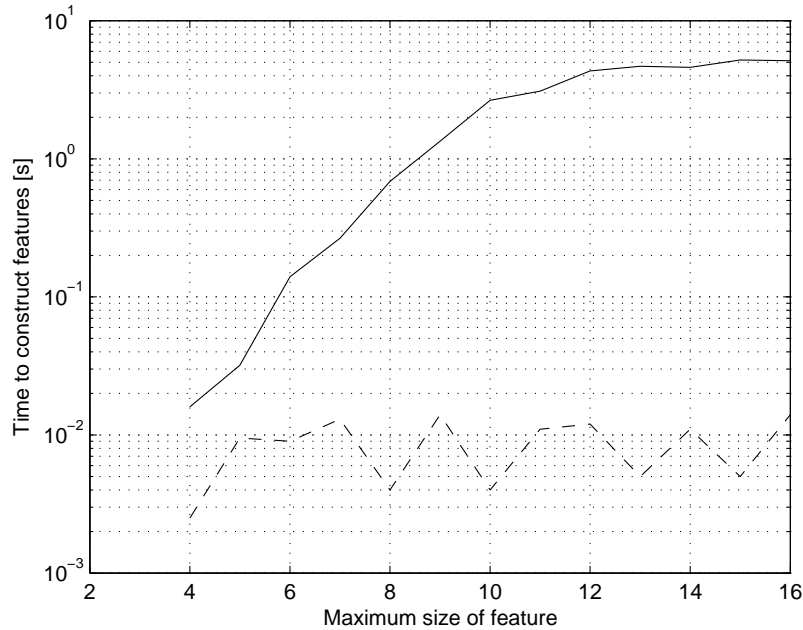


Figure 4.3: Feature construction times for RSD (solid) and HiFi (dashed)

## 4.4 Discussion of Experiments with RELF and HiFi

Experiments presented in the preceding sections clearly indicate that RELF and HiFi outperform state-of-the-art systems by several orders of magnitude for propositionalization with the hierarchical bias. RELF and HiFi were also shown to be significantly faster than Progol. Although Progol is a much more general system, it did not find any non-hierarchical features in our experiments and thus did not come up with anything unachievable by RELF or HiFi.

In most cases, we have been also able to achieve predictive accuracies close to the best results reported in literature. On two out of three datasets, RELF outperformed HiFi. RELF's ability to construct very long features makes it also quite appealing also from the datamining point of view. On the other hand, HiFi is applicable also to unsupervised or semi-supervised learning, which is not possible for RELF. Therefore we believe that both RELF and HiFi should have its place.

Although there are definitely datasets where cyclic features could provide better predictive accuracies, one can always merge results from different propositionalization algorithms and feed the propositional learners with such merged propositionalized tables. Algorithms for construction of a limited class of features such as RELF and HiFi would

be useful also in such cases.

# Chapter 5

## Complexity of Feature Construction

In this chapter, we elaborate complexity of feature construction. We will ignore extension computations and the fact that it is possible to prune some features, which do not cover any example, i.e. we will focus solely on the generation of features constrained by templates. We believe it is useful to elaborate complexity of this problem because the setting with syntactical constraints given by templates, which is used throughout this thesis, has already been used in ILP [44], but its complexity was unknown<sup>1</sup>.

In the technical discussion in this chapter, we will use unary representation of numbers used to bound sizes of features. This is because a number  $n$  can be represented by  $O(\log n)$  bits, which could allow existence of features with size exponential in the input size (in the combined size of the template and of the binary representation of  $n$ ). By choosing to use the unary representation of numerical parameters, we will be dealing with the so called strong **NP**-completeness [42].

### 5.1 A Negative Result

**Lemma 5.1:** *Let  $\tau = (\gamma, \mu)$  be a template and  $n \in \mathbb{N}$  be a number represented in unary notation (i.e. number  $n$  is represented by a string  $111 \dots 1$  with  $n$  ones). The problem of deciding, whether there is at least one feature correct w.r.t.  $\tau$  and  $n$ , is in **NP**.*

**Proof:** (Sketch) It suffices to show how to construct a polynomial-sized certificate and a polynomial-time verifier. The certificate will be a pair  $(F, \theta)$ , where  $F$  is a conjunction

---

<sup>1</sup>A proof of **NP**-hardness of the feature existence problem was promised to be given soon in a work-in-progress paper [45], but it has not been shown since then.

of literals such that  $|F| \leq n$  and  $\theta$  is a substitution. Clearly, the certificate has size polynomial in  $|\tau|$  and in the unary representation of  $n$ .

Now, it remains to show how we can verify correctness of the solution in polynomial time, but this is easy. First, we check that  $|F| \leq n$  and that  $\text{lits}(F\theta) \subseteq \gamma$ . Then we check whether every variable appears both as an input and as an output and whether every variable appears only once as an output, which is easy because  $\theta$  uniquely determines, which variable appearance is an input and which is an output. If none of these checks fails, we may output *yes* and accept the feature.  $\square$

**Lemma 5.2:** *Let  $\tau = (\gamma, \mu)$  be a template and  $n \in \mathbb{N}$  be a number represented in unary notation (i.e. number  $n$  is represented by a string  $111 \dots 1$  with  $n$  ones). The problem of deciding, whether there is at least one feature correct w.r.t.  $\tau$  and  $n$ , is **NP-hard**.*

**Proof:** (Sketch) We will prove this theorem by reduction from the graph coloring problem. Let  $G = (V, E)$  be the graph to be colored and let the set of *colors* be  $\{\text{red}, \text{green}, \text{blue}\}$ . First, we make the edges oriented such that an edge between two vertices  $v_i, v_j$  will be pointing from  $v_i$  to  $v_j$  if  $i < j$ . The template  $\tau$  will be constructed as follows. The first declared predicate will be

$$\text{graph}(-e_1, -e_2, \dots, -e_{|E|}, -v_1, \dots, -v_{|V|})$$

where each  $e_i$  will correspond to one edge  $e \in E$  ( $n = |E|$ ). We add the six following declared predicates (called *edge predicates*)

$$\text{edge}_{rb}(+\text{rout}_{e_k}, +e_k, -\text{bin}_{e_k}), \text{edge}_{br}(+\text{bout}_{e_k}, +e_k, -\text{rin}_{e_k}), \text{edge}_{rg}(+\text{rout}_{e_k}, +e_k, -\text{gin}_{e_k}),$$

$$\text{edge}_{gr}(+\text{gout}_{e_k}, +e_k, -\text{rin}_{e_k}), \text{edge}_{bg}(+\text{bout}_{e_k}, +e_k, -\text{gin}_{e_k}), \text{edge}_{gb}(+\text{gout}_{e_k}, +e_k, -\text{bin}_{e_k})$$

for each edge  $e_k \in E$  pointing from  $v_i$  to  $v_j$ . Next, for each vertex  $v_i$  we add three declared predicates (called *vertex predicates*)

$$\text{red}_i(+v_i, +\text{rin}_{e_{i_1}}, \dots, +\text{rin}_{e_{i_n}}, -\text{rout}_{e_{j_1}}, \dots, -\text{rout}_{e_{j_m}})$$

$$\text{green}_i(+v_i, +\text{gin}_{e_{i_1}}, \dots, +\text{gin}_{e_{i_n}}, -\text{gout}_{e_{j_1}}, \dots, -\text{gout}_{e_{j_m}})$$

$$\text{blue}_i(+v_i, +\text{bin}_{e_{i_1}}, \dots, +\text{bin}_{e_{i_n}}, -\text{bout}_{e_{j_1}}, \dots, -\text{bout}_{e_{j_m}}),$$

where each of the above declared predicates corresponds to a vertex  $v_i$  and  $v_{i_1}, \dots, v_{i_n}$  correspond to vertices from which an edge points to  $v_i$  and vertices  $v_{j_1}, \dots, v_{j_m}$  correspond to vertices to which an edge points from  $v_i$ . Finally, we set the maximum feature size  $n = 1 + |V| + |E|$ .

It is trivial to check that the above construction is polynomial-time in the size of  $G$ . Now, it remains to show that a feature  $F$ ,  $|F| \leq 1 + |V| + |E|$  exists, which complies to the above constructed template, if and only if a 3-coloring of  $G$  exists.

( $\Rightarrow$ ) First, notice that there must be exactly one edge predicate for each  $e \in E$  and exactly one vertex predicate for each  $v \in V$  in a correct feature  $F$ . Otherwise, there would be an unsatisfied output  $v_i$  or  $e_i$ . Therefore each vertex has exactly one color represented by a vertex predicate. Furthermore, colors of two adjacent vertices must be different due to types of input and output arguments of the respective vertex predicates. So, the coloring given by the vertex predicates represents a correct coloring of  $G$ .

( $\Leftarrow$ ) If we have a coloring of  $G = (V, E)$ , we may construct a correct feature with size bounded  $|F| = 1 + |V| + |E|$ . We add one literal  $graph(-e_1, -e_2, \dots, -e_{|E|}, -v_1, \dots, -v_{|V|})$ . Next, we add one vertex literal for each  $v \in V$  (choosing the particular predicate according to the coloring of the given vertex). Finally, we add one edge literal for each  $e \in G$  (again, we choose the particular predicate according to the coloring of the vertices connected by this edge). It is easy to check that this corresponds to a correct feature.  $\square$

**Theorem 5.1:** *Let  $\tau = (\gamma, \mu)$  be a template and  $n \in N$  be a number represented in unary notation (i.e. number  $n$  is represented by a string  $111\dots 1$  with  $n$  ones). The problem of deciding, whether there is at least one feature correct w.r.t.  $\tau$  and  $n$  (called feature existence problem), is **NP**-complete.*

**Proof:** Follows directly from Lemmas 5.1 and 5.2.  $\square$

.

The next example shows a conversion of an instance of the graph coloring problem to an instance of the feature existence problem.

**Example 5.1:** Let us assume that we should find a coloring of the graph shown in the left-most panel of Fig. 5.1. First, we make the graph directed as shown in the center panel of Fig 5.1. Then, the template constructed according to the proof of Lemma 5.2 will look as follows:

$$\begin{aligned} & graph(-e_1, -e_2, -e_3, -e_4, -e_5, -e_6, -e_7, -v_1, -v_2, -v_3, -v_4, -v_5) \\ & red_1(+v_1, -rout_{e_1}, -rout_{e_2}, -rout_{e_3}), green_1(+v_1, -gout_{e_1}, -gout_{e_2}, -gout_{e_3}), \\ & blue_1(+v_1, -bout_{e_1}, -bout_{e_2}, -bout_{e_3}), red_2(+v_2, +rin_{e_1}, -rout_{e_4}), \\ & red_2(+v_2, +gin_{e_1}, -gout_{e_4}), blue_2(+v_2, +bin_{e_1}, -bout_{e_4}), \end{aligned}$$

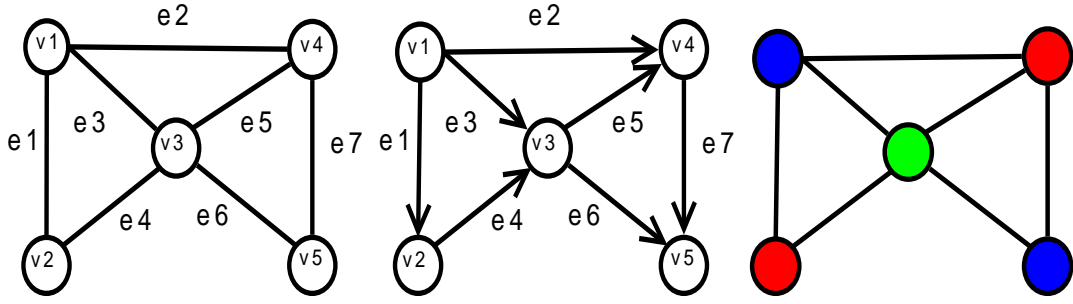


Figure 5.1: Illustration of proof of Theorem 5.2

$$\begin{aligned}
& red_3(+v_3, +rin_{e_3}, +rin_{e_4}, -rout_{e_5}, -rout_{e_6}), green_3(+v_3, +gin_{e_3}, +gin_{e_4}, -gout_{e_5}, -gout_{e_6}), \\
& blue_3(+v_3, +bin_{e_3}, +bin_{e_4}, -bout_{e_5}, -bout_{e_6}), red_4(+v_4, +rin_{e_2}, +rin_{e_5}, -rout_{e_7}), \\
& green_4(+v_4, +gin_{e_2}, +gin_{e_5}, -gout_{e_7}), blue_4(+v_4, +bin_{e_2}, +bin_{e_5}, -bout_{e_7}), \\
& red_5(+v_5, +rin_{e_6}, +rin_{e_7}), green_5(+v_5, +gin_{e_6}, +gin_{e_7}), blue_5(+v_5, +bin_{e_6}, +bin_{e_7}), \\
& edge_{rb}(+rout_{e_1}, +e_1, -bin_{e_1}), edge_{br}(+bout_{e_1}, +e_1, -rin_{e_1}), edge_{rg}(+rout_{e_1}, +e_1, -gin_{e_1}), \\
& edge_{gr}(+gout_{e_1}, +e_1, -rin_{e_1}), edge_{bg}(+bout_{e_1}, +e_1, -gin_{e_1}), edge_{gb}(+gout_{e_1}, +e_1, -bin_{e_1}), \\
& edge_{rb}(+rout_{e_2}, +e_2, -bin_{e_2}), edge_{br}(+bout_{e_2}, +e_2, -rin_{e_2}), edge_{rg}(+rout_{e_2}, +e_2, -gin_{e_2}), \\
& edge_{gr}(+gout_{e_2}, +e_2, -rin_{e_2}), edge_{bg}(+bout_{e_2}, +e_2, -gin_{e_2}), edge_{gb}(+gout_{e_2}, +e_2, -bin_{e_2}), \\
& edge_{rb}(+rout_{e_3}, +e_3, -bin_{e_3}), edge_{br}(+bout_{e_3}, +e_3, -rin_{e_3}), edge_{rg}(+rout_{e_3}, +e_3, -gin_{e_3}), \\
& edge_{gr}(+gout_{e_3}, +e_3, -rin_{e_3}), edge_{bg}(+bout_{e_3}, +e_3, -gin_{e_3}), edge_{gb}(+gout_{e_3}, +e_3, -bin_{e_3}), \\
& edge_{rb}(+rout_{e_4}, +e_4, -bin_{e_4}), edge_{br}(+bout_{e_4}, +e_4, -rin_{e_4}), edge_{rg}(+rout_{e_4}, +e_4, -gin_{e_4}), \\
& edge_{gr}(+gout_{e_4}, +e_4, -rin_{e_4}), edge_{bg}(+bout_{e_4}, +e_4, -gin_{e_4}), edge_{gb}(+bout_{e_4}, +e_4, -gin_{e_4}), \\
& edge_{rb}(+rout_{e_5}, +e_5, -bin_{e_5}), edge_{br}(+bout_{e_5}, +e_5, -rin_{e_5}), edge_{rg}(+rout_{e_5}, +e_5, -gin_{e_5}), \\
& edge_{gr}(+gout_{e_5}, +e_5, -rin_{e_5}), edge_{bg}(+bout_{e_5}, +e_5, -gin_{e_5}), edge_{gb}(+gout_{e_5}, +e_5, -bin_{e_5}), \\
& edge_{rb}(+rout_{e_6}, +e_6, -bin_{e_6}), edge_{br}(+bout_{e_6}, +e_6, -rin_{e_6}), edge_{rg}(+rout_{e_6}, +e_6, -gin_{e_6}), \\
& edge_{gr}(+gout_{e_6}, +e_6, rin_{e_6}), edge_{bg}(+bout_{e_6}, +e_6, -gin_{e_6}), edge_{gb}(+gout_{e_6}, +e_6, -bin_{e_6}), \\
& edge_{rb}(+rout_{e_7}, +e_7, -bin_{e_7}), edge_{br}(+bout_{e_7}, +e_7, -rin_{e_7}), edge_{rg}(+rout_{e_7}, +e_7, -gin_{e_7}), \\
& edge_{gr}(+gout_{e_7}, +e_7, -rin_{e_7}), edge_{bg}(+bout_{e_7}, +e_7, -gin_{e_7}), edge_{gb}(+gout_{e_7}, +e_7, -bin_{e_7}).
\end{aligned}$$



Now, any correct feature represents a coloring of the graph in Fig. 5.1. Let us construct a feature that will represent the particular coloring shown in the right-most panel as an example. The feature will look as follows

$$\begin{aligned}
& \text{graph}(E_1, E_2, E_3, E_4, E_5, E_6, E_7, V_1, V_2, V_3, V_4, V_5), \text{blue}_1(V_1, \text{Bout}_{e_1}, \text{Bout}_{e_2}, \text{Bout}_{e_3}), \\
& \quad \text{red}_2(V_2, \text{Rin}_{e_1}, \text{Rout}_{e_4}), \text{green}_3(V_3, \text{Gin}_{e_3}, \text{Gin}_{e_4}, \text{Gout}_{e_5}, \text{Gout}_{e_6}), \\
& \text{red}_4(V_4, \text{Rin}_{e_2}, \text{Rin}_{e_5}, \text{Rout}_{e_7}), \text{blue}_5(V_5, \text{Bin}_{e_6}, \text{Bin}_{e_7}), \text{edge}_{br}(\text{Bout}_{e_1}, E_1, \text{Rin}_{e_1}), \\
& \quad \text{edge}_{br}(\text{Bout}_{e_2}, E_2, \text{Rin}_{e_2}), \text{edge}_{bg}(\text{Bout}_{e_3}, E_3, \text{Gin}_{e_3}), \text{edge}_{rg}(\text{Rout}_{e_4}, E_4, \text{Gin}_{e_3}), \\
& \quad \text{edge}_{gr}(\text{Gout}_{e_5}, E_5, \text{Rin}_{e_5}), \text{edge}_{gb}(\text{Gout}_{e_6}, E_6, \text{Bin}_{e_6}), \text{edge}_{rb}(\text{Rout}_{e_7}, E_7, \text{Bin}_{e_7})
\end{aligned}$$

Even though showing that the feature existence problem is **NP**-complete could suffice for our discussion here, we believe that it is also interesting to see what is the obstacle which prevents us from being able to solve the general case of feature existence problem by the Horn-SAT reduction described in 2.4.4.4. The problem is that in order to decide whether a feature with length less than  $n$  exists, it is necessary (in the worst case) to generate a model of the respective Horn-SAT problem, which would maximize number of variables with value *true* (recall that *false* value means that a literal is added to a feature). This problem can be shown to be **NP**-hard by reduction from MAX Horn-SAT, which is an **NP**-hard problem [19].

**Theorem 5.2:** *Finding a solution to a Horn-SAT problem, which makes the maximum number of variables true, is an **NP**-hard problem.*

**Proof:** (Sketch) Let a given set  $H$  of Horn clauses contain the following clauses:

$$\begin{aligned}
& P_{i_1,1} \vee \neg P_{i_1,2} \vee \dots \vee \neg P_{i_1,n} \\
& \quad \dots \\
& P_{i_k,1} \vee \neg P_{i_k,2} \vee \dots \vee \neg P_{i_k,n} \\
& \quad \neg P_{i_{k+1},2} \vee \dots \vee \neg P_{i_{k+1},n} \\
& \quad \dots \\
& \quad \neg P_{i_h,2} \vee \dots \vee \neg P_{i_h,n}
\end{aligned}$$

Let  $m$  be the number of boolean variables in the set. Our task (MAX Horn-SAT) is to make as much of these clauses as possible satisfied. We create  $2 \cdot m$  copies of each clause

and we add one negated variable  $D_i^j$  to each of these copies (variable  $D_i^j$  corresponds to  $j$ -th copy of  $i$ -th clause). The new set  $NH$  of clauses will look as follows:

$$\begin{aligned}
& P_{i_1,1} \vee \neg P_{i_1,2} \vee \dots \vee \neg P_{i_1,n} \vee \neg D_1^1 \\
& \dots \\
& P_{i_1,1} \vee \neg P_{i_1,2} \vee \dots \vee \neg P_{i_1,n} \vee \neg D_1^{2 \cdot m} \\
& \dots \\
& P_{i_k,1} \vee \neg P_{i_k,2} \vee \dots \vee \neg P_{i_k,n} \vee \neg D_k^{2 \cdot m} \\
& \neg P_{i_{k+1},1} \vee \dots \vee \neg P_{i_{k+1},n} \vee \neg D_{k+1}^1 \\
& \dots \\
& \neg P_{i_{k+1},1} \vee \dots \vee \neg P_{i_{k+1},n} \vee \neg D_{k+1}^{2 \cdot m} \\
& \dots \\
& \neg P_{i_h,1} \vee \dots \vee \neg P_{i_h,n} \vee \neg D_h^{2 \cdot m}
\end{aligned}$$

We claim that a solution of this Horn-SAT problem, which maximizes the number of variables with *true* value, also encodes a solution of the original MAX Horn-SAT problem.

Let us denote  $\mathcal{P}$  the set of variables in the original set of clauses and  $\mathcal{D}$  the set of the new variables  $D_i^j$ . It holds (to simplify notation, we use 0 for *false* and 1 for *true*):

$$\begin{aligned}
& \arg \max_{m \in \text{models}(NH)} \left( \sum_{P_i \in \mathcal{P}} P_i + \sum_{D_i \in \mathcal{D}} D_i \right) = \\
& = \arg \max_{m \in \text{models}(NH)} \left( \sum_{P_i \in \mathcal{P}} P_i + 2m \cdot \#(\text{clauses from } H \text{ satisfied by model } m) \right) \quad (5.1)
\end{aligned}$$

The equality above holds because when a clause is satisfied, the best what we can do to increase the criterion is to make all the corresponding variables  $D_i^j$  *true* as this increases the criterion by  $2 \cdot m$ . Also, notice that if the criterion in Eq. 5.1 is maximized, then

$$2m \cdot \#(\text{clauses from } H \text{ satisfied by model } m)$$

and consequently also

$$\#(\text{clauses from } H \text{ satisfied by model } m)$$

are maximized as well, because  $\sum_{P_i \in \mathcal{P}} P_i < 2m$ . This finishes the proof.  $\square$

Finding correct features with respect to some template  $\tau$  is a hard problem, as we have seen. Its inherent hardness is caused by the fact that for some templates  $\tau$ , correct  $\tau$ -features have some minimum length. This is contrasted by numerous frequent pattern mining systems (e.g. WARMR [9]), which do not bound size of possible features from below. At first sight, the framework which we use could seem to bring us higher theoretical complexity than is the complexity encountered by these systems. A more careful look, however, reveals that this complexity exhibits also in WARMR where it is manifested through bigger number of features (patterns), which are generated. Thus, if we have apriori knowledge about possible structure of informative features, we may provide this knowledge to a propositionalization system by means of templates. If no such knowledge is available, it is possible to create such templates, which will lead to roughly the same sets of features as the sets generated by WARMR.

## 5.2 A Positive Result

We have already seen that if the number of hierarchical features is polynomial in  $n$ , the problem of enumerating the hierarchical features can be solved in time polynomial in  $n$  and in the total number of hierarchical features even if we consider that some examples are given (cf. Theorem 4.4). Here, we strengthen the results for the case of sole feature construction and show that if HiFi does not consider examples, it runs in output-polynomial time.

**Theorem 5.3:** *Let  $\tau(n) = (\gamma(n), \mu(n))$  be a template such that  $|\gamma(n)| = O(n^c)$  for some  $c \geq 1$ , then the feature construction part of HiFi (Algorithm 17 without extension computation and redundancy filtering) can construct all correct hierarchical features w.r.t.  $\tau$  and  $n$  in output-polynomial time.*

**Proof:** Any pos feature, which is stored in the set *PosFeatures*, and any combination of pos features, which is generated by procedure *BuildCombinations(.)*, is used at least once in the resulting set of generated features, which is guaranteed by Lemma 4.3. Thus at any time, we can bound the number of pos features in both of these sets by  $n \cdot P(n)$ , where  $P(n)$  is number of correct features, because no feature can contain more than  $n$  pos features (recall that we say that a pos feature  $F^+$  is contained in a feature if and only if  $F^+ \subset F$  and  $F \setminus F^+$  is a neg feature).

Brief combinatorial reasoning implies that generating pos features with root equal to a given predicate  $p$  using already generated pos features takes time polynomial in the number of pos features just being generated. In more detail: combinations of already generated pos features are created iteratively. At each step the new combinations of pos features are combined with single pos features from the set  $PosFeatures$ ; each such step takes time at most quadratic in the size of the already generated combinations and, always, every generated combination of pos features appears at least in one correct feature. Thus generation of combinations of pos features is polynomial in the total number of features correct w.r.t.  $\tau$  and  $n$ . An analogical reasoning can be applied also to generation of pos features with a given predicate of their root from these combinations.

As there are only  $O(n^c)$  declared predicates, time complexity of Algorithm 17 is polynomial in  $P(n)$ .  $\square$

# Chapter 6

## Conclusions

In this thesis, we have studied several aspects of propositionalization. In Chapter 2, we have described necessary background information and we have briefly described several prominent propositionalization systems. In Chapter 3, we have developed  $\theta$ -subsumption algorithms RESUMER2 and RECOVER. The main principle behind these two algorithms is utilization of randomized restarted strategies. RESUMER2 is a complete heuristic randomized algorithm, whose completeness is guaranteed by use of unbounded cutoff sequences. It uses a modified restarted strategy where subsequent pairs of restarts are statistically independent. On real-life data from PTC domain [18], RESUMER2 was shown to significantly outperform state-of-the-art  $\theta$ -subsumption algorithm Django [30] for cases of complex data. The other algorithm, RECOVER, is a heuristic estimation algorithm, which uses restarts to obtain a maximum-likelihood estimate of coverage (RECOVER) or extension (RECOVER-E). In experiments, we have shown that RECOVER's accuracy can be sufficient to achieve good accuracies and that its runtimes are favorable with respect to Django.

In Chapter 4, we have developed algorithms RELF and HiFi for propositionalization constrained to hierarchical features. The algorithms exploit properties of hierarchical features. The first algorithm, RELF, exploits mostly monotonicity of irrelevancy and redundancy. The second algorithm, HiFi, focuses more on syntactical restrictions of features. In experiments on three real-life datasets, we have shown that RELF and HiFi are able to achieve feature lengths unachievable by state-of-the-art systems RSD [44] and Progol [31]. We have also shown that predictive accuracies obtained by RELF and HiFi in conjunction with random forest classifiers [5] are close to the best results reported in literature. A slightly higher accuracies were obtained with RELF.

Finally in Chapter 5, we have elaborated complexity of feature construction. We have

shown that deciding whether a feature exists complying to a template  $\tau$  and with size less than some  $n \in N$ , is **NP**-complete for general features. We have also shown that if we restrict ourselves to hierarchical features, all features with respect to a template  $\tau$  and with size less than some  $n \in N$  can be enumerated in output-polynomial time.

# Bibliography

- [1] Marta Arias, Roni Khardon, and Jerome Maloberti. Learning Horn expressions with Logan-H. *Journal of Machine Learning Research*, 8:549–587, 2007.
- [2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 1 edition, 2009.
- [3] Christian Bessiere and Jean-Charles Regin. MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In *Principles and Practice of Constraint Programming*, pages 61–75, 1996.
- [4] Bela Bollobas. *Modern Graph Theory*. Springer, 1998.
- [5] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.
- [6] H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 408–421. Springer-Verlag, 2001.
- [7] Chad M. Cumby and Dan Roth. Learning with feature description logics. In *Inductive Logic Programming, 12th International Conference*, pages 32–47, 2002.
- [8] Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.
- [9] Luc Dehaspe and Hannu Toivonen. Discovery of frequent datalog patterns. *Data Min. Knowl. Discov.*, 3(1):7–36, 1999.
- [10] Saso Dzeroski and Nada Lavrač. *Relational Data Mining*. Springer, 2001.
- [11] Holger Fröhlich, Jörg K. Wegner, Florian Sieker, and Andreas Zell. Optimal assignment kernels for attributed molecular graphs. In *ICML '05: Proceedings of the 22nd international conference on Machine learning*, pages 225–232, New York, NY, USA, 2005. ACM.

- [12] A. Giordana and L. Saitta. Phase transitions in relational learning. *Machine Learning*, 41(2):217–251, 2000.
- [13] Carla P. Gomes, Cèsar Fernández, Bart Selman, and Christian Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.
- [14] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
- [15] Georg Gottlob, Nicola Leone, and Francesco Scarcello. On the complexity of some inductive logic programming problems. *New Gen. Comput.*, 17(1):53–75, 1999.
- [16] S. Muggleton H. Lodhi. Is mutagenesis still challenging. In *ILP-05 Late-Breaking Papers*, pages 35–40, 2005.
- [17] W. D. Harvey. *Nonsystematic backtracking search*. PhD thesis, Stanford University, 1995.
- [18] C. Helma, R. D. King, S. Kramer, and A. Srinivasan. The predictive toxicology challenge 2000-2001. *Bioinformatics*, 17(1):107–108, 2001.
- [19] Brigitte Jaumard and Bruno Simeone. On the complexity of the maximum satisfiability problem for horn formulas. *Inf. Process. Lett.*, 26(1):1–4, 1987.
- [20] S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 27 May 1994.
- [21] Mark-A. Krogel, Simon Rawles, Filip Železný, Peter A. Flach, Nada Lavrač, and Stefan Wrobel. Comparative evaluation of approaches to propositionalization. In *Proceedings of the 13th International Conference on Inductive Logic Programming*, pages 197–214. Springer, 2003.
- [22] Mark-A. Krogel and Stefan Wrobel. Transformation-based learning using multirelational aggregation. In *ILP '01: Proceedings of the 11th International Conference on Inductive Logic Programming*, pages 142–155, London, UK, 2001. Springer-Verlag.
- [23] Ondřej Kuželka. A statistical study of a combinatorial problem. bachelor’s thesis, czech technical university, 2007.



- [24] Ondřej Kuželka and Filip Železný. Fast estimation of first-order clause coverage through randomization and maximum likelihood. In *ICML 2008: 25th International Conference on Machine Learning*, 2008.
- [25] Ondřej Kuželka and Filip Železný. Hifi: Tractable propositionalization through hierarchical feature construction. In Filip Železný and Nada Lavrač, editors, *Late Breaking Papers, the 18th International Conference on Inductive Logic Programming*, 2008.
- [26] Ondřej Kuželka and Filip Železný. A restarted strategy for efficient subsumption testing. *Fundamenta Informaticae*, 89(1):95–109, 2008.
- [27] Ondřej Kuželka and Filip Železný. Block-wise construction of acyclic relational features with monotone irreducibility and relevancy properties. In *ICML 2009: The 26th International Conference on Machine Learning*, 2009.
- [28] N. Lavrač, D. Gamberger, and V. Jovanoski. A study of relevance for learning in deductive databases. *Journal of Logic Programming*, 40(2/3):215–249, August/September 1999.
- [29] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.
- [30] J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.
- [31] S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
- [32] Shan-Hwei Nienhuys-Cheng and Ronald de Wolf. *Foundations of Inductive Logic Programming (Lecture Notes in Computer Science)*. Springer, 1997.
- [33] Sigfried Nijssen. Data mining using logic. Master’s thesis, Leiden University, 2000.
- [34] G. Plotkin. *A note on inductive generalization*. Edinburgh University Press, 1970.
- [35] Luc De Raedt. Logical settings for concept-learning. *Artif. Intell.*, 95(1):187–201, 1997.
- [36] Liva Ralaivola, Sanjay J. Swamidass, Hiroto Saigo, and Pierre Baldi. Graph kernels for chemical informatics. *Neural Netw.*, 18(8):1093–1110, 2005.

- [37] Walter Rudin. *Real and Complex Analysis*. McGraw-Hill, 1987.
- [38] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.
- [39] Tobias Scheffer and Ralf Herbrich. Unbiased assessment of learning algorithms. In *In IJCAI-97*, pages 798–803, 1997.
- [40] Tobias Scheffer, Ralf Herbrich, and Fritz Wyszotzki. Efficient theta-subsumption based on graph algorithms. In *Inductive Logic Programming Workshop*, pages 212–228, 1996.
- [41] M. Sebag and C. Rouveirol. Tractable induction and classification in first-order logic via stochastic matching. In *IJCAI97*, pages 888–893. MK, 1997.
- [42] Michael Sipser. *Introduction to the Theory of Computation, Second Edition*. Course Technology, 2005.
- [43] A. Srinivasan and S. H. Muggleton. Mutagenesis: Ilp experiments in a non-determinate biological domain. In *Proceedings of the 4th International Workshop on Inductive Logic Programming, volume 237 of GMD-Studien*, pages 217–232, 1994.
- [44] F. Železný and N. Lavrač. Propositionalization-based relational subgroup discovery with RSD. *Machine Learning*, 62(1-2):33–63, 2006.
- [45] Filip Železný. A bottom set strategy for tractable feature construction. In *Work-in-Progress Track of the 14th Int. Conf. on Inductive Logic Programming*, 2004.
- [46] Filip Železný. Tractable construction of relational features. In *Znalosti 05, Bratislava*, 2005.
- [47] Toby Walsh. Search in a small world. In *IJCAI '99: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
- [48] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, San Francisco, 2nd edition, 2005.
- [49] Huayu Wu. Randomization and restart strategies. Master's thesis, University of Waterloo, 2006.

- [50] M. Yannakis. Algorithms for acyclic database schemes. In *International Conference on Very Large Data Bases (VLDB '81)*, pages 82–94, 1981.
- [51] Monika Žáková, Filip Železný, Javier Garcia-Sedano, Cyril Masia Tissot, Nada Lavrač, Petr Křemen, and Javier Molina. Relational data mining applied to virtual engineering of product designs. In *ILP06*, volume 4455 of *LNAI*, pages 439–453. Springer, 2007.
- [52] Filip Železný. Efficient construction of relational features. In *Proceedings of the 4th Int. Conf. on Machine Learning and Applications*, pages 259–264. IEEE Computer Society Press, 2005.
- [53] Filip Železný, Ashwin Srinivasan, and C. David Page. Lattice-search runtime distributions may be heavy-tailed. In *Proceedings of the 12th International Conference on Inductive Logic Programming*. Springer, 2002.

# Appendix A

## Algorithmic Details

### A.1 Generators of Random $\theta$ -subsumption problems

In this section, we describe generators of random data used in experiments in Chapter 3 to measure runtime distributions of  $\theta$ -subsumption algorithms. The first algorithm (Algorithm 15) is a variant of random graphs model by Erdos and Rényi [4]. This random graph model is parametrized by two parameters: number of vertices  $n$  and probability that a pair of vertices is by an edge of random orientation (this parameter is denoted as the *connectivity parameter*). A minor distraction from the directed version of the original model of Erdos and Rényi is that there may be loops (i.e. cycles of length 2) in their model whereas there are no loops in the model used in this thesis. The second algorithm (Algorithm 16) creates so-called *scale-free* graphs<sup>1</sup>, which were developed to approximate some of the properties of real world networks (social, biological, etc.) such as power-law distribution of vertex degrees.

After a random graph is generated, each of its vertices is colored randomly either *red* or *black*. A  $\theta$ -subsumption problem is then constructed as follows. We generate two random graphs  $G_C$  and  $G_e$  with parameters  $(n_C, p_C, n_e, p_e)$ . Then we create a first-order representation of these graphs giving rise to clauses  $C'$  and  $e$  writing  $edge(v_1, v_2)$  for vertices connected by a directed edge pointing from  $v_1$  to  $v_2$  and  $red(v_1)$  ( $black(v_1)$ ) for vertices colored red (black, respectively). Finally, we variabilize clause  $C'$  obtaining clause

---

<sup>1</sup>These graphs are sometimes denoted as *small-world graphs*, but the small world property, i.e. short distance between nearly any two vertices is also inherent to Erdos-Rényi graphs with sufficient connectivity parameter.

---

**Algorithm 15** *RandomGraph*( $n, p$ ): A generator of uniform random graphs

---

- 1: **Input:** Integer  $n$ , Real  $p$ ;
  - 2: Let  $V$  be a set of  $n$  vertices and  $G$  an empty edge set.
  - 3: **for**  $\forall \{v_i \in V, v_j \in V | v_i \neq v_j\}$  **do**
  - 4:   With probability  $p$ ,  $G \leftarrow G \cup \{v_i, v_j\}$
  - 5: **end for**
  - 6: For all edges in  $G$  choose a random orientation, and for all vertices in  $V$  choose a random color with uniform probability from  $\{red, black\}$ .
  - 7: **return** graph with vertex set  $V$  and edge set  $G$
- 

$C$ . The corresponding  $\theta$ -subsumption problem is then to check whether  $C \preceq_{\theta} e$ . In some cases, however, creating pairs hypothesis-example is not sufficient (e.g. in evaluation of algorithm RECOVER and we need to have a fixed set of examples. In such case, we simply generate one hypothesis  $C$  and  $m$  examples and the respective  $\theta$ -subsumption problem is then simply to compute the extension of  $C$  w.r.t. the  $m$  examples. It is also possible to make the set of examples fixed. However, in this case the statistical parameters of the experiments are different than in the other case.

---

**Algorithm 16** *ScaleFreeGraph*( $n, k$ ): A generator of scale-free random graphs

---

- 1: **Input:** Integers  $n, k$ ;
  - 2: Let  $V$  be a set containing one vertex  $v_1$ ,  $G$  be an empty edge set.
  - 3: **for**  $i \leftarrow 2$  to  $n$  **do**
  - 4:    $k' \leftarrow \min(i - 1, k)$
  - 5:   Create vertex  $v_i$
  - 6:   Connect  $v_i$  to  $k'$  distinct vertices  $v_1, \dots, v_k$  chosen from the set  $V$  with probability proportional to their degrees
  - 7:    $G \leftarrow G \cup \{(v_i, v_j) | j = 1 \dots k\}$
  - 8: **end for**
  - 9: For all vertices in  $V$  choose a random color with uniform probability from  $\{red, black\}$ .
  - 10: **return** graph with with vertex set  $V$  and edge set  $G$
-

## A.2 Canonical ordering $\prec_c$

Definition 4.8 introduced properties every canonical ordering  $\prec_c$  on (pos) features should have, but we did not describe any such ordering. Such an ordering is given by the following rules ( $F_1^+, F_2^+, F_1^+ \neq F_2^+$  are (pos) features):

- If  $|F_1^+| < |F_2^+|$ , then  $F_1^+ \prec_c F_2^+$
- If  $|F_1^+| = |F_2^+|$  and  $\text{predicate}(\text{root}(F_1^+)) \prec_L \text{predicate}(\text{root}(F_2^+))$ , where  $\prec_L$  denotes lexicographic ordering of strings, then  $F_1^+ \prec_c F_2^+$ .
- If  $|F_1^+| = |F_2^+|$  and  $\text{predicate}(\text{root}(F_1^+)) = \text{predicate}(\text{root}(F_2^+))$ , then let  $\text{Sub}(F_1^+)$  and  $\text{Sub}(F_2^+)$  be sets containing direct subfeatures, where by direct subfeatures of a (pos) feature  $F$  we mean pos features, whose roots *consume* output variables of  $F$ , of  $F_1^+$  and  $F_2^+$  ordered using  $\prec_c$ . Let  $i$  be the index of the first direct subfeatures such that  $F_{1i}^+ \neq F_{2i}^+$  ( $F_{1i}^+ \in \text{Sub}(F_1^+)$  and  $F_{2i}^+ \in \text{Sub}(F_2^+)$ ). Then  $F_1^+ \prec_c F_2^+$  if and only if  $F_{1i}^+ \prec_c F_{2i}^+$ .

---

**Algorithm 17** *HiFi*: Given a template and a set of examples, *HiFi* computes the propositionalized table.

---

```

1: Input: Template  $\tau$ , Integer  $n$  (maximum feature size), Set examples;

2:  $PosFeatures \leftarrow []$  /* PosFeatures is an associative array of sets, whose elements
   are ordered according to canonical ordering  $\prec_c$  */

3:  $OrderedDefs \leftarrow$  topologically sorted predicate definitions computed from  $\tau$ 

4: for  $\forall d \in OrderedDefs$  do
5:    $predicate \leftarrow d.predicate$ 
6:   for  $\forall example \in examples$  do
7:      $partiallyGroundedLits \leftarrow \{predicate(-, -, \dots, -)\}$ 
8:     for  $i = 1 \dots arity(predicate)$  do
9:       if  $d.modes[i] = OUTPUT$  then
10:         $Combinations \leftarrow BuildCombinations(n, PosFeatures[d.types], example)$ 
11:         $partiallyGroundedLiterals' \leftarrow \{\}$ 
12:        for  $\forall pgLiteral \in partiallyGroundedLits$  (*) do
13:          for  $\forall c \in Combinations$  do
14:             $newLiteral \leftarrow$  copy of  $pgLiteral$  with  $i$ -th argument set to  $c$ 
15:            if  $CheckSize(newLiteral) = false$  then
16:              goto (*)
17:            end if
18:            if  $CheckSubsumption(newLiteral, example) = true$  then
19:               $partiallyGroundedLits' \leftarrow partiallyGroundedLits' \cup \{newLit\}$  f
20:              Record domain of  $newLit$  w.r.t.  $example$ 
21:            end if
22:          end for
23:        end for
24:         $partiallyGroundedLits \leftarrow partiallyGroundedLits'$ 
25:      end if
26:    end for
27:    Filter redundant pos features from  $PartiallyGroundedLits$ 
28:     $PosFeatures[d.inputType] \leftarrow PosFeatures[d.inputType] \cup$ 
       $partiallyGroundedLits$ 
29:  end for
30: end for
31: return  $\{s \mid s \text{ has no inputs}\}$ 

```

---

---

**Algorithm 18** *BuildCombinations*: Given a set of already generated pos features, an example and a limit on feature length, function *BuildCombinations* creates set of all non-reducible pos features covering the example.

---

```

1: Input: Integer maxSize, Ordered set PosFeatures containing pos features with
   given inputType, Example example;

2: PosFeatures  $\leftarrow \{s \mid s \in \text{PosFeatures} \text{ and } s \preceq_{\theta} \text{example}\}$ 
3: Combinations1  $\leftarrow \{\}$ 
4: for  $\forall s \in \text{PosFeatures}$  do
5:   if checkSize(s, maxSize) then
6:     Combinations1  $\leftarrow \text{Combinations}_1 \cup \{s\}$ 
7:   end if
8: end for
9: n  $\leftarrow 1$ 
10: repeat
11:   Combinationsn+1  $\leftarrow \{\}$ 
12:   for  $\forall c \in \text{Combinations}_n$  (*) do
13:     for  $\forall s \in \{s \in \text{PosFeatures} \mid s \text{ is in } \text{PosFeatures} \text{ after last element of } c\}$  do
14:       newComb  $\leftarrow c \cup \{s\}$ 
15:       if checkSize(c, maxSize) then
16:         goto (*)
17:       end if
18:       if checkRedundancy(c, s) = true and
         checkSubsumption(newComb, example) = true then
19:         Combinationsn+1  $\leftarrow \text{Combinations}_{n+1} \cup \{\text{newComb}\}$ 
20:       end if
21:     end for
22:   end for
23:   n  $\leftarrow n + 1$ 
24: until Combinationsn+1 =  $\{\}$ 
25: return  $\cup_{i=1}^n \text{Combinations}_i$ 

```

---



# Appendix B

## List of Software Used

- Java J2SE 6.0 and Netbeans 6.5.1
- Matlab R2007a
- Typesetting system  $\text{\LaTeX}$
- WEKA (Waikato Environment for Knowledge Analysis)

# Appendix C

## The Enclosed CD Contents

- Directory **Thesis** contains this thesis in PDF.
- Directory **Source** contains source codes in Java for the algorithms presented in this thesis.