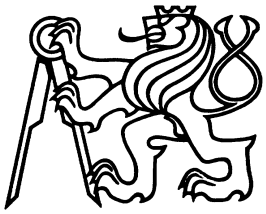
CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering

MASTER'S THESIS

2014 Andrea Fuksová

CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Electrical Engineering
Department of Computer Science



MASTER'S THESIS

Fast Relational Learning Using Bounded LGG

Efektivní relační strojové učení s využitím omezeného nejmenšího společného zobecnění

Student: Andrea Fuksová
Specialization: Artificial Intelligence
Program: Open Informatics
Advisor: Ing. Ondřej Kuželka Ph.D.
Academic year: 2013/2014

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with ethical principles for writing academic thesis.

In Prague 12th May 2014 .....

Andrea Fuksová

Acknowledgement

I would like to thank to my supervisor Ondřej Kuželka for his worthwhile advices, patient guidance and devoted time. I would like to thank to Filip Železný for his advices and support. I would like to thank to my colleagues from IDA Radomír Černoch, Karel Jalovec and Gustav Šourek for help with implementation problems.

Access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005), is greatly appreciated.

I was supported by the Czech Science Foundation project P202/12/2032.

Andrea Fuksová

Title: **Fast relational learning using bounded LGG**

Author: Andrea Fuksová
Program: Open Informatics
Specialization: Artificial Intelligence

Type of thesis: Master's Thesis
Advisor: Ing. Ondřej Kuželka Ph.D., KU Leuven
Guarantor: doc. Ing. Filip Železný, Ph.D.

Abstract: In this thesis we explore specific bottom-up learning procedures for classifying data in the form of relational structures. The key idea is to find a discriminative set of a few specific structures, which are homomorphic to as many positive examples and as few negative examples as possible. Our learning approach is based on finding Plotkin's least general generalization which we reformulated in terms of relational structures. The least general generalization is used in specific bounded versions exploiting local consistency techniques from the constraint satisfaction programming. We also provide an equivalent formulation of our approach in terms of the first order logic.

Key words: Relational machine learning, least general generalization, constraint satisfaction

Název: **Efektivní relační strojové učení s využitím omezeného nejmenšího společného zobecnění**

Abstrakt:
Úkolem této práce je prozkoumat specifické algoritmy relačního strojového učení pro klasifikaci dat ve formě relačních struktur. Základem je nalezení množiny specifických struktur, které by odlišovaly negativní a pozitivní trénovací příklady. Tyto struktury by měly být homomorfní k co největšímu počtu pozitivních příkladů a k co nejmenšímu počtu negativních příkladů. Naše metody jsou založeny na hledání Plotkinova nejmenšího společného zobecnění. Tuto operaci jsme přeformulovali do terminologie relačních struktur. Nejmenší společné zobecnění používáme ve specifické omezené verzi, která je založená na technikách z CSP. Zároveň uvádníme ekvivalentní formulaci používaných metod v terminologii logiky prvního řádu. *Autor:* Andrea Fuksová
Klíčová slova: Relační strojové učení, nejmenší společné zobecnění, CSP

Contents

Introduction

Machine learning is a subfield of artificial intelligence which aims to learn a hypothesis from a big number of examples. There are many domains and many types of data which can be explored with many different techniques. Most approaches are based on learning from data represented as vectors of numbers. However some data can not be represented in this form and require to be encoded in a structured way because the vector representation does not fully express their properties. Such problems arise very often for example in biology. For such structured representation, for example graphs, first order logic clauses or relational structures can be used. Relational machine learning is a part of machine learning concerned with structured data.

One of the basic tasks of relational machine learning is to find a set of structures (for example first order logic clauses) which can be used as rules. They should be contained in as many positive examples as possible and as few negative examples as possible. The score of one structure is then determined by the examples in which it is contained. For instance the dataset can contain molecules with bacteria-killing ability as positive examples and molecules without this property as negative examples. In this case we try to find substructures which are typical for the bacteria-killing molecules.

Most of the previous approaches in this field related to our work are based on theoretical background from inductive logic programing (ILP) [8]. The decision whether a clause is contained in another clause is then often formulated as decision whether there exists so-called θ-subsumption between the clauses.

There are two basic approaches to induction of relational structures in relational machine learning. First approach is represented by so-called top-down methods and the second one by so-called bottom-up methods.

The top-down methods are based on specialization. They begin with a small structure which is very general and try to specialize it by extending it. The main disadvantage of this approach is the so-called plateau effect [1]. It means that the score of a structure does not have to be changed when the clause is specialized. Some of the most popular tools based on top-down methods are Aleph [25] and FOIL [12]. More recent systems nFoil and kFOIL combine FOIL's structure search with naive Bayes, respectively SVM learning algorithms [13].

As opposed to the top-down methods the bottom-up methods are based on generalization of examples. The score of a structure is then changed by every other new example included in the generalization process. An example of such approach can be found in [15]. An example of a state-of-the-art bottom-up learner is Progolem [16] which is based on theory described in [26]. Another learner based on bottom-up methods is described in [2].

A basic operation which can be used for generalization of learning examples is least general generalization (LGG) defined by Plotkin [20]. An LGG of a set of structures computed by Plotkin's method can be very large (exponential in the number of structures). Therefore it is necessary to use a suitable reduction. From the theoretical point of view the so-called θ-reduction is the best choice because it preserves all properties of LGG.

θ-subsumption, which is normally used to score candidate structures, is an NP-complete problem and θ-reduction, which is used to reduce the size of LGGs, is a co-NP-complete problem. Therefore the so-called bounded subsumption and bounded LGG were introduced in [9] and in [10]. θ-subsumption can be straightforwardly formulated as a constraint satisfaction problem (CSP) [14]. Algorithms for solving CSP have exponential runtime in general. The bounded approaches are based on polynomial local consistency methods from CSP.

One of the goals of this thesis is to explore some of the possible bounded subsumption and bounded LGG methods and compare their performance and also to evaluate their performance compared to basic θ-subsumption and θ-reduction. For this purpose it was necessary to implement our own CSP solver because the existing solvers are designated for a straightforward use and do not allow to modify an existing formulation of CSP which is crucial in our implementation of reduction algorithms.

Most techniques from ILP have their equivalent counterparts in the field of relational structures. Another main goal of the thesis is to formulate the theory based on ILP into relational structures. Such a formulation is equivalent to ILP and is more accessible for most scientific audience.

Chapter 1

Theory

In this chapter we provide definition of basic terms and some propositions important for the presented work. Our learner deals with structured data. Such structures can be treated as FOL clauses or relational structures. In future sections we describe both approaches and provide correspondence between them since those two formulations are equivalent when we restrict ourselves to non-recursive function-free clauses.

First we start with a section concerned with relational structures. In this case we treat our learning data as a set of relational structures. We have a set of examples such that every example is one relational structure. Every example has also a label indicating whether the example is positive or negative. Positive examples can be for example molecules with a specific function and negative examples molecules without this function. If we assume that this specific function is determined by a specific substructure which is common in the positive examples and not in the negative examples, we would like to find those substructures and obtain in this way a universal rule which tells us how to distinguish the positive and the negative examples. We assume that this specific function does not have to be caused by only one characteristic substructure but that there might exist more substructures which can cause this function.

Our learner is based on the following idea. We try to find a set of substructures such that the set can be used as a classifier. If an examples contains at least one of the substructures from our learned set, we classify it as positive. If an example does not contain any of the learned substructures we classify it as negative. The property "to be a substructure of another structure" is understood in the sense of homomorphism of relational structures defined in Section 1.1.1. Our set of learned structures is then used as follows: If at least one of the learned relational structures is homomorphic to an example, we classify this example as positive, otherwise we classify it as negative.

Our methods are based on the bottom-up approach as mentioned in Introduction. It means that we generalize the positive examples to find their common property, i.e. a common substructure. For this purpose we use least general generalization (LGG) defined in Section 1.1.3. The result of such generalization is a relational structure which is homomorphic to all structures which were included in the generalization.

Deciding whether a relational structure is homomorphic to another structure can be tested by solving a CSP. This is described in Section 1.1.2.

To avoid some exponential-time procedures like solving CSP or reduction of LGG

without loosing its "least generality" so-called bounded operations are introduced in Sections 1.1.4 and 1.1.6. Theses bounded operations are based on local consistency techniques from CSP. For this purpose terms like generalized arc consistency or treewidth of relational structure are introduced in Section 1.1.

In Section 1.2 we provide equivalent theoretical formulation in terms of first order logic. This time we treat learning examples as FOL Horn clauses and we use θ-subsumption instead of homomorphism. Corresponding bounded operations are also introduced.

1.1 Relational structures

1.1.1 Introduction to relational structures

In this thesis we will describe almost all methods in terms of relational structures. Therefore in this section we provide some basic definitions of these structures and introduce some useful mathematical operations on them. The definitions were taken from [3]. The formulation in terms of relational structures can be easily reduced to a formulation where relational structures are replaced by first-order logic clauses and homomorphism is replaced by so-called θ-subsumption. We will show this conversion in chapter 1.2.

Definition 1.1. *Vocabulary* σ is a finite set of relation symbols or predicates. Every relation symbol in a vocabulary has an arity associated to it.

Definition 1.2. *Relational structure* $\mathbb{A}$ of type σ is a pair consisting of a set $\mathcal{U}_A$, called the *universe* of $\mathbb{A}$, and a sequence of relations $\mathcal{R}_A$. There exists one relation $R^A \in \mathcal{R}_A$ for each relation symbol $R \in \sigma$ and this relation has the same arity as R.

Definition 1.3. A structure $\mathbb{B}$ is called an *induced substructure* of a structure $\mathbb{A}$ of type σ, if the universe $\mathcal{U}_B \subset \mathcal{U}_A$ and for all $R \in \sigma$ it holds $R^B = R^A \cap \mathcal{U}_B^m$, where m is the arity of R.

Definition 1.4. A structure $\mathbb{B}$ of type is called an *substructure* of a structure $\mathbb{A}$ of type σ, if the universe $\mathcal{U}_B \subset \mathcal{U}_A$ and for all $R \in \sigma$ it holds $R^B \subset R^A \cap \mathcal{U}_B^m$, where m is the arity of R.

In some future examples we will use a simplified notation to work with the relational structures. We will use the notation $R_i^A(a_1, \ldots, a_r)$ inspired by Prolog to emphasize that the tuple $(a_1, \ldots, a_r)$ belongs to the set of the relation R_i^A.

Definition 1.5. *Gaifman graph* of a relational structure $\mathbb{A} = (\mathcal{U}_A; R^1, \ldots, R^n)$ is the graph with vertex set $\mathcal{U}_A$, where (a, b) is an edge if and only if $a \neq b$ and there is a relation $R^i \in \mathcal{R}_A$ such that a and b belong to the same tuple of the relation R^i.

Definition 1.6. A *tree decomposition* of a graph $G = (V, E)$ is a labeled tree T such that:

1. Every node of T is labeled by a non-empty subset of V.
2. For every edge $(v, w) \in E$, there is a node of T whose label contains $\{v, w\}$.
3. For every $v \in V$, the set of nodes of T, whose labels contain v, is a subtree of T.

The *width* of a tree decomposition T is the maximum cardinality of a label in T minus 1. The *treewidth of a graph* G is the smallest number k such that G has a tree decomposition of width k. The *treewidth of a relational structure* is the treewidth of its Gaifman graph.

Note that the treewidth of a tree (containing at least one edge) is one.

Definition 1.7. A *homomorphism* from a structure $\mathbb{A}$ to a structure $\mathbb{B}$ of the same type is a mapping $f : \mathcal{U}_A \to \mathcal{U}_B$ such that for every $m-$ary $R \in \sigma$ and every $(a_1, \ldots, a_m) \in R^A$ we have $(f(a_1), \ldots, f(a_m)) \in R^B$. If this homomorphism exists, we say that $\mathbb{A}$ is *homomorphic* to $\mathbb{B}$ and denote it by $\mathbb{A} \to \mathbb{B}$. If a homomorphism does not exist we write $\mathbb{A} \nrightarrow \mathbb{B}$. If $\mathbb{A} \to \mathbb{B}$ and $\mathbb{B} \to \mathbb{A}$ we say that $\mathbb{A}$ and $\mathbb{B}$ are *homomorphically equivalent* (denoted by $\mathbb{A} \approx \mathbb{B}$).

It is obvious, that if a structure $\mathbb{A}$ is an induced substructure of $\mathbb{B}$, then $\mathbb{A}$ is homomorphic to $\mathbb{B}$.

Definition 1.8. An *endomorphism* h of a relational structure $\mathbb{A}$ is a homomorphism from $\mathbb{A}$ to itself. An endomorphism h is said to be an *automorphism* if it is bijective.

Definition 1.9. A structure is a *core* if every its endomorphism is an automorphism. A *core* of a relational structure $\mathbb{A}$ is an induced substructure $\mathbb{B}$ such that $\mathbb{A} \to \mathbb{B}$ and $\mathbb{B}$ is a core.

All cores of a structure are isomorphic (i.e. there exists bijective homomorphism between them) and we will denote any of the cores of a relational structure $\mathbb{A}$ as $\texttt{core}(\mathbb{A})$. It holds that a structure $\mathbb{A}$ and its core are homomorphically equivalent. In many problems it would be useful to work only with cores instead of structures. Unfortunately deciding whether a structure is a core is co-NP-complete task.

Deciding homomorphism between two structures can be very easily formulated as a task of constraint satisfaction programming (CSP). In the next section we provide basic overview of CSP problems.

1.1.2 Constraint satisfaction problem

The constraint satisfaction problem is a well known task appearing in many applications. The problem of deciding homomorphism between two relational structures can be easily formulated as a constraint satisfaction problem. In later sections we also show that it can be used for deciding about so called theta-subsumption between two logical clauses.

In this section we provide some basic definitions from the domain of the constraint satisfaction programming and an overview of some filtering techniques. These definitions can be found in [22] or in short version in [4].

Definition 1.10. A *Constraint Satisfaction Problem (CSP)* is a tuple (X, D, C) where $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{Dom(x_1), \ldots, Dom(x_n)\}$ is a set of ordered finite domains, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c_i is a pair $(var(c_i), rel(c_i))$, where $var(c_i) = (x_{i_1}, \ldots, x_{i_k})$ is an ordered subset of X, and $rel(c_i)$ contains the allowed combinations of values for the variables in $var(c_i)$. Each tuple $\tau \in R^{c_i}$ is an ordered list of values $(a_{i_1}, \ldots, a_{i_k})$. A tuple $\tau \in rel(c_i)$ is *valid* iff all the values in the tuple are present in the domain of the corresponding variables.

The assignment of a value a to a variable x_i is denoted by (x_i, a). Any tuple $\tau = (a_1, \ldots, a_k)$ can be viewed as a set of value to variable assignments $\{(x_1, a_1), \ldots, (x_n, a_k)\}$.

The ordered set of variables over which a tuple τ is defined is $var(\tau)$. For any $Z \subset var(\tau)$ we denote $\tau[Z]$ the subtuple of τ that includes only assignments to the variables in Z.

Using CSP in relational structures

In case that we want to find out whether $\mathbb{A} \to \mathbb{B}$, we formulate the CSP task as follows:

- The elements of $\mathcal{U}_A$ are variables.
- The elements of $\mathcal{U}_B$ are values the variables can take.
- Every relation $R^A \in \mathcal{R}_A$ determines a constraint.
- The tuples contained in the relation R^B determine the valid tuples for relation R^A.

In this case of the CSP task we try to find a mapping $f : \mathcal{U}_A \to \mathcal{U}_B$ so that for every $R^A \in A$ and for every tuples $(x_1, \ldots, x_r) \in R^A$ it holds $(f(x_1), \ldots, f(x_r)) \in R^B$.

We will call this CSP formulation of the task as standard in future text.

Dual formulation of CSP

The above formulation of a CSP for deciding homomorphism is not the only possibility. In some cases it can be useful to encode the task into CSP with only binary constraints. We define the problem of deciding whether $\mathbb{A} \to \mathbb{B}$ as follows:

- There exists exactly one CSP variable for every element of $\mathcal{U}_A$.
- Domains of variables of this type are all elements of $\mathcal{U}_B$.
- There is exactly one CSP variable V_τ for every tuple $\tau \in R^A$ for every relation in $R^A \in \mathcal{R}_A$.
- In the domain of such variable V_τ, there are all tuples from R^B.
- For every r-ary relation $R^A_k \in \mathcal{R}_A$ containing m tuples we create $m \times r$ constraints in this way:
  - Let $R^A_k = \{(a_{11}, \ldots, a_{1r}), \ldots, (a_{m1}, \ldots, a_{mr})\}$.
  - Let us denote the variable created from the tuple $\tau = (a_{i1}, \ldots, a_{ir}) \in R^A_k$ as V_τ.
  - Let us denote the CSP variable corresponding to $a_{ij} \in \mathcal{U}_A$ as $V_{a_{ij}}$.
  - The new constraint c is: $var(c) = (V_\tau, V_{a_{ij}})$, $(\gamma, b) \in rel(c)$ for every tuple $\gamma \in R^B_k$ such that b occurs at the j-th position in γ .

Example 1.11. Let us have two relational structures $\mathbb{A} = (\mathcal{U}_A, hasCar, hasLoad)$, $\mathbb{B} = (\mathcal{U}_A, hasCar, hasLoad)$.

- $\mathcal{U}_A = \{a, b, c, d\}$, $hasCar^A = \{(a), (b)\}$, $hasLoad^A = \{(a, c), (b, d)\}$.
- $\mathcal{U}_B = \{k, l, m, n, o\}$, $hasCar^B = \{(j), (k), (l)\}$, $hasLoad^B = \{(k, m), (k, o), (l, n)\}$.

Here we have CSP variables

$$X = \{V_a, V_b, V_c, V_d, V_{hasCar^A(a)}, V_{hasCar^A(b)}, V_{hasLoad^A(a,c)}, V_{hasLoad^A(b,d)}\}$$

. The domains of the variables are:

- $Dom(V_a) = Dom(V_b) = Dom(V_c) = Dom(V_d) = \{j, k, l, m, n, o\}$,
- $Dom(V_{hasCar^A(a)}) = Dom(V_{hasCar^A})(b) = \{hasCar^B(j), hasCar^B(k), hasCar^B(l)\}$,
- $Dom(V_{hasLoad^A(a,c)}) = Dom(V_{hasLoad^A(b,d)})$

  $$= \{hasLoad^B(k,m), hasLoad^B(k,o), hasLoad^B(l,n)\}$$

  .

The constraints are defined this way:

- $var(c_1) = (V_a, V_{hasCar^A(a)})$, $rel(c_1) = \{(j, hasCar^B(j)), (k, hasCar^B(k)), (l, hasCar^B(l))\}$,
- $var(c_2) = (V_b, V_{hasCar^A(b)})$, $rel(c_2) = \{(j, hasCar^B(j)), (k, hasCar^B(k)), (l, hasCar^B(l))\}$,
- $var(c_3) = (V_a, V_{hasLoad^A(a,c)})$, $rel(c_3) = \{(k, hasLoad^B(k,m)), (k, hasLoad^B(k,o)), (l, hasLoad^B(l,n))\}$,
- $var(c_4) = (V_c, V_{hasLoad^A(a,c)})$, $rel(c_4) = \{(m, hasLoad^B(k,m)), (o, hasLoad^B(k,o)), (n, hasLoad^B(l,n))\}$,
- $var(c_5) = (V_b, V_{hasLoad^A(b,d)})$, $rel(c_5) = \{(k, hasLoad^B(k,m)), (k, hasLoad^B(k,o)), (l, hasLoad^B(l,n))\}$,
- $var(c_6) = (V_d, V_{hasLoad^A(b,d)})$, $rel(c_6) = \{(m, hasLoad^B(k,m)), (o, hasLoad^B(k,o)), (n, hasLoad^B(l,n))\}$.

Filtering techniques in CSP

Definition 1.12. An assignment τ is *consistent* iff for all constraints c_i, where $var(c_i) \subseteq var(\tau)$ it holds $\tau[var(c_i)] \in rel(c_i)$.

Definition 1.13. A *solution* to a CSP is a consistent assignment to all variables.

For solving a CSP task a backtracking algorithm can be used. Its basic version can be improved by some constraint propagation techniques. These techniques enable forbidding values or combinations of values for some variables if a given subset of its constraints cannot be satisfied otherwise. A very wide class of techniques are the local consistencies. Local consistencies are used prior to and during search to filter domains and discover inconsistencies early. In case of discovering inconsistency, the algorithm for local consistency returns false, otherwise it returns true.

We now provide a basic backtracking algorithm for solving CSP according to [24]:

1: **function** BACKTRACKING($level$)
2: **if** $notUsedVars$ is empty **then**

```
3:          return true
4:      end if
5:      choose some v ∈ notUsedVars
6:      for all a ∈ Dom(v) do
7:          assign value a to variable v
8:          if Local consistency then
9:              if Backtracking(level + 1) then
10:                 return true
11:             end if
12:             restore(level)
13:         end if
14:     end for
15:     return false
16: end function
```

The function Restore($level$) means restoring all domains of variables to the state before assigning value a to v.

Definition 1.14. A value $a \in Dom(x_i)$ is GAC-supported in a constraint c_j iff there exists $\tau \in rel(c_j)$ such that $\tau[x_i] = a$ and τ is valid. In this case, we say that τ is a GAC-support of a in c_j.

Definition 1.15. A CSP is *Generalized Arc Consistent* (GAC) iff for all $x_i \in X$, $Dom(x_i)$ is non-empty and for all $a \in Dom(x_i)$, a is GAC-supported in each constraint c_j, s.t. $x_i \in var(c_j)$.

When we talk about just arc consistency (AC) we usually assume, that our constraint problem contains only binary constraints. The term GAC is used for general CSPs.

Here we provide one of the versions of algorithm for testing generalized arc consistency. It is called Generalized arc consistency 3 or just GAC3. This algorithm was described in [22]. We provide the algorithm according to [24].

```
1: function GAC3(stack)
2:     while stack is not empty do
3:         choose a variable v from stack
4:         for all constraints c such that v ∈ var(c) do
5:             for all uninstantiated variables u ∈ var(c) do
6:                 if not Revise(v, c) then return false
7:                 end if
8:             end for
9:         end for
10:    end while
11:    return true
12: end function
```

In fact in the original algorithm the step 5. looks this way:
for all uninstantiated variable $u \in var(c) \backslash v$ **do**.
We use the modified version because in the original version domains would not be pruned according to unary constraints. Another option is to prune the domains according to unary constraint prior to the first call of GAC3. A CSP where for all unary constraints $c_i =$

$(\{x\}, rel(c_i))$ it holds $Dom(x) \subset rel(c_i)$ is called node consistent. So generalized arc consistency is for us a combination of node consistency and generalized arc consistency.

```
1: function REVISE(v, c)
2:     changed := false
3:     for all j ∈ Dom(v) do
4:         if not EXISTS(v, c, j) then
5:             remove j from Dom(v)
6:             changed := true
7:         end if
8:     end for
9:     if changed then
10:        push v into stack
11:    end if
12:    if Dom(v) then is empty then
13:        return false
14:    else
15:        return true
16:    end if
17: end function
```

```
1: function EXISTS(v, c, j)
2:     for all τ = (x_1, ..., x_k) ∈ rel(c) such that τ[v] = j do
3:         valid = true
4:         for all i := 1 : k do
5:             y := τ[x_i]
6:             if y ∉ Dom(x_i) then
7:                 valid := false
8:                 break
9:             end if
10:        end for
11:        if valid then
12:            return true
13:        end if
14:    end for
15:    return false
16: end function
```

Definition 1.16. We say that a relational structure $\mathbb{A}$ is *generalized arc consistent* with respect to a structure $\mathbb{B}$ of the same type (denoted by $\mathbb{A} \lhd_{GAC} \mathbb{B}$) if and only if the generalized arc consistency algorithm executed on the CSP representation of problem $\mathbb{A} \to \mathbb{B}$ returns true.

Definition 1.17. A *hypergraph corresponding to a relational structure* $\mathbb{A}$ is a hypergraph G_A constructed as follows:

- For every element in $a \in \mathcal{U}_A$ there is one vertex v_a in G_A.
- For every $R^A \in \mathcal{R}_A$ and every tuple $(a_1, \ldots, a_r) \in R^A$ there is a hyperedge containing the vertices $v_{a_1}, \ldots, v_{a_r}$.

Definition 1.18. A hypergraph G is *acyclic* if the iteration of the following rules on G produces the empty hypergraph:

- Remove a hyperedge contained in another hyperedge.
- Remove a vertex which is contained in at most one hyperedge.

A relational structure is said to be *acyclic* if its corresponding hypergraph is acyclic.

Proposition 1.19. If $\mathbb{A}$ is acyclic and $\mathbb{A} \lhd_{GAC} \mathbb{B}$ then $\mathbb{A} \to \mathbb{B}$.

Many other kinds of local consistencies are described in details in [22] and [4]. The problem is that many types of local consistencies require binary constraints.

Next, we introduce so called singleton consistencies. The details can be found for example in [21].

Definition 1.20. A problem is *singleton arc consistent* (SAC) iff it has non-empty domains and for any instantiation of a variable, the resulting subproblem can be made arc consistent.

Definition 1.21. A problem is *singleton generalized arc consistent* (SGAC) iff it has non-empty domains and for any instantiation of a variable, the resulting problem can be made generalized arc consistent.

The notion of a singleton consistency is general, and can be applied to other levels and types of local consistency than arc consistency. Here we provide the algorithm as stated in [5]

```
1: function SAC(X)
2:     repeat
3:         changed := false
4:         for all v ∈ X do
5:             for all a ∈ Dom(v) do
6:                 remove all values except a from Dom(v)
7:                 if not ARC CONSISTENCY({v}) then
8:                     return all oter values into Dom(v)
9:                     return all values into domains removed during ARC CONSISTENCY({v})
10:                    Dom(v) := Dom(v)\a
11:                    ARC CONSISTENCY({v})
12:                    changed := true
13:                end if
14:            end for
15:        end for
16:    until changed = false
17: end function
```

Definition 1.22. We say that a relational structure $\mathbb{A}$ is *singleton arc consistent* with respect to a structure $\mathbb{B}$ of the same type (denoted by $\mathbb{A} \lhd_{SAC} \mathbb{B}$) if and only if the singleton arc consistency algorithm executed on the CSP representation of problem $\mathbb{A} \to \mathbb{B}$ returns true.

The k-Consistency Test

The k-consistency test is another filtering technique for CSP belonging to the class of local consistency techniques. First we provide some helpful definitions and later we will discuss its role in homomorphism testing in more details.

Definition 1.23. Given the two relational structures $\mathbb{A}$ and $\mathbb{B}$ a *partial homomorphism* is a mapping $f : \mathcal{U}_{A'} \to \mathcal{U}_B$, where $\mathbb{A}' = (\mathcal{U}_{A'}, \mathcal{R}_{A'})$ is a substructure of $\mathbb{A}$. That is f defines a homomorphism from $\mathbb{A}'$ to $\mathbb{B}$.

It means that for every r-ary relational symbol $R^A \in \mathcal{R}_A$ and every tuple $(a_1, \ldots, a_r) \in R^A$ such that $a_1, \ldots, a_r \in \mathcal{U}_{A'}$ it holds $(f(a_1), \ldots, f(a_r)) \in R^B$.

Definition 1.24. If f and g are partial solutions we say that g extends f, denoted by $f \subseteq g$, if $Dom(f) \subseteq Dom(g)$ and $f(a) = g(a)$ for every $a \in Dom(f)$. In this case we also say that f is a *projection* of g to $Dom(f)$.

We now describe the k-consistency algorithm. We provide the version for relational structures because it is more convenient.

1. Given structures $\mathbb{A}$ and $\mathbb{B}$
2. Let H be the collection of all partial solutions f with $|Dom(f)| \leq k + 1$
3. For every f in H with $|Dom(f)| \leq k$ and every $a \in \mathcal{U}_A$, if there is no g in H such that $f \subseteq g$ and $a \in Dom(g)$, remove f and all its extensions from H
4. Repeat step 3 until H is unchanged
5. If H is empty then reject, else accept

Let $\mathcal{A}$ and $\mathcal{B}$ be two classes of relational structures of the same type. We denote by $hom(\mathcal{A}, \mathcal{B})$ the class of problems of deciding homomorphism from $\mathbb{A} \in \mathcal{A}$ to $\mathbb{B} \in \mathcal{B}$. If $\mathcal{A}$ is a class of all finite structures of the same type, we write $hom(*, \mathcal{B})$. Similarly if $\mathcal{B}$ is the class of all finite structures of a certain type, we write $hom(\mathcal{A}, *)$.

In the work [3] the authors state two basic problems:

- The first is called k-width problem. This problem means to characterize all structures $\mathbb{A}$ for which k consistency solves $hom(\mathcal{A}, *)$. These structures are called k-width structures.
- The second is to characterize all structures $\mathbb{B}$ for which the k consistency algorithm solves $hom(*, \mathcal{B})$. These are so called width-k structures.

There it was proved that $\mathbb{A}$ has k-width iff $\texttt{core}(\mathbb{A})$ has treewidth at most k. So if the core of $\mathbb{A}$ has the treewidth at most k, the k consistency algorithm gives us correct answer to homomorphism between $\mathbb{A} \to \mathbb{B}$. There, it is also stated that for every fixed $k \geq 1$ it is an NP-complete problem to decide if a given structure has a core of treewidth at most k. Therefore it is an NP-complete problem to decide if a given structure has k-width.

In the terms of k consistency, we can think about the arc consistency algorithm as 1 consistency algorithm. [1]

Definition 1.25. We say that a structure $\mathbb{A}$ is *k-consistent* w.r.t. a structure $\mathbb{B}$ of the same type (denoted by $\mathbb{A} \lhd_k \mathbb{B}$) if and only of the k consistency algorithm run on structures $\mathbb{A}$ and $\mathbb{B}$ returns true.

Proposition 1.26. If $\mathbb{A}$ has treewidth at most k and $\mathbb{A} \lhd_k \mathbb{B}$ then $\mathbb{A} \to \mathbb{B}$.

1.1.3 Least general generalization

The key idea of our learning algorithm is based on the so called least general generalization of two or more relational structures. We adopted this term from the first order logic as it was defined by Plotkin [20]. The original definition is based on the θ-subsumption for clauses. We instead of that use this term in the sense of homomorphism for relational structures.

Definition 1.27. A relational structure $\mathbb{C}$ is said to be a *least general generalization* of the relational structures $\mathbb{A}$ and $\mathbb{B}$ (denoted by $LGG(\mathbb{A}, \mathbb{B})$) if and only if $\mathbb{C} \to \mathbb{A}$ and $\mathbb{C} \to \mathbb{B}$ and for every other relational structure $\mathbb{D}$ such that $\mathbb{D} \to \mathbb{A}$ and $\mathbb{D} \to \mathbb{B}$ it holds $\mathbb{D} \to \mathbb{C}$.

The LGG for two given structures does not have to be unique. The basic algorithm for finding an LGG of two first order logic clauses can be found for example in [19]. The algorithm for two relational structures we provide here is similar to the definition of the graph product [6].

Let us have two relational structures $\mathbb{A}$ and $\mathbb{B}$ of the same type σ. We construct a relational structure $\mathbb{C}$ such that:

- We start with an empty $\mathcal{U}_C$ and empty sets for all relational symbols. Then we go over all relational symbols $R_k \in \sigma$ to create elements of the universe and tuples in relational sets.
- For every pair of tuples $\tau = (a_1, \ldots, a_r) \in R_k^A$ and $\gamma = (b_1, \ldots, b_r) \in R_k^B$ (tuples with the same relational symbol) we create a new tuple $\rho = (c_1, \ldots, c_r)$ as follows and insert it into the set R_k^C.
- If there is no element in $\mathcal{U}_C$ corresponding to the pair of elements a_i and b_i, we create in $\mathcal{U}_C$ a new element corresponding to this pair. The element c_i in ρ is this new element.
- If there already exists an element d corresponding to the pair a_i and b_i in $\mathcal{U}_C$, we use this element as c_i in ρ.
- If we have already processed all suitable pairs of relational tuples and we have empty $\mathcal{U}_C$, we just add one element into $\mathcal{U}_C$. This could happen if we have some empty relations in the structures.

The structure $\mathbb{C}$ obtained by the previous algorithm is the LGG of $\mathbb{A}$ and $\mathbb{B}$. We now provide a key idea why it should hold. First we construct a homomorphism $\mathbb{C} \to \mathbb{A}$. Every element $c_{ab} \in \mathcal{U}_C$ corresponding to the pair of elements $a \in \mathcal{U}_A$ and $b \in \mathcal{U}_B$ is mapped into

[1] The algorithm we call k consistency is often called strong $(k+1)$ consistency in the CSP literature. Our notation corresponds to the notation of Atserias et al.

the element a. If there is a tuple $\rho \in R_k^C$ it was based on some tuple $\tau \in R_k^A$ containing the respective elements from $\mathcal{U}_A$. Analogically for $\mathbb{C} \to \mathbb{B}$.

Let us consider a relational structure $\mathbb{D}$ such that $\mathbb{D} \to \mathbb{A}$ and $\mathbb{D} \to \mathbb{B}$. Let us denote the homomorphism between $\mathbb{D}$ and $\mathbb{A}$ by f and the homomorphism between $\mathbb{D}$ and $\mathbb{B}$ by g. We now can easily define a mapping $h : \mathcal{U}_D \to \mathcal{U}_C$ in the following way:

- If there is a relation $\delta = (d_1, \ldots, d_r) \in R_k^D$, there have to exist relations $\tau = (f(d_1), \ldots, f(d_r)) \in R_k^A$ and $\gamma = (g(d_1), \ldots, g(d_r)) \in R_k^B$.
- We define h for $d_1, \ldots, d_r$ such that $h(d_i) = c$ where c is an element corresponding to the pair of $f(d_i)$ and $g(d_i)$.
- The elements of $\mathcal{U}_D$ which are not involved in any relation can be mapped into an arbitrary element of $\mathcal{U}_C$.

It is obvious that a structure found by this algorithm can be very large. If we denote by $|R^A|$ the number of tuples in relation R^A and by $|\mathcal{R}_A| = \sum_{i=1}^{m} |R_i^A|$. We can see, that the $|\mathcal{R}_C| = \sum_{i=1}^{m}(|R_i^A| \cdot |R_i^B|) = \mathcal{O}(|\mathcal{R}_A| \cdot |\mathcal{R}_B|)$. And the universe $|\mathcal{U}_C| = \mathcal{O}(|\mathcal{U}_A| \cdot |\mathcal{U}_B|)$.

After applying the above procedure it would be useful to make the resulting structure $\mathbb{C}$ smaller. It is clear that the `core`$(\mathbb{C})$ satisfies the conditions of LGG but can be significantly smaller than $\mathbb{C}$. The conditions for LGG are also satisfied by any structure $\mathbb{D}$ homomorphically equivalent to $\mathbb{C}$. Unfortunately finding such a structure is co-NP-complete. We now provide the basic idea of finding a structure covering a set of relational structures. For now we let alone the problem with size of structures and with finding a suitable reduction.

Definition 1.28. Let $\mathbb{A}_1, \ldots, \mathbb{A}_m$ be relational structures. A structure $\mathbb{C}$ is said to be a *least general generalization* of the structures $\mathbb{A}_1, \ldots, \mathbb{A}_m$ (denoted by $LGG(\mathbb{A}_1, \ldots, \mathbb{A}_m)$) if and only if $\forall i \in \{1, \ldots, m\} : \mathbb{C} \to \mathbb{A}_i$ and for every other structure $\mathbb{D}$ such that $\forall i \in \{1, \ldots, m\} : \mathbb{D} \to \mathbb{A}_i$ it holds $\mathbb{D} \to \mathbb{C}$.

Let us denote the relational structure which is least general generalization of two relational structures $\mathbb{A}$ and $\mathbb{B}$ as $LGG(\mathbb{A}, \mathbb{B})$. We can find a structure covering some set of relational structures by applying this procedure iteratively to the examples. Let us imagine, that we want to find a relational structure homomorphic to all structures in a set $S = \{\mathbb{A}_1, \ldots, \mathbb{A}_m\}$. We will proceed this way:

1. $\mathbb{L} = \mathbb{A}_1$, $i = 1$
2. for $i = 2$ to m do: $\mathbb{L} := LGG(\mathbb{L}, \mathbb{A}_i)$
3. return $\mathbb{L}$

It is obvious that $\mathbb{L}$ is homomorphic to every structure $\mathbb{A}_i \in S$. It can be also shown that $\mathbb{L}$ is $LGG(\mathbb{A}_1, \ldots, \mathbb{A}_m)$.

1.1.4 X-homomorphism

The general homomorphism problem is NP-complete. In this section we introduce other techniques useful to solve our basic problem of finding a structure covering given set of examples. Some of the stated propositions are proved in [9] and [10] for FOL formulation.

Definition 1.29. Let X be a possibly infinite set of relational structures. Let $\mathbb{A}$, $\mathbb{B}$ be relational structures of the same type not necessarily from X. We say, that $\mathbb{A}$ is *x-homomorphic* to $\mathbb{B}$ w.r.t X (denoted by $\mathbb{A} \to_X \mathbb{B}$) if and only if for every structure $\mathbb{C} \in X$, it holds $(\mathbb{C} \to \mathbb{A}) \Rightarrow (\mathbb{C} \to \mathbb{B})$. If $\mathbb{A} \to_X \mathbb{B}$ and $\mathbb{B} \to_X \mathbb{A}$ then $\mathbb{A}$ and $\mathbb{B}$ are called *x-equivalent* w.r.t. X (denoted by $\mathbb{A} \approx_X \mathbb{B}$). We will call the relation $\to_X$ *x-homomorphism* (or *bounded homomorphism*) w.r.t. X and the relation $\approx_X$ *x-equivalence* w.r.t. X.

When it is clear from the context, we can omit the phrase "w.r.t. X" or use just the term bounded homomorphism.

Proposition 1.30. Homomorphism is x-homomorphism w.r.t the set X of all relational structures.

The x-homomorphism is a weaker alternative to the homomorphism. If we properly choose the set X, we can use a polynomial time algorithm to check the x-homomorphism.

Proposition 1.31. Let X be a set of relational structures. Then x-homomorphism w.r.t. X is a transitive and reflexive relation on relational structures and x-equivalence w.r.t. X is an equivalence relation on relational structures.

Definition 1.32. Let X be a set of relational structures. Let $\lhd_X$ be a relation such that for any two relational structures $\mathbb{A}$ and $\mathbb{B}$ it holds: $(\mathbb{A} \to \mathbb{B}) \Rightarrow (\mathbb{A} \lhd_X \mathbb{B})$ and $(\mathbb{A} \lhd_X \mathbb{B}) \Rightarrow (\mathbb{A} \to_X \mathbb{B})$. Then $\lhd_X$ is called *x-prehomomorphism* w.r.t. the set X.

1.1.5 Indroducing variables and constants

In our datasets some important information is often contained not only in the relational symbols and in the relations between elements but also in the names of the elements. For instance if we explore a dataset describing chemical bonds between atoms, it can contain a relation with relational symbol *bond* of arity 5 with a structure: $(atom1, atom2, C, H, single)$. This structure asserts that there is a single bond between two atoms with identifiers $atom1$ and $atom2$, the $atom1$ is carbon and $atom2$ is hydrogen. If we want to find a discriminative pattern in such chemical structures, we need to keep the information about the type of bond and types of atoms. We do not want to allow the tuple $bond(atom3, atom4, C, O, double)$ to be mapped into the tuple $bond(atom1, atom2, C, H, single)$. From the view of relational structures this mapping would be correct but from the chemical point of view this makes no sense.

For this purpose we introduce in our relational structures special unary relational symbols indicating "names" of the elements. So if the original structure from the previous paragraph is of type $\sigma = \{bond\}$, looks this way: $\mathbb{A} = (\mathcal{U}_A, bond^A)$, where $bond^A = \{(atom1, atom2, C, O, double), (atom1, atom3, C, O, double)\}$ we create new structure by extending the vocabulary by unary relations. The new vocabulary is

$$\sigma' = \{\sigma^b, \sigma^n\}, \text{ where } \sigma^b = \sigma \text{ and}$$
$$\sigma^n = \{name_{atom1}, name_{atom2}, name_{atom3}, name_C, name_O, name_{double}\}.$$

These unary relations contain only the elements with corresponding names, for example $name_{atom1} = \{atom1\}$, $name_O = \{O\}$ etc. The following example presents the result of our extension on LGG:

Example 1.33. Let us have the two relational structures $\mathbb{A}$ and $\mathbb{B}$:

- $\mathcal{U}_A = \{atom1, atom2, atom3, H, O, single\}$, $\mathcal{R}_A = \{bond^A\}$,
- $bond^A = \{(atom1, atom2, O, H, single), (atom1, atom3, O, H, single)\}$,
- $\mathcal{U}_B = \{atom4, atom5, atom6, H, O, K, single\}$, $\mathcal{R}_B = \{bond^B\}$,
- $bond^B = \{(atom4, atom5, O, K, single), (atom4, atom6, O, H, single)\}$

The LGG of the two structures is then structure $\mathbb{C}$ with only one relation in $\mathcal{R}_C$:

$$bond^C = \{(atom14, atom25, oo, hk, s), (atom14, atom26, oo, hh, s),$$
$$(atom14, atom35, o, hk, s), (atom14, atom36, oo, hh, s)\}$$

If we want to introduce our special unary relations indicating names, the vocabulary of structures will contain not only the relation symbol *bond* but also the symbols for those unary relations. It is also necessary to extend the universe of both structures by elements which will be contained in the name relations. Now the two structures are as follows:

- $\mathcal{U}_A = \mathcal{U}_B = \{atom1, atom2, atom3, atom4, atom5, atom6, H, O, K, single\}$
- The relations $bond^A$ and $bond^B$ remain unchanged
- Both of them contain new relations $name_O = \{(O)\}$, $name_H = \{(H)\}$, $name_K = \{(K)\}$, $name_{single} = \{(single)\}$, $name_{atom1} = \{(atom1)\}$, $name_{atom2} = \{(atom2)\}$, $name_{atom3} = \{(atom3)\}$, $name_{atom4} = \{(atom4)\}$, $name_{atom5} = \{(atom5)\}$, $name_{atom6} = \{(atom6)\}$.

The resulting structure $\mathbb{C}$ is then:

$$\begin{aligned}
\mathcal{U}_C =& \{atom11, atom22, atom33, atom44, atom55, atom66, hh, oo, kk, ss, atom14,\\
& atom25, atom26, atom35, atom36, hk\}\\
\mathcal{R}_C =& \{bond^C, name^C_{atom1}, name^C_{atom2}, name^C_{atom3}, name^C_{atom4}, name^C_{atom5}, name^C_{atom6}\\
& name^C_O, name^C_H name^C_K, name^C_{single}\}\\
bond^C =& \{(atom14, atom25, oo, hk, s), (atom14, atom26, oo, hh, s), (atom14, atom35, o, hk, s),\\
& (atom14, atom36, oo, hh, s)\}\\
name_O =& \{(oo)\},\\
name_H =& \{hh)\},\\
name_K =& \{(kk)\},\\
name_{single} =& \{(ss)\},\\
name_{atom1} =& \{(atom11)\},\\
name_{atom2} =& \{(atom22)\},\\
name_{atom3} =& \{(atom33)\},\\
name_{atom4} =& \{(atom44)\},\\
name_{atom5} =& \{(atom55)\},\\
name_{atom6} =& \{(atom66)\}.
\end{aligned}$$

Note that for example the element $atom11$ is involved only in its name relation. Some elements like $atom14$ or hk are not involved in any name relation but are involved in *bond* relation and some elements like ss are involved both in name relation and in *bond* relation.

We have to add the name relations into set of relations of every relational structure we currently work with. For every name relation we add exactly one element into the universe and involve this element in the corresponding name relation. In every structure originating as LGG of two such structures there will be always exactly one element in every name relation.

In the subsequent text we will sometimes denote the set of relational symbols resulting from original relations by σ^b and the set of the unary relational symbols resulting from the names of elements by σ^n. If necessary we will denote the set of relations of a structure $\mathbb{A}$ corresponding to symbols from σ^n by $\mathcal{R}^n_A$ and those corresponding to symbols from σ^b by $\mathcal{R}^b_A$.

In one of the following sections we will concern with a restriction on relational structures regarding the relations from σ^n. We will use these restrictions to distinguish some elements of universe in the equivalent way which is used in the first order logic to distinguish variables and constants. We will treat the elements involved in relations from σ^n as constants and other elements as variables. For simplicity we will use the conditions $isVariable(a)$ and $isConstant(a)$ to distinguish them. Sometimes we just say that a satisfies the condition $isVariable$.

Let us have two structures $\mathbb{A}$ and $\mathbb{B}$ and element elements $a \in \mathcal{U}_A$ and $b \in \mathcal{U}_B$ such that $name^A_1 = \{(a)\}$, $name^B_1 = \{(b)\}$ and those elements are not involved in any relation from σ^b. Such elements are not interesting for homomorphism testing $\mathbb{A} \to \mathbb{B}$ because mapping of a is trivial. We can therefore afford to omit such elements during formulation of CSP. In Section 2.1.7 we also explain how to omit all elements satisfying $isConstant$ (i.e. involved in a name relation) during formulation of CSP. Some modifications in constraints based on relations from σ^b have to be done in that case.

1.1.6 Bounded LGG

In Section 1.1.3 we mentioned that we need to somehow reduce the structure resulting from the basic version of the algorithm for finding LGG. Instead of finding a smaller structure homomorphically equivalent to the resulting structure we find a smaller structure x-equivalent to the resulting structure. Some of the stated propositions are proved in [9] and [10] for FOL formulation.

For this purpose we use a special algorithm called Element-elimination algorithm. We provide its pseudo code:

1: **function** ELEMENT-ELIMINATION($\mathbb{A}$)
2: Set $\mathbb{B} := \mathbb{A}$.
3: Select an element $a \in \mathcal{U}_B$ such that $isVariable(a)$ and $\mathbb{B} \lhd_X \mathbb{B}'$. Where $\mathbb{B}' =$REDUCE($\mathbb{B}$,a)
4: **if** no such element a exists **then**
5: **return** $\mathbb{B}$ and finish.
6: **else**
7: Set $\mathbb{B} := \mathbb{B}'$.
8: Go to step 3.

9: **end if**
10: **end function**
1: **function** REDUCE($\mathbb{B}$,a)
2: $\mathbb{B}' := \mathbb{B}$
3: remove all tuples τ such that $a \in \tau \in R_i^{B'}$ from $R_i^{B'}$
4: remove a from $\mathcal{U}_{B'}$
5: **return** $\mathbb{B}'$
6: **end function**

The resulting structure $\mathbb{B}$ is a substructure of the input $\mathbb{A}$. Therefore it holds $\mathbb{B} \to \mathbb{A}$. The x-prehomomorphism implies the x-homomorphism and x-homomorphism is a transitive relation. Therefore it holds that $\mathbb{A} \to_X \mathbb{B}$. There also does not exist an element $a \in \mathcal{U}_B$ such that $isVariable(a)$ which could be removed from $\mathbb{B}$ so that the structure $\mathbb{B}$ would be homomorphic to the resulting structure. Because if such an element exists, it would be removed in the step 3 of the algorithm (thank to the implication $\mathbb{B} \to \mathbb{B}'$ then $\mathbb{B} \lhd_X \mathbb{B}'$).

Note that the input structure of the algorithm has exactly one element in every relation from $\mathcal{R}_A^n$ as we have discussed in Section 1.1.5.

The output structure $\mathbb{B}$ of the Element-elimination algorithm is a core. We justified that there is no element satisfying $isVariable$ which can be removed. We can consider that neither an element satisfying $isConstant$ can be removed. Let us consider an element $a \in \mathcal{U}_B$ satisfying the condition $isConstant$, which means that a is involved in a relation with a relational symbol $R^B \in \mathcal{R}_B^n$. We know that there is exactly one element in every such relation. Therefore $R^B = \{(a)\}$. Let B' be a structure obtained from $\mathbb{B}$ by removing a from $\mathcal{U}_B$ and all tuples containing a from $\mathcal{R}_B$. It holds that $R^{B'} = \emptyset$. Therefore $\mathbb{B} \not\to \mathbb{B}'$ because the element a can not be mapped into any element from $\mathcal{U}'_B$ such that the mapping would be involved in a tuple from $R^{B'}$.

We will further denote the output of ELEMENT ELIMINATION algorithm with an input structure $\mathbb{A}$ and with using some x-prehomomorphism $\texttt{El-Elim}_X(\mathbb{A})$.

Proposition 1.34. Let X be a set of relational structures, $\texttt{core}(\mathbb{A}) \in X$ be a core of relational structure $\mathbb{A}$. Then $\texttt{El-Elim}_X(\mathbb{A}) \approx \texttt{core}(\mathbb{A})$ and $|\texttt{El-Elim}_X(\mathbb{A})| = |\texttt{core}(\mathbb{A})|$.

Proposition 1.35. Let X and Y be sets of relational structures. Then
$\mathbb{B} = \texttt{El-Elim}_X(\texttt{El-Elim}_Y(\mathbb{A}))$ satisfies:

1. $\mathbb{B} \to \mathbb{A}$, $\mathbb{A} \to_{X \cap Y} \mathbb{B}$, where $\to_{X \cap Y}$ is x-homomorphism w.r.t the set $X \cap Y$.
2. $\mathbb{B}$ is a core (not necessarily the core of $\mathbb{A}$).

We already described the construction of least general generalization and the algorithm for reduction of its result. Now we will concern ourselves with the bounded version of LGG.

Definition 1.36. Let X be a set of relational structures. A relational structure $\mathbb{B}$ is said to be a *bounded least general generalization* of structures $\mathbb{A}_1, \ldots, \mathbb{A}_m$ w.r.t the set X (denoted by $LGG_X(\mathbb{A}_1, \ldots, \mathbb{A}_m)$) if and only if $\forall i \in \{1, \ldots, m\} : \mathbb{B} \to \mathbb{A}_i$ and for every structure $\mathbb{C} \in X$ it holds $(\forall i \in \{1, \ldots, m\} : \mathbb{C} \to \mathbb{A}_i) \Rightarrow (\mathbb{C} \to \mathbb{B})$.

Note that the the set X serves only as a reference set to test the validity of the statement $(\forall i \in \{1, \ldots, m\} : \mathbb{C} \to \mathbb{A}_i) \Rightarrow (\mathbb{C} \to \mathbb{B})$ for all $\mathbb{C} \in X$. However neither the structures $\mathbb{A}_1, \ldots, \mathbb{A}_m$ nor the structure $\mathbb{B}$ have to be from the set X.

Let us denote the set of all bounded least general generalizations of structures $\mathbb{A}_1, \ldots, \mathbb{A}_m$ as S_{LGG_X}, the set of their all conventional LGGs as S_{LGG} and finally the set of all structures homomorphic to all $\mathbb{A}_1, \ldots, \mathbb{A}_m$ as $S_{\rightarrow}$. Then following relations hold:

- $S_{LGG} \subseteq S_{LGG_X}$,
- $S_{LGG_X} \subseteq S_{\rightarrow}$.

The bounded LGG of two relational structures can be computed in polynomial time for many practically interesting sets X. The next advantage of this operation is that the resulting structure can be smaller than the core of conventional LGG. Here we provide an algorithm for computing bounded LGG of a set of relational structures.

Proposition 1.37. Let X be a set of relational structures and let $\lhd_X$ be an x-prehomomorphism w.r.t the set X then the structure $\mathbb{B}$ obtained by

$$\mathbb{B} = \texttt{El-Elim}_X(LGG(\mathbb{A}_n, \texttt{El-Elim}_X(LGG(\mathbb{A}_{n-1}, \texttt{El-Elim}_X(LGG(\mathbb{A}_{n-2}, \ldots)))))) .$$

is a bounded least general generalization of clauses $\mathbb{A}_1, \ldots, \mathbb{A}_n$ w.r.t. the set X.

1.1.7 Connected components

We often need to work with connected components of our structures in our algorithms. For this purpose we use following definition:

Definition 1.38. We say that a relational structure $\mathbb{A}$ which satisfies $\exists R \in \mathcal{R}^b_A : R \neq \emptyset$ is *connected* if there does not exist a decomposition $\mathbb{B}_1, \ldots, \mathbb{B}_n$, where $n > 1$ such that:

1. $\forall i \in \{1, \ldots, n\}$: $\mathbb{B}_i$ is a substructure of $\mathbb{A}$ and $\mathcal{U}_{B_i} \neq \emptyset$,
2. $\forall i \in \{1, \ldots, n\} : \exists R \in \mathcal{R}^b_{B_i} : R \neq \emptyset$
3. $\forall R \in \sigma^b, \forall i, j \in \{1, \ldots, n\}, i \neq j : R^{B_i} \cap R^{B_j} = \emptyset$,
4. $\forall R \in \sigma^b : \bigcup_{i \in \{1,\ldots,n\}} R^{B_i} = R^A$,
5. $\bigcup_{i \in \{1,\ldots,n\}} \mathcal{U}_{B_i} = \mathcal{U}_A$,
6. $\forall a \in \mathcal{U}_A : (a \in \bigcap_{i \in \{1,\ldots,n\}} \mathcal{U}_{B_i}) \Leftrightarrow (\exists R \in \sigma^n$ such that $R^A = \{(a)\})$,
7. $\forall a \in \mathcal{U}_A : (\exists R \in \sigma^n$, such that $R^A = \{(a)\}) \Rightarrow (\forall i \in \{1, \ldots, n\} : R^{B_i} = \{(a)\})$.

If such decomposition exists and every $\mathbb{B}_i$ is connected, we call $\mathbb{B}_1, \ldots, \mathbb{B}_n$ *connected components* of $\mathbb{A}$.

Let us explain the definition in an informal way. We basically want to decompose the structure in terms of relational tuples. We try to group together those of tuples (from relations from σ^b) which have common elements satisfying $isVariable$. We require the groups to do not overlap and to cover all tuples from all relations from $\mathcal{R}^b_A$ (points 3. and 4.). If such groups can be created, we compose new relational structures from them. It is necessary to keep the information about name relations in the smaller structures too, but they are

not considered during grouping, they are added into the substructures automatically with elements involved in them (point 6. a 7.).

According to this definition a structure containing in universe some elements satisfying $isVariable$ which are not involved in any relation would be classified as connected. It does not matter. We do not care about such elements, because they neither are contained in our original examples not are created during LGG or during any operation we use. In addition would not have an influence on homomorphism. The condition $\exists R \in \mathcal{R}^b : R \neq \emptyset$ is important. Without this condition every structure could be decomposed into arbitrary number of structures with empty relations from from σ^b.

1.1.8 Local consistencies and x-homomorphism

So far we have not mentioned any concrete set X practically useful for testing x-homomorphism and for finding bounded LGG. In this section we describe a few such sets. Specifically we will concern with acyclic relational structures and structures with bounded treewidth. We will also provide their relations to some filtering procedures from CSP, because some of these procedures can be used as x-prehomomorphism. For this purpose the following proposition will be important.

Proposition 1.39. Let X be a set of relational structures. A relation $\lhd_X$ is an x-prehomomorphism w.r.t. X if and only if it satisfies the following conditions:

1. If $\mathbb{A} \to \mathbb{B}$ then $\mathbb{A} \lhd_X \mathbb{B}$.
2. If $\mathbb{A} \in X$ and $\mathbb{A} \lhd_X \mathbb{B}$ then $\mathbb{A} \to \mathbb{B}$.
3. If $\mathbb{A} \in X$, $\mathbb{A} \lhd_X \mathbb{B}$ and $\mathbb{B} \lhd_X \mathbb{C}$ then $\mathbb{A} \lhd_X \mathbb{C}$.

From this proposition it follows that if we have an algorithm which checks the homomorphism of the type $\mathbb{A} \to \mathbb{B}$ for all $\mathbb{A} \in X$ and satisfies the conditions 1 and 3 from the proposition, then this algorithm can be used as a procedure which checks x-prehomomorphism with respect to the set X.

For instance, from the proposition 1.19 it follows that for an acyclic structure $\mathbb{A}$ it holds $(\mathbb{A} \lhd_{GAC} \mathbb{B}) \Rightarrow (\mathbb{A} \to \mathbb{B})$. In [10] the following property is proved: If $\mathbb{A}$ is acyclic, $\mathbb{A} \lhd_{GAC} \mathbb{B}$ and $\mathbb{B} \lhd_{GAC} \mathbb{C}$ then $\mathbb{A} \lhd_{GAC} \mathbb{C}$. From this knowledge and from the proposition 1.39 it follows:

Proposition 1.40. The relation $\lhd_{GAC}$ between relational structures is an x-prehomomorphism w.r.t. the set of all acyclic relational structures. Therefore the GAC algorithm can be used to check x-subsumption w.r.t the set of acyclic relational structures.

In Section 1.1.2 we stated the k-Consistency algorithm and defined the relation between relational structures $\mathbb{A} \lhd_k \mathbb{B}$ which means that $\mathbb{A}$ is k consistent w.r.t $\mathbb{B}$. We also stated that this algorithm decides homomorphism $\mathbb{A} \to \mathbb{B}$ for the structures $\mathbb{A}$ with treewidth at most k. Similarly as for $\lhd_{GAC}$ it can be proved:

Proposition 1.41. The relation $\lhd_k$ between relational structures is an x-prehomomorphism w.r.t the set X_k of all relational structures with treewidth at most k.

There exist some other CSP filtering techniques which can be candidates for being x-prehomomorphism, for instance path consistency, singleton arc consistency or singleton path consistency. It can be shown that these procedures can be used to obtain x-prehomomorphism w.r.t. some sets although these sets may be given only implicitly by the particular local consistency techniques.

1.1.9 Structures Constrained by Language Bias

In Section 1.1.4 we mentioned that we can introduce some restrictions on relational structures regarding the relations from σ^n. Sometimes we have an apriori knowledge about problem domain. In Example 1.33 the positions in tuples contained in the relation *bond* have specific meaning. In this example it would be meaningful to keep information about the types of atoms and about the type of bond. This information is preserved in those tuples $bond(a, b, c, d, e)$ where all the elements c, d, e satisfy the condition $isConstant$, i.e. every of these elements is involved in some relation from σ^n. We therefore do not want to keep the tuples in which this information is missing. For this purpose we introduce a special language bias.

Definition 1.42. *Constant language bias* is a set $\mathcal{LB} = \{(R_i/arity_i, \{i_1, \ldots, i_k\})\}$ where R_i are relational symbols from σ^b, $a_i \in \mathbf{N}$ and $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, arity_i\}$. A tuple $(b_1, \ldots, b_k) \in R_i$ is said to comply with language bias $\mathcal{LB}$ if all its elements on positions $\{i_1, \ldots, i_k\}$ satisfy the constraint $isConstant$. A relational structure $\mathbb{A}$ is said to comply with $\mathcal{LB}$ if all its tuples comply with it.

For simplicity we will sometimes use the notation inspired by mode declarations known from Progol. We will write $R_1(x, \#, x)$, $, R_2(\#, x)$ to determine the constant language bias $\{(R_1/3, \{2\}), (R_2/2, \{1\})\}$. So the structures complying with this language bias would have to have at the position 2 in the relation R_1 and at the position 1 in the relation R_2 only elements involved in relations from σ^n.

So the relational structure $\mathbb{A} = (\mathcal{U}_A, R_1^A, R_2^A, name_a, name_b)$ where $R_1^A = \{(X, Y, Z), (W, Y, U)\}$, $R_2^A = \{(X, V)\}$, $name_a = \{Y\}$, $name_b = \{X\}$ complies with the above language bias. However the structure $\mathbb{B} = (\mathcal{U}_B, R_1^B, R_2^B, name_a, name_b)$ where $R_1^B = \{(X, Y, Z), (W, Y, U)\}$, $R_2^B = \{(X, V)\}$, $name_a = \{Y\}$, $name_b = \{\}$ does not comply to it.

Proposition 1.43. Let $\mathcal{LB}$ be a language bias and let $X_{\mathcal{LB}}$ be the set of all relational structures complying with $\mathcal{LB}$. Let $\mathbb{A}$ be a relational structure. If $\mathbb{A}_{\mathcal{LB}}$ is a relational structure obtained from $\mathbb{A}$ by removing all tuples that do not comply with $\mathcal{LB}$ then $\mathbb{A}_{\mathcal{LB}} \to \mathbb{A}$ and $\mathbb{A} \to_X \mathbb{A}_{\mathcal{LB}}$ w.r.t. the set $X_{\mathcal{LB}}$.

In case that we remove some tuples, it can happen that there will be some elements in $\mathcal{U}_{A_{\mathcal{LB}}}$ which are not involved in any relation. Such elements can be also removed when constructing the structure $\mathbb{A}_{\mathcal{LB}}$ and the above proposition will still hold. The above proposition gives us an idea how to construct bounded LGG with respect to a set of relational structures complying with certain language bias $X_{\mathcal{LB}}$.

Proposition 1.44. Let $\mathcal{LB}$ be a language bias and let $X_{\mathcal{LB}}$ be a set of all relational structures complying with it. Let $\mathbb{B} = LGG(\mathbb{A}_1, \ldots, \mathbb{A}_n)$ be a least general generalization of structures $\mathbb{A}_1, \ldots, \mathbb{A}_n$. If $\mathbb{B}_{\mathcal{LB}}$ is a structure obtained from $\mathbb{B}$ by removing all tuples not complaining with $\mathcal{LB}$ then $\mathbb{B}_{\mathcal{LB}}$ is a bounded LGG w.r.t. the set $X_{\mathcal{LB}}$ of structures $\mathbb{A}_1, \ldots, \mathbb{A}_n$. Let X

be a set of relational structures and $\mathbb{C}$ a structure obtained by $\texttt{El-Elim}_X(\mathbb{B}_{\mathcal{LB}})$, then $\mathbb{C}$ is $LGG_{X \cap X_{\mathcal{LB}}}(\mathbb{A}_1, \ldots, \mathbb{A}_n)$.

1.2 Formulation in first order logic form

In this section we show that it is possible to formulate our problem equivalently in the terms of first order logic. We can deal with clauses instead of relational structures. We can test θ-subsumption between clauses instead of homomorphism between relational structures. The original design of used algorithms introduced in [10] and [9] was based on this logical formulation. The equivalent formulation in terms of relational structures provided in this work should be more accessible for most computer science community. The translation of homomorphism decision into a CSP is also more natural for relational structures.

If we treat our input data as a set of relational structures and try to find a structure homomorphic to some of them, we basically look for a common substructure. If we work with the logical formulation we treat our examples as Horn clauses. We try to find a set of rules where every rule is also a Horn clause. The rules and examples have then the form:

$$l_1 \wedge l_2 \wedge \cdots \wedge l_k \Rightarrow \ isPositive\,. \tag{1.1}$$

This rule 1.1 can be equivalent rewritten in the form:

$$\neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_k \vee isPositive\,. \tag{1.2}$$

Here the symbols l_i denote literals, the literal *isPositive* means that example is classified as positive. The important thing is that even the negative examples have this form, i.e. they have the literal *isPositive* in their heads. This can be a little bit confusing but it makes sense when we create our theory as a conjuncion of such Horn clauses.

When we explore our datasets, we look for a common substructure in the sense of 1.1. However the equivalent expressions stated in 1.2 allows us us to use a theory based on clauses. In the following example we illustrate in more detail the correspondence between the two notations.

Example 1.45. Let an example in our dataset be written this way:

$$bond(x_1, x_2, H, C, single),\ bond(x_2, x_3, C, H, double),\ bond(x_4, x_5, C, O, double),$$
$$bond(x_4, x_6, C, O, double).$$

We can either treat it as relational structure $\mathbb{A}$:

$\mathcal{U}_A = \{x_1, x_2, x_3, x_4, x_5, x_6, H, C, O, single, double\}$,
$\mathcal{R}_A = (bond, name_{x1}, \ldots, name_{x6}, name_O, name_H, name_C, name_{single}, name_{double})$,
$bond = \{(x_1, x_2, H, C, single),\ (x_2, x_3, C, H, double),\ (x_4, x_5, C, O, double), (x_4, x_6, C, O, double)\}$,
$name_{x1} = \{(x_1)\},\ name_{x_2} = \{(x_2)\}, \ldots,\ name_{x_6} = \{(x_6)\},\ name_H = \{(H)\}\ \ldots$

Or we can treat it as a Horn clause. For instance if the example is positive it will have the form:

$$bond(x_1, x_2, H, C, single) \wedge bond(x_2, x_3, C, H, double) \wedge bond(x_4, x_5, C, O, double) \wedge$$
$$bond(x_4, x_6, C, O, double) \Rightarrow isPositive,$$

which is equivalent to:

$$\neg bond(x_1, x_2, H, C, single) \vee \neg bond(x_2, x_3, C, H, double) \vee \neg bond(x_4, x_5, C, O, double) \vee$$
$$\neg bond(x_4, x_6, C, O, double) \vee isPositive.$$

During the learning we learn just the left part of the implication because we assume that the head is always the same. We try to find a theory H in this form:

$$\begin{aligned}
&(\neg l_{11} \vee \neg l_{12} \vee \ldots \neg l_{1i_1} \vee isPositive) \wedge \\
&(\neg l_{21} \vee \neg l_{22} \vee \ldots \neg l_{2i_2} \vee isPositive) \wedge \\
&\vdots \\
&(\neg l_{n1} \vee \neg l_{n2} \vee \ldots \neg l_{ni_n} \vee isPositive) \wedge .
\end{aligned}$$

We require H to fulfill $H \models E$ for as many positive examples E as possible and to fulfill $H \not\models E$ for almost all negative examples. Now we can see why we used the same literal in heads of the positive and the negative examples. Let us supppose that we want to classify a general example $E = \neg l_1 \vee \neg l_2 \vee \cdots \vee \neg l_n \vee isPositive$. Let $H = (\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_m)$, where φ_i are Horn clauses. We classify the example as possitive if and only if $H \models E$. This happens if and only if there is at least one $\varphi_i \in H$ such that $\varphi_i \preceq_\theta E$.

It can be seen from the example 1.45 that predicate symbols in first order logic have here the same meaning as the relational symbols in relational structures. The elements of universe of relational structures have the similar function as constants and variables in FOL. We do not work with functional symbols of arity more than 0. In the above example we consider all terms to be constants. However during LGG, we usually obtain some variables too. In the formulation by terms of relational structures we needed to introduce the special unary name relations to preserve information contained in names of the input elements. In the first order logic the respective terms are constants. We explain this correspondence in more details later.

1.2.1 Theta Subsumption and Least General Generalization

Definition 1.46. Let A and B be clauses. The clause A *θ-subsumes* B (denoted by $A \preceq_\theta B$), if and only if there is a substitution θ such that $A\theta \subseteq B$. If $A \preceq_\theta B$ and $B \preceq_\theta A$, we call A and B be *θ-equivalent* and write $A \approx_\theta B$.

As we mentioned above the relation θ-subsumption between clauses is equivalent to homomorphism between relational structures.

Definition 1.47. The *least general generalization* of clauses $A_1, \ldots, A_n$ (denoted by $LGG(A_1, \ldots, A_n)$) is a clause B such that $B \preceq_\theta A_i$ for all $i \in (1, \ldots, n)$ and if there exists another clause C such that for all $i \in \{1, \ldots, n\}$: $C \preceq_\theta A_i$ then $C \preceq_\theta B$.

The least general generalization algorithm for two clauses was proposed in [20]. An algorithm for finding it can be found in [19] or in [17]. This algorithm is similar to that one

we provided for relational structures. We provide this algorithm in a simplified form because we do not have to deal with functional symbols of arity ≥ 1 and with negated predicates.

Least general generalization of two literals with same predicate symbol and same sign $l_1 = p(u_1, \ldots, u_n)$ and $l_2 = p(v_1, \ldots, v_n)$ is a literal $l_3 = p(t_1, \ldots, t_n)$ obtained in the following way:

1. For $i = 1 \ldots n$:
2. If the terms v_i and u_i are the same, then set the term $t_i := u_i$.
3. If u_i and v_i differ and there is $1 \leq j < i$ such that $u_i = u_j$ and $v_i = v_j$ then $t_i := t_j$.
4. If u_i and v_i differ and there is no such j satisfying the above condition then create a new variable not contained in l_1 and l_2 and set t_j to this variable.

Note that if two literals contain the same constant at the same position, the LGG of such literals will have the same constant at this position. This property holds for variables too, but does not influence θ-subsumption testing, because any variable can be substitued by any other term. For variables it is just important to replace the same pairs of terms with the same new variable as stated in the step 3.

Least general generalization of two clauses A and B is obtained in this way:

1. Begin with an empty clause C.
2. For all pairs of literals $l_1 \in A$ and $l_2 \in B$ such that l_1 and l_2 have the same predicate symbols (and same signs) create literal l as their LGG and extend the clause C by it: $C := C \vee l$.

The algorithm is similar to the one for relational structures. Let us denote the number of literals of a clause A by $|A|$. The size of least general generalization of two clauses A and B obtained by the above algorithm can be up to $|A| \cdot |B|$. The resulting clause can be reduced so that the reduction is θ-equivalent to the original clause. Such a reduction is also $LGG(A, B)$. For this purpose we provide a definition of θ-reduction.

Definition 1.48. Let A be a clause. If there is another clause $\tilde{A}$ such that $A \approx_\theta \tilde{A}$ and $|\tilde{A}| \leq |A|$ then A is said to be *θ-reducible*. Minimal $\tilde{A}$ such that $A \approx_\theta \tilde{A}$ is called *θ-reduction*.

The θ-reduction has the equivalent function for clauses as the core has for relational structures.

1.2.2 Formulation as CSP

The problem of θ-subsumption can be formulated as a CSP similarly as homomorphism. If we want to decide whether $A \preceq_\theta B$, we need to find a substitution for variables of A such that $A\theta \subseteq B$. The transformation into CSP is as follows:

- For every variable of A there exists exactly one CSP variable.
- In every domain of every variable there are all terms of B.

- There is exactly one constraint $c_l = (var(c_l), rel(c_l))$ corresponding to every literal $l = pred_l(t_1, \ldots, t_k) \in A$. Let us denote $(i_1, \ldots, i_m) \subseteq (1, \ldots, k)$ the positions in l such that the terms t_{i_j} are variables. Then $var(c_l) = (t_{i_1}, \ldots, t_{i_m})$ and $rel(c_l)$ is constructed as follows:

  1. Let L_l be the set of all literals l' in B such that $l' \preceq_\theta l$.
  2. We now construct the set of tuples in $rel(c_l)$ in this way: For all $l' \in L_l$, where $l' = p_l(t'_1, \ldots, t'_k)$ it holds $(t'_{i_1}, \ldots, t'_{i_m}) \in rel(c_l)$.

The construction is analogical to the one for the relational structures. The condition stated in point 1 is a little big stronger then the corresponding condition for relational structures. Similarly we have to choose only those literals from B which have the same predicate symbol as l. In addition point 1 tells us that if there is a constant at the i-th position in l, there has to be the same constant at the i-th position in all $l' \in L_l$. In the section concerned with relational structures we had to introduce special name relations to ensure mapping satisfying these properties. In the basic formulation for relational structures these unary relations are involved as ordinary unary conditions. However in the case of relational structures we can use the same formulation as we use for clauses without changing any previously stated properties because such a formulation is equivalent to the original one. This is described in more details in Section 2.1.7.

Of course theta subsumption between clauses can be also formulated as CSP in the alternative dual way. We do not provide the full description of this, because it is quite the same as for relational structures and can be also found in [9].

1.2.3 X-subsumption and bounded LGG

Analogically as we defined bounded operations for relational structures, we now provide definitions of x-subsumption and x-presubsumption for clauses.

Definition 1.49. Let X be a possibly infinite set of clauses. Let A, B be clauses not necessarily from X. We say that A *x-subsumes* B w.r.t. X (denoted by $A \preceq_X B$) if and only if for every clause $C \in X$ $(C \preceq_\theta A) \Rightarrow (C \preceq_\theta B)$. If $A \preceq_X B$ and $B \preceq_X A$ then A and B are called *x-equivalent* w.r.t. X (denoted by $A \approx_X B$). For a given set X, the relation $\preceq_X$ is called *x-subsumption* w.r.t. X and the relation $\approx_X$ *x-equivalence* w.r.t. X.

Similar propositions which hold for relational structures hold also for clauses. θ-subsumption is x-subsumption w.r.t. the set of all clauses. The operation x-subsumption w.r.t. set of clauses X is transitive and reflexive relation on clauses, x-equivalence w.r.t. X is an equivalence relation on clauses.

Definition 1.50. Let X be set a set of clauses. Let $\lhd_X$ be a relation such that for any two clauses A and B: $(A \preceq_\theta B) \Rightarrow (A \lhd_X B)$ and $(A \lhd_X B) \Rightarrow (A \preceq_X B)$. Then $\lhd_X$ is called *x-presubsumption* w.r.t. the set X.

The operation x-subsumption for clauses is equivalent to the relation x-homomorphism for relational structures and x-presubsumption is equivalent to x-prehomomorphism. The equivalent proposition for clauses as Proposition 1.39 holds therefore for clauses. Therefore we can use it for finding a reduction of a clause x-equivalent to the clause. For this purpose

we can use so called Literal-elimination algorithm analogical to the Element-elimination algorithm.

Literal-elimination algorithm:

1. Given a clause A which should be reduced.
2. Set $A' := A$.
3. Select a variable y such that $A' \lhd_X A' \backslash L$, where L is a set of all literals of A' containing variable y. If there is no such variable, return A' and finish.
4. Set $A' := A' \backslash L$.
5. Go to step 3.

If A is an input clause of this algorithm and the operation $\lhd_X$ is an x-presubsumption w.r.t. the set X, then the output clause A' satisfies the following conditions:

- $A' \preceq_\theta A$ and $A \preceq_X A'$ w.r.t. the set X.
- $|A'| \leq |A_\theta|$ where A_θ is θ-reduction of a subset of literals of A with maximum length.
- If $A_\theta \in X$ then $A' \approx_X A_\theta$ and $|A'| = |A_\theta|$ regardless the set X.

All proofs of the above propositions can be found in the work [9].

The x-presubsumption with respect to some useful sets of clauses can be again tested by using CSP techniques. We just define the problem of θ-subsumption between two clauses as a CSP and then run one of the filtering techniques. If we want to test x-presubsumption w.r.t. the set of all clauses of treewidth at most k, we can use k-consistency algorithm. For testing x-presubsumption w.r.t. the set of all acyclic clauses we use generalized arc consistency algorithm etc.

The definition of treewidth of a clause A is provided according to its Gaifman graph. In this graph there is exactly one vertex for each variable from A and one edge for each pair of variables u and v such that $u \neq v$ and u and v both appear in a common literal $l \in A$.

Definition 1.51. A clause C is said to be *acyclic* if the iteration of the following rules on C produces the empty clause:

1. Remove a literal such that all its variables are contained in another literal.
2. Remove a variable which is contained in at most one literal.

Definition 1.52. Let X be set of clauses. A clause B is said to be a bounded least general generalization of clauses $A_1, \ldots, A_n$ w.r.t. the set X (denoted by $B = LGG_X(A_n, \ldots, A_n)$) if and only if and only if $B \preceq_\theta A_i$ for all $i \in \{1, \ldots, n\}$ and if for every other clause $C \in X$ such that $C \preceq_\theta A_i$ for all $i \in \{1, \ldots, n\}$ it holds $C \preceq_\theta B$.

Again the bounded LGG w.r.t. the set of clauses X can be obtained by applying literal elimination algorithm (with bounded presubsumption w.r.t. X) on the result of least general generalization algorithm.

The language bias introduced in Section 1.1.9 has much simpler form for clauses. We choose some predicate and restrict certain positions in corresponding literals to contain only constants.

Definition 1.53. Constant language bias is a set $\mathcal{LB} = \{(p_i/arity_i, \{i_1, \ldots, i_k\})\}$ where p_i are predicate symbols and $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, arity_i\}$. A literal $l = p_i(t_1, \ldots, t_k)$ is said to comply with language bias $\mathcal{LB}$ if it contains constants in all positions $i_1, \ldots, i_k$. A clause C is said to comply with language bias $\mathcal{LB}$ if all its literals comply with it.

For simplicity we will again write for example $p_1(x, x, \#),\ p_2(\#, \#, x, x)$ to choose important predicate symbols and denote by # positions in corresponding literals, which are required to contain only constants. The set of clauses complying with some language bias can be again used as a reference set of clauses for bounded operations. Bounded LGG with respect to the set of all clauses complying with a language bias $\mathcal{LB}$ can be computed from original LGG by removing from given clause all literals not complying with it.

Chapter 2

Algorithms

2.1 Learning algorithm

Our learning procedure is based on homomorphism between relational structures. The input is a set of examples in the form of relational structures. Every example has a label indicating whether this is a positive or a negative example. For instance we can have a dataset containing protein structures with or without a specific function. Every structure has a label indicating whether this protein has this specific function or not. Our goal is then to find a discriminative set of structures which will cover preferably all positive examples and if possible no negative examples. So the result of our learning procedure is a set of structures. If some example is covered by at least one of the resulting structures, we classify it as positive, otherwise we classify it as negative.

If we say that an example $\mathbb{E}$ is covered by another structure $\mathbb{H}$ (or $\mathbb{H}$ covers example $\mathbb{E}$), we mean that it holds $\mathbb{H} \to \mathbb{E}$ (or $H \preceq_\theta E$ for clauses). Sometimes we will use the word cover also for bounded homomorphism and bounded subsumption. By describing general principles of the used procedures it is not always needed to specify exactly which kind of coverage we have in mind because both operations are tightly related and often both hold. If necessary, we will always specify which operation we mean. We will denote as a hypothesis some relational structure created during learning, which can be further refined and improved or which can be also returned as the output of the learning procedure.

Our learning algorithm uses all algorithms described in previous sections. The basic idea is to apply (bounded) LGG iteratively on positive examples. If we apply bounded LGG on two structures, it often happens that as a side effect a lot of other examples not yet involved in LGG will be covered too. This side effect is desirable if positive examples are covered but unwanted if negative examples are covered. In addition if an example not involved in LGG algorithm is covered only by bounded homomorphism, we do not know whether it is covered by homomorphism. However if an example is not covered by a bounded homomorphism, we are sure that it is also not covered by homomorphism, because it holds $(\mathbb{A} \to \mathbb{B}) \Rightarrow (\mathbb{A} \to_X \mathbb{B})$. If we want to ensure that an example E not involved in LGG and covered by bounded homomorphism will be covered by a hypothesis $\tilde{H}$ by means of homomorphism, we have to modify the hypothesis in this way: $\tilde{H} := LGG_X(\tilde{H}, E)$. In the following algorithm we have to use this approach for examples in the set *newPosCovered*. This complication disappears if we directly use homomorphism instead of bounded homo-

morphism to test which examples are covered as this side effect.

2.1.1 Basic Procedure

1: **function** LEARNING ALGORITHM(positive examples $\mathcal{E}^+$, negative examples $\mathcal{E}^-$, seed example S)
2: $Open := ()$
3: $Closed := \{\}$
4: $PositiveCovBySeed := \{S\}$
5: $NegativeCovBySeed := \{E \in \mathcal{E}^- | S \lhd_X E\}$
6: $BestScore := |NegativeCovBySeed| - |PositiveCovBySeed|$
7: $BestTriple := (S, PositiveCovBySeed, NegativeCovBySeed)$
8: Add the triple $(S, PositiveCovBySeed, NegativeCovBySeed)$ to $Open$
9: Add the pair $(PositiveCovBySeed, NegativeCovBySeed)$ in the set $Closed$
10: **while** $Open \neq \emptyset$ **do**
11: $(H, \{E^+_{i_1}, \ldots, E^+_{i_n}\}, \{E^-_{j_1}, \ldots, E^-_{j_m}\})$:=remove best element from $Open$
12: **for all** $E^* \in CandidateExamples(\mathcal{E}^+ \backslash \{E^+_{i_1}, \ldots, E^+_{i_n}\})$ **do**
13: $H^* := LGG_X(H, E^*)$
14: $PosCovered := \{E^+_{i_1}, \ldots, E^+_{i_n}\} \cup \{E^*\}$
15: $NegCovered := \{E^-_{j_1}, \ldots, E^-_{j_m}\}$
16: $\tilde{H} := H^*$
17: **repeat**
18: $NewPosCovered := \{E \in (\mathcal{E}^+ \backslash PosCovered) | H^* \lhd_X E\}$
19: $\tilde{H} := LGG_X(\tilde{H}, C_1, \ldots, C_k)$ where $C_i \in NewPosCovered$
20: $NewNegCovered := \{E \in (\mathcal{E}^- \backslash NegCovered) | H^* \lhd_X E\}$
21: **if** $|NewNegCovered| + |NegCovered| > maxNegCovered$ **then**
22: $hypothesisUseful := false$
23: **break**
24: **else**
25: $H^* := \tilde{H}$
26: $NegCovered := NegCovered \cup NewNegCovered$
27: $PosCovered := PosCovered \cup NewPosCovered$
28: **end if**
29: **until** $NewPosCovered \neq \emptyset$
30: **if** $hypothesisUseful$ and $(PosCovered, NegCovered) \notin Closed$ **then**
31: Add the triple $(H^*, PosCovered, NegCovered)$ to $Open$
32: Add the pair $(PosCovered, NegCovered)$ to $Closed$
33: $Score := |NegCovered| - |PosCovered|$
34: **if** $Score < BestScore$ **then**
35: $BestScore := Score$
36: $BestTriple := (H^*, PosCovered, NegCovered)$
37: **end if**
38: **else**
39: $\tilde{H} := H^*$
40: **for all** $C_i \in NewPosCovered$ **do**
41: $\tilde{H} := LGG_X(H^*, C_i)$
42: $NewNegCovered := \{E \in (\mathcal{E}^- \backslash NegCovered) | \tilde{H} \lhd_X E\}$

43: **if** $|NewNegCovered| + |NegCovered| > maxNegCovered$ **then**
44: **break**
45: **else**
46: $H^* := \tilde{H}$
47: $NegCovered := NegCovered \cup NewNegCovered$
48: $PosCovered := PosCovered \cup \{C_i\}$
49: **end if**
50: **end for**
51: $Score := |NegCovered| - |PosCovered|$
52: **if** $Score < BestScore$ **then**
53: $BestScore := Score$
54: $BestTriple := (S, PosCovered, NegCovered)$
55: **end if**
56: **end if**
57: **end for**
58: **end while**
59: run post-processing procedures
60: **return** *bestTriple*
61: **end function**

The algorithm contains some user definable parameters, for example $maxNegCovered$. This number specifies how many negative example we allow to be covered by one hypothesis. In the datasets we explored by our methods so far this parameter can be set to zero. The number of repetitions of the cycle "while $Open \neq \emptyset$" can be unreasonably high. Therefore we usually reduce the number by introducing a parameter called $maxHypotheses$ determining the maximum number of this cycle repetitions. Another similar parameter is called $maxCandidateExamples$. This parameter determine how many examples from the set $(\mathcal{E}^+ \backslash \{E^+_{i_1}, \ldots, E^+_{i_n}\})$ we will use for extension of one hypothesis obtained from the set *Open* as its best element. The examples are selected from this set by random.

2.1.2 Dealing with large structures

In our experiments we always use a suitable language bias. This language bias can be included in the basic version of the LGG algorithm. We do not have to create tuples (literals) not complying with it and remove them later. We just do not create them at all. This approach reduces the size of a structure resulting from the algorithm, but this structure can be too large to perform element elimination without any preprocessing in reasonable time. So in our implementation we incorporated a special algorithm for dealing with large structures originating from LGG algorithm. The procedure is based on the idea that the whole structure contains only a few connected components important for distinguishing the positive examples. We also prefer small components to be preserved. For this purpose we use the following algorithm. We will denote by variables the elements of universe satisfying the condition $isVariable$.

1: **function** LGG REDUCTION ALGORITHM(relational structure H,$maxVariables$)
2: Decompose the input structure to connected components
3: Sort the components from the smallest (in the sense of count of tuples) to the largest
4: Create an empty list *selectedComponents*
5: **while** $|usedVariables| < maxVariables$ **do**

```
6:        A := remove the smallest component from their list
7:        if A ◁_X C does not hold for any C ∈ selectedComponents then
8:            add A to selectedComponents
9:            add all variables in A into usedVariables
10:       end if
11:   end while
12:   if |usedVariables|>maxVariables then
13:       A := remove last from selectedComponents
14:       create empty structure Ã
15:       while |usedVariables| < maxVariables do
16:           pick a random variable v from A
17:           add v to Ã
18:           add to Ã all tuples involving v including elements in those tuples
19:           add to Ã name relational tuples for added elements
20:           add all new variables in Ã into usedVariables
21:       end while
22:       add Ã to selectedComponents
23:   end if
24:   S := compose a structure from all structures in selectedComponents
25:   return S
26: end function
```

Let us denote the input structure of the above algorithm by $\mathbb{A}$ and the output structure by $\mathbb{B}$. $\mathbb{B}$ is a substructure of $\mathbb{A}$. So that $\mathbb{B} \to \mathbb{A}$. Now we provide an idea why this heuristic approach can be useful.

We can consider that in the case of homomorphism mapping the elements contained in a single connected component in the source structure will be mapped into elements from a single connected component in the other structure. This fact allows us to use the $\lhd_X$ testing only for single components and not for the whole structure composed from them. In addition we know that every element from *selectedComponents* can be homomorphically mapped to itself therefore it is enough to test bounded homomorphism only for the "new" component instead of for all components in the list *selectedComponents*.

2.1.3 Post-processing of a learned hypothesis

The learning algorithm gives us a relational structure, which can be further processed. In this section we provide some possible procedures for doing this. This procedures can be optionally performed in the step denoted by post-processing procedures in the learning algorithm.

Component elimination and Variable elimination

We iteratively try to remove one component after another and test whether the reduced structure is x-homomorphic to some not yet covered negative example. If not, we remove the component from the structure, otherwise we preserve it. This procedure could help us to find a better hypothesis because the original structure can be too large and therefore too specific, which can lead to worse generalization on unseen examples. We provide two algorithms which can be optionally performed in the post-processing step of the learning

algorithm to further reduce the learned structure. In our implementation either one of them or both or none of them can be performed.

The following algorithm shows the reduction procedure based on components. The input is a set of negative examples and a triple $(H, PosCovered, NegCovered)$, which is an output of learning algorithm.

1: **function** COMPONENT ELIMINATION(negative examples $\mathcal{E}^-$,hypothesis with covered examples $(H, posCovered, negCovered)$)
2: $components :=$ decompose H into connected components
3: Sort the *components* from the smallest (in the sense of number of tuples) to the largest
4: $H^* := H$
5: **while** $components \neq \emptyset$ **do**
6: $C :=$ remove largest from *components*
7: $\tilde{H} := H^* \backslash C$
8: $newNegCovered := \{E \in (\mathcal{E}^- \backslash negCovered) | \tilde{H} \lhd_X E\}$
9: **if** $newNegCovered = \emptyset$ **then**
10: $H^* := \tilde{H}$
11: **end if**
12: **end while**
13: **return** $(H^*, posCovered, negCovered)$
14: **end function**

Another possibility how to reduce an obtained hypothesis is the VARIABLE ELIMINATION algorithm. This algorithm proceeds similarly as COMPONENT ELIMINATION but removes only variables and their corresponding tuples instead of components. This approach is slower than COMPONENT ELIMINATION but allows stronger reduction. It is reasonable to use first COMPONENT ELIMINATION to fast rough reduction and after that VARIABLE ELIMINATION to slower firm reduction.

1: **function** VARIABLE ELIMINATION(negative examples $\mathcal{E}^-$,hypothesis with covered examples $(H, posCovered, negCovered)$)
2: $variables :=$ elements from H satisfying $isVariable$
3: $H^* := H$
4: **for all** $v \in variables$ **do**
5: $T :=$ all tuples from H^* containing v
6: set $\tilde{H}$ to H^*
7: from $\tilde{H}$ remove a and tuples from T
8: $newNegCovered := \{E \in (\mathcal{E}^- \backslash negCovered) | \tilde{H} \lhd_X E\}$
9: **if** $newNegCovered = \emptyset$ **then**
10: $H^* := \tilde{H}$
11: **end if**
12: **end for**
13: **return** $(H^*, posCovered, negCovered)$
14: **end function**

In those two above algorithms we do not update the information about new covered positive examples. We basically do not need to do so because our main motivation in this algorithms is to simplify a learned hypothesis but such a score update can provide us more accurate information about the quality of a hypothesis. As it was seen in the learning exam-

ple, this is not easy when we want to store examples covered by means of homomorphism but do not want to test it explicitly. In this case we have to perform LGG.

Now we provide an algorithm, which we perform to update the score of a hypothesis. We call this algorithm LGG SCORE UPDATE. We perform this update of score in the end of the learning algorithm in case of using at least one of the algorithms for reduction. In this phase the input of the LGG SCORE UPDATE algorithm is the triple resulting from the reduction algorithms and the same sets of positive and negative examples as are in the input of the learning algorithm.

We will describe in the following section that we perform the learning algorithm repeatedly and remove covered positive examples after every run. If we obtain a new hypothesis from the learning algorithm, the set *posCovered* in the resulting triple contains only those positive examples, which were given in the input of the learning algorithm. Therefore we need to perform a score update to obtain an information about covering those positive examples, which were not contained in the input of the learning algorithm. For this purpose we again use LGG SCORE UPDATE, but this time we provide it all positive examples in the input.

1: **function** LGG SCORE UPDATE($\mathcal{E}^+, \mathcal{E}^-, (H, posCovered, negCovered)$)
2: $variables :=$ elements from H satisfying $isVariable$
3: $H^* := H$
4: $newPosCovered := \{E \in (\mathcal{E}^+ \backslash posCovered) | H \lhd_X E\}$
5: **for all** $E \in newPosCovered$ **do**
6: $\tilde{H} := LGG_X(H^*, E)$
7: $newNegCovered := \{E \in (\mathcal{E}^- \backslash negCovered) | \tilde{H} \lhd_X E\}$
8: **if** $newNegCovered = \emptyset$ **then**
9: $H^* := \tilde{H}$
10: $posCovered := posCovered \cup \{E\}$
11: **end if**
12: **end for**
13: **return** $(H^*, posCovered, negCovered)$
14: **end function**

2.1.4 Learning a set of hypotheses

In this section we describe how to obtain a complete set of hypotheses. The algorithm is based on repetition of LEARNING ALGORITHM, optionally including post-processing algorithms.

1: **function** CLASSIFIER LEARNING($\mathcal{E}^+, \mathcal{E}^-$,$minPositiveCovered$)
2: $allClasifiers := \{\}$
3: **for** $i = 1 : outerRep$ **do**
4: $setHypotheses := \{\}$
5: $notUsedPositive := \mathcal{E}^+$
6: $j := 0$
7: $listHypotheses := \{\}$
8: **while** $j < innerRep$ and $|notUsedPositive| > minPositiveCovered/2$ **do**
9: $j := j + 1$
10: $S :=$ choose a random example from $\mathcal{E}^+$
11: $(H, posCovered, negCovered) :=$LEARNING ALGORITHM($notUsedPositive, \mathcal{E}^-, S$)

```
12:            (H, posCovered, negCovered) :=
13:                                  LGG SCORE UPDATE(E+, E−, (H, posCovered, negCovered))
14:            if |posCovered| > minPositiveCovered then
15:                add H to listHypotheses
16:                j := 0
17:                remove posCovered from notUsedPositive
18:            end if
19:        end while
20:        listHypotheses to allClasifiers
21:    end for
22:    select final classifier from allClasifiers
23:    return final classifier
24: end function
```

An important task before this algorithm is the finding of a threshold *minPositiveCovered*. This part is described in details in the next section. An important part of this algorithm is the selection of a final classifier from a set of lists. In our experiments we usually set the parameter *maxNegCovered* to zero. Therefore the score of a hypothesis is fully determined by the size of the set *posCovered* of the respective hypothesis. In our implementation we allow two possibilities how to select the final classifier from the set $allClasifiers$. The first is to select the best list from $allClasifiers$ in the sense of count of all positive examples covered together by all hypotheses of the list minus all negative examples covered by all hypotheses. The second one is to use all hypotheses in $allClasifiers$ as a final classifier. However in case that we allow the threshold *maxNegCovered* to be greater than zero the second approach is not good enough. A possible choice can be for example using pseudo boolean optimization.

2.1.5 How to set the *minPositiveCovered* parameter

In this section we describe how to set the threshold *minPositiveCovered*. The threshold can be set by user. We allow it to be set to certain percentage of positive examples. Another option is to use our heuristic approach based on randomized labels of examples described in this section. First we randomize labels of input examples so that we preserve the ratio of positive and negative examples described in this section. After that we run the learning algorithm with randomized labels and test how many fake positive examples are covered by the obtained hypothesis. This we perform repeatedly to obtain a more reliable estimation. The idea behind this approach is that we can find out how many examples can be covered randomly regardless their positivity or negativity.

```
1: function SET MINIMUM COVERED(E+, E−)
2:     p := |E+|
3:     n := |E−|
4:     setSizes := {}
5:     for i = 1 : countRepetition do
6:         E := {E+} ∪ {E−}
7:         R+ := randomly select p examples from E
8:         R− := E\R+
9:         S := random example from R+
10:        (H, posCovered, negCovered) := LEARNING ALGORITHM(R+, R−, S)
```

11: $s := |posCovered|$
12: add s to $setSizes$
13: **end for**
14: $t :=$ guess a threshold from $setSizes$
15: **return** t
16: **end function**

How to exactly obtain the threshold from $setSizes$ can be set up by user. We sort the elements in $setSizes$ from the largest to the smallest and choose m-th number in the row as the threshold t. So we allow the user to choose the number m to determine which number to take as the threshold. We usually set the parameter $countRepetition$ to 10 and the parameter m to 3. This setting seems to be quite appropriate in our experiments.

2.1.6 Variants of Element elimination

The element elimination algorithm described in Section 1.1.6 exploits x-prehomomorphism to test whether the current relational structure is x-homomorphic to the reduced structure. We can also use homomorphism instead of x-prehomomorphism in this step of the algorithm. In this way the algorithm outputs a core of the input relational structure. Whereas by the original element elimination we obtain a structure which is just x-equivalent to the input structure and which is a core but not necessarily a core of the input structure.

In fact, we always use a language bias in our experiments so basically this algorithm also outputs bounded LGG but this procedure is able to find a core of the input structure if no language bias is used.

A disadvantage of this approach is that homomorphism testing is NP-complete task. We can reduce the impact of using such operation by an additional improvement, which at least reduces the number of homomorphism tests.

Let us imagine that we find out that it holds $\mathbb{A} \to \mathbb{B}$. We solved this problem using CSP. Therefore we obtained not only the information that $\mathbb{A} \to \mathbb{B}$ but also the concrete mapping $f : \mathcal{U}_A \to \mathcal{U}_B$. Now we denote the range of f as $Range(f)$. We can consider an induced substructure $\mathbb{C}$ of the structure $\mathbb{B}$ such that $\mathcal{U}_C = Range(f)$. It is obvious that $\mathbb{A} \to \mathbb{C}$ because the mapping f is also the required homomorphism from $\mathbb{A}$ to $\mathbb{C}$.

We can use the above fact in our element elimination algorithm where we use homomorphism testing. In every reduction step we reduce our current structure in the way described in the previous paragraph. Our enhanced element elimination is then as follows:

1: **function** ELEMENT ELIMINATION COMPLETE(relational structure $\mathbb{A}$)
2: Set $\mathbb{B} := \mathbb{A}$.
3: Select an element $a \in \mathcal{U}_B$ such that $isVariable(a)$, $\mathbb{B}' =$REDUCE($\mathbb{B}$,a), $f : \mathcal{U}_B \to \mathcal{U}_{B'}$ and f is a homomorphism
4: **if** no such element a exists **then**
5: **return** $\mathbb{B}$ and finish.
6: **else**
7: $\mathbb{C} :=$ induced substructure of $\mathbb{B}'$ such that $\mathcal{U}_C = Range(f)$
8: Set $\mathbb{B} := \mathbb{C}$.
9: Go to step 3.
10: **end if**
11: **end function**

We can see that in the step 7. we can equivalently use an induced substructure of $\mathbb{B}$ and we would obtain the same result. The result of this algorithm is a core of the input structure $\mathbb{A}$. It is obvious that $\mathbb{A} \approx \mathbb{B}$. And $\mathbb{B}$ has no induced substructure $\mathbb{B}'$ such that $\mathbb{B} \to \mathbb{B}'$.

We can also remove more elements from the structure $\mathbb{B}$ in case of standard element elimination. Let us imagine that we found out that $\mathbb{A} \lhd_{GAC} \mathbb{B}$. We tested it so that we formulated the homomorphism decision from $\mathbb{A}$ to $\mathbb{B}$ as a CSP and then used the GAC3 algorithm. Let us consider the case of the standard CSP formulation, which means that the elements from $\mathcal{U}_A$ are CSP variables and elements from $\mathcal{U}_B$ are in their domains. The GAC3 algorithm pruned the domains of CSP variables. It can now happen that some elements from $\mathcal{U}_B$ do not occur in any domain. In this case we are sure that we can remove them from the structure $\mathbb{B}$ and the relation $\lhd_{GAC}$ will still hold. So we create a set $D := \bigcup_{a \in \mathcal{U}_B} Dom(a)$. We now create a relational structure $\mathbb{C}$ such that $\mathbb{C}$ is an induced substructure of $\mathbb{B}$ and $\mathcal{U}_C = D$. Now we can easily consider that $\mathbb{A} \lhd_{GAC} \mathbb{C}$.

The most important improvement which speeds up our implementation is as follows. We create a formulation of a CSP at the beginning of our elimination algorithm. If we find out that a structure can be reduced, we only remove redundant variables from the CSP domain. It is not necessary to create a completely new formulation, which is extremely time consuming. This is one of the main reasons why we had to implement our own CSP solver.

A similar approach can be used if we use other local consistency technique. In the case of dual CSP formulation we proceed similarly.

2.1.7 Remarks to CSP implementation

We implemented our own CSP solver and algorithms for local consistency GAC3 and SAC. The new implementation of CSP solver was necessary because existing solvers like Choco are too complicated and are rather designated for a straightforward use. However our algorithms often require to use only specific techniques like concrete local consistency technique or to modify a created CSP formulation by for example removing some variables as described in the previous section. In our implementation we use some enhancements in the CSP formulation and in the solver which can contribute to speed-up of the algorithms. Some of them we describe in this section.

In this section we will concern with techniques used for solving CSP formulated for homomorphism decision $\mathbb{A} \to \mathbb{B}$.

Let us consider that we have structures containing hundreds of elements but most of them are involved in just a few tuples and let us consider the case of standard CSP formulation as described in Section 1.1.2. If we create the same domains containing all elements from $\mathcal{U}_B$ for all CSP variables, the first run of GAC3 would take a lot of time. Because the algorithm has to iterate over all variables, for every variable over all its constraints and for all the constraints over all values contained in the domain of the variable and check whether this value can satisfy the constraint. In this case many values in the domains are redundant and we can discard them prior to the first arc consistency checking. During composition of the task we in fact do not add all elements from $\mathcal{U}_B$ to all variable domains. We add into $Dom(a)$ the value v if and only if $\exists \tau \exists c_i$ such that $\tau \in rel(c_i)$ and $\tau[a] = v$. After the basic construction we can perform the following algorithm to prune the domains:

1: **function** PREPROCESS DOMAINS(constraint satisfaction problem (X, D, C))

2: **for all** $c_i = ((x_{i_1}, \ldots, x_{i_n}), rel(c_i)) \in C$ **do**
3: **for all** $k := 1 : arity(c_i)$ **do**
4: $\tilde{D} := \{v | \exists \tau \in rel(c_i), \tau[x_{i_k}] = v\}$
5: $Dom(x_{i_k}) := Dom(x_{i_k}) \cap \tilde{D}$
6: **end for**
7: **end for**
8: **end function**

We can see that during this procedure we made the problem node consistent and pruned the domains also according to constraints of a higher arity. In the case of dual formulation we do not use the algorithm PREPROCESS DOMAINS but we still do not create domains exactly as described in Section 1.1.2. We describe the construction in more details after explaining how to simplify both constructions of CSP according to name relations.

Another remark to CSP formulation is related to the name relations. We first describe the case of the standard formulation. We mentioned in Section 1.2.2 that the unary name relations would appear as unary constraints in the original formulation. As we mentioned before our arc consistencies always ensure node consistency too. We ensure this by PREPROCESS DOMAINS and by slightly modified version of GAC3 algorithm. Our implementation of SAC uses GAC3 so that the node consistency also holds after one run of SAC. These unary constraints ensure that every element $a \in \mathcal{U}_A$ satisfying *isConstant* can be mapped into at most one element $b \in \mathcal{U}_B$. After first run of any arc consistency algorithm every tuple $\tau \in c_i$ such that $a \in var(c_i)$ and $\tau[a] \neq b$ can not be evaluated as *valid* in the algorithm EXISTS. Therefore we can discard those tuples during creating our CSP and we do not create CSP variables for elements from $\mathcal{U}_A$ satisfying *isConstant*. Therefore the construction of a CSP for relational structures is now the same as for FOL clauses. For example: We can map the tuple $\tau = (x, y, z) \in R_i^A$ where $name_1^A = \{(x)\}$ into a tuple $\gamma = (k, l, m) \in R_i^B$ only if $name_1^B = \{(k)\}$.

A similar approach can be used for the dual formulation. Let us imagine two elements $x \in \mathcal{U}_A$ and $y \in \mathcal{U}_B$ such that $name_1(x)$ and $name_1(y)$. Originally we should create a constraint c such that $vars(c) = (v_x, v_{name_1(x)})$, $rel(c) = (y, name_1(y))$. After first run of arc consistency algorithm the domains of v_x and $v_{name_1(x)}$ would be $Dom(v_x) = \{y\}$, $Dom(v_{name_1(x)}) = name_1(y)$. It can be seen that the values for such CSP variables are always unique and we do not have to involve them in CSP variables. However we have to slightly modify the formulation. Let us have a CSP variable v_τ corresponding to a tuple $\tau = (x, y, z) \in R_i^A$ where again $name_1^A = \{(x)\}$. We put in the domain of v_τ only those tuples $\gamma = (k, l, m) \in R_i^B$ where $name_1^B = \{(k)\}$.

Now we describe how to construct domains for CSP variables corresponding to elements from $\mathcal{U}_A$ satisfying $isVariable$. We add into $Dom(a)$ the value v if and only if $\exists \tau \exists R_i$ such that $a \in \tau \in R_i^A$ and $\exists \gamma \in R_i^B$ such that τ can be mapped into γ and v is in γ at the same position as a is in τ.

Constraint testing

Let us again assume that we solve a problem $\mathbb{A} \to \mathbb{B}$ formulated as a CSP. The algorithms REVISE and GAC3 do not differ according to standard and dual formulation. The algorithm GAC3 is designed for general problems with constraints of various arity but it can be also used as arc consistency algorithm for the dual formulation, which contains only binary

constraints. However for example the algorithm Exists differs for the standard and for the dual formulation. In both cases we use one table for every tuple $\tau \in R_i^A$ in every $R_i^A \in \mathcal{R}_A$. Such a table has one row for each tuple $\gamma \in R_i^B$ such that τ can be mapped into γ. There is one column in the table for every element from τ satisfying $isVariable$.

In the case of dual formulation we have one constraint c for every pair $\tau \in R_i^A$ and $a \in \mathcal{U}_A$ where $a \in \tau$ and a satisfies $isVariable$. We now describe the principle of the function Exists(v, c, j) where v is a CSP variable, c is a constraint and j is a possible value for v. Let us assume that $vars(c) = (v_\tau, v_a)$. There are two possible scenarios of the function:

1. Exists(v_τ, c, γ): We look into the table of τ at the row corresponding to γ and at the column corresponding to the variable a. Let us assume that the there is a value y. If $y \in Dom(a)$ we return true otherwise we return false.

2. Exists(v_a, c, y): We look into the table of τ. For all $\gamma \in Dom(v_\tau)$ we look into the row corresponding to γ. If at least one of this rows contains y at the position corresponding to a, we return true otherwise we return false.

Next difference lists

In case of standard formulation we use the tables straightforwardly. The algorithm Exists works exactly as it is described in Section 1.1.2. The iteration over tuples is implemented as iteration over rows of the table. To speed up the search we enhanced the tables by using next difference lists [7]. We can encode every element from $\mathcal{U}_B$ as a positive integer. Let us assume that the relation $\tau = (x, y, z) \in R_i$ contains only elements satisfying $isVariable$. We sort its table lexicographically and assign to every element in the i-th row and in the j-th column a pointer to the row index higher than i containing next different value in the column j or a null pointer. The table 2.1.7 presents an example of such a table for tuple τ. The columns correspond to variables x, y, z. The firs row $(1^{*4}, 3^{*2}, 7^{*2})$ means that $x = 1, y = 3, z = 7$. The numbers in the superscripts marked with $*$ are pointers to rows.
The pointers in first row $(1^{*4}, 3^{*2}, 7^{*2})$ mean that:

1. The closest row with a value of x different from 1 is the row number 4.
2. The closest row with a value of y different from 3 is the row number 2.
3. The closest row with a value of z different from 7 is the row number 2.

row number	x	y	z
1.	1^{*4}	3^{*2}	7^{*2}
2.	1^{*4}	4^{*4}	8^{*3}
3.	1^{*4}	4^{*4}	9^{*4}
4.	2^{*0}	3^{*0}	7^{*5}
5.	2^{*0}	3^{*0}	9^{*0}

Table 2.1: Example of next difference lists

The pointers point always to higher rows. The pointers $*0$ means that there is no higher number row containing different value for the variable.

Let us assume that in the algorithm Exists we find out that $1 \notin Dom(x)$. We can now continue directly to the row number 4 because it is the first row containing different

value for x. If we also find out that $2 \notin Dom(x)$, we will be sure there is no valid tuple in the table. If the input parameters of the function are EXISTS(v, c, j), the lists also help us to effectively test whether $\tau[v] = j$.

Storing domains for backtracking

An important problem which needs to be concerned is to find an effective way to keep correct data during backtracking. In our case we need to store correct domains of our variables. In every run of the backtracking algorithm we assign a value to a uninstantiated variable or return a solution if there is no variable left. An assignment of a value is performed by removing all other values from the domain of variable. After the assignment we perform arc consistency algorithm which further prunes domains of other variables. Finally if no variable has an empty domain, we run again the backtracking algorithm, otherwise we return false.

If an embedded run of the backtracking algorithm returns false, we need to restore the domains. For this purpose we store for every CSP variable a special linked list containing values deleted from its domain. The lists contain breakpoints to separate out particulars levels of backtracking.

- At the beginning every domain list is empty.
- Before every assignment of a value, we add a breakpoint in the end of all domain lists.
- If the function RESTORE is called, we refresh domains of all variables using convenient values from the domain lists.

The function RESTORE is a part of BACKTRACKING algorithm stated in Section 1.1.2. To refresh the domain of a variable x we iteratively remove the last element from its domain list and add this element into $Dom(x)$. We terminate after removing a breakpoint.

Chapter 3

Experiments and results

3.1 Experiments and results

In this section we provide description of experiments and their results. The main contribution of this work should be to explore the influence of usage of different versions of x-homomorphism or homomorphism and compare them in terms of runtime and accuracy. We provide comparisons of these different approaches and after that we provide a comparison of our implementation with state of the art algorithms for exploring structured data.

3.1.1 Description of datasets

The explored datasets come from various problem domains like toxicity prediction of small molecules, estimation of therapeutic potential of antimicrobial peptides and estimation of their adverse effects or searching for characterization of CAD documents. Their detailed description of most of them can be found in [9]. The description of Hexose is for example in [6].

- CAD
  - Class-labeled CAD documents describing product structures.
- CAMEL
  - Peptides labeled according to their antimicrobial activity.
- MUTA
  - Organic molecules marked according to their mutagenicity.
- RANDOM
  - Peptides labeled according to their antimicrobial activity
- PTC FM, PTC FR, PTC MM, PTC MR
  - Organic molecules marked according to their carcinogenicity for male and female mice and rats

- Hexose
  - Hexose-binding protein domains and non-Hexose-binding protein domains.

3.1.2 Optional parameters

Our implementation contains many parameters that can more or less determine the results. The LEARNING ALGORITHM contains for example the parameters *maxHypotheses*, *maxCandidateExamples* and *maxNegCovered*. Their influence is described in Section 2.1.1. It seems to be reasonable to set *maxHypotheses* to about tens of repetitions. The parameter *maxCandidateExamples* can be for example set to $\sqrt{2 \times |PositiveExamples|}$. But there is no strict recommendation for proper setting. We should only keep in mind that too high values of *maxHypotheses* and *maxCandidateExamples* can lead to overfitting and too low values to obtaining only poor and improper hypotheses. As we mentioned before, we always set *maxNegCovered* to zero in the experiments reported here..

Other options determining the behavior of the LEARNING ALGORITHM are whether to turn on or off the algorithms COMPONENT ELIMINATION and VARIABLE ELIMINATION. We can also choose whether we use homomorphism or bounded homomorphism to test which examples are covered. We can also choose which kind of ELEMENT ELIMINATION to use. Our implementation enables to choose whether to use the variants of ELEMENT ELIMINATION and ELEMENT ELIMINATION COMPLETE with additional reduction as described in 2.1.6. We always use the variants with the additional reduction.

Other important options are which formulation of CSP and which local consistency technique shall be used. We can also set how to obtain the threshold *minPositiveCovered*. We can either choose a percentage of positive examples or use the procedure SET MINIMUM COVERED to select the threshold automatically. In case of SET MINIMUM COVERED we choose number of repetitions of learning with fake labels and the number m, which means that we choose the m-th largest number of positive examples covered by a learned hypothesis with fake labels as our threshold. Other important parameters in the algorithm CLASSIFIER LEARNING are *innerRep* and *outerRep* and the choice of final selection of hypotheses from the learned set.

The concrete way how to set the parameters is described in Appendix.

In following sections we provide comparisons of usage of different CSP formulations: bounded homomorphism using GAC, bounded homomorphism using SAC and complete homomorphism. We test these approaches mainly on eight different datasets. Some of the runs were also tested on one additional dataset Hexose. The dataset Hexose is much more difficult than the other datasets because it contains larger structures. Therefore we did not perform on it all runs with various settings that we performed for the other datasets. We describe most of the results on this dataset separately in Section 3.1.6.

3.1.3 Cross-validation results

In this section we provide results for 10-fold cross-validation. We used the following common setting for all experiments in this section:

- $maxNegCovered = 0$

- $maxHypotheses = 30$
- Component elimination and Variable elimination are both performed
- $maxCandidateExamples = \sqrt{2 \times |PositiveExamples|}$
- $innerRep = 10$
- $outerRep = 3$
- Used Set minimum covered with 10 repetitions and 3rd highest value chosen as threshold

We changed other parameters to test their influence on accuracy and runtime. We run experiments with following combination of parameters:

1. Standard formulation, GAC consistency, bounded homomorphism, bounded version of Element elimination, in tables denoted as std GAC
2. Dual formulation, GAC consistency, bounded homomorphism, bounded version of Element elimination, in tables denoted as dual GAC
3. Standard formulation, SAC consistency, bounded homomorphism, bounded version of Element elimination, in tables denoted as std SAC
4. Dual formulation, SAC consistency, bounded homomorphism, bounded version of Element elimination, in tables denoted as dual SAC
5. Standard formulation, GAC consistency, complete homomorphism, element elimination complete, in tables denoted as complete

We test in experiments the influence of the above combinations with different maximum number of variables in one structure. This is determined by the parameter $maxVariables$ in algorithm LGG reduction algorithm. All experiments with one dataset were performed with an identical seed. A very important note is that we used a language bias in all our experiments. Therefore all experiments use at least bounded LGG w.r.t. the set of all structures complying with a language bias and not the standard LGG. So technically we use bounded LGG even if we use Element elimination complete.

At most 100 variables per structure

First we set the parameter $maxVariables$ to 100. The table 3.2 shows average runtime of complete learning and the table 3.1 accuracy of these experiments. The entry "$> 48h$" means that at least one of the folds did not finish in 48 hours. As we can see from the table 3.1, there is no difference in accuracy by using standard or dual formulation of CSP. The difference between accuracy by using SAC or GAC as x-prehomomorphism is usually approximately one percent. The usage of homomorphism and complete element elimination gives us slightly different accuracies then the bounded versions but the differences are still very small. Sometimes the complete version gives better results than the bounded but sometimes also worse.

If we look at the table 3.2, we can see that there is a difference in runtime of the different five scenarios. In all datasets except CAD and Hexose the version using complete homomorphism and ELEMENT ELIMINATION COMPLETE is the fastest. This result is really surprising since the homomorphism testing is an exponential-time procedure whereas the bounded homomorphism testing with GAC or SAC as x-prehomomorphism is a polynomial-time procedure. However these tests used structures reduced to contain at most 100 variables. It was therefore necessary to perform the same set of tests on structures containing more variables.

Dataset	std GAC		std SAC		dual GAC		dual SAC		complete	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
cad	0.8896	0.1273	0.8896	0.1273	0.8896	0.1273	0.8896	0.1273	0.9096	0.1085
camel	0.8609	0.1268	0.8609	0.1268	0.8609	0.1268	0.8609	0.1268	0.8400	0.1174
muta	0.7671	0.0866	0.7779	0.0858	0.7671	0.0866	0.7779	0.0858	0.7843	682.6488
ptc fm	0.6050	0.0659	0.6050	0.0581	0.6050	0.0659	>48h		0.5938	0.0553
ptc fr	0.6640	0.0514	0.6583	0.0482	0.6640	0.0514	>48h		0.6410	0.0456
ptc mm	0.6045	0.0428	0.5924	0,04447	0.6045	0.0428	>48h		0.6165	0.0630
ptc mr	0.5585	0.0748	0.5585	0.0748	0.5585	0.0748	>48h		0.5582	0.0763
random	0.8400	0.1101	0.8450	0.0832	0.8400	0.1101	0.8450	0.0832	0.8500	0.1054
hexose	0.6563	0.0896	–	–	0.6563	0.0896	–	–	0.6938	0.1299

Table 3.1: Results of 10-fold cross-validation, 100 variables, accuracy

Dataset	std GAC		std SAC		dual GAC		dual SAC		complete	
	mean	stdev	mean	stdev	mean	stdev	mean	stdev	mean	stdev
cad	14.80	6.95	650.66	649.14	37.79	21.43	941.70	1427.91	18.65	7.87
camel	179.51	43.17	4261.72	2165.88	151.13	112.67	6530.36	1413.77	101.39	56.94
muta	174.83	66.73	6306.08	1711.65	465.85	802.06	47467.87	22441.76	115.68	55.78
ptc fm	190.82	93.83	14133.44	6848.70	158.27	70.88	>48h		86.42	53.08
ptc fr	82.59	31.60	27520.04	19378.72	152.63	74.66	>48h		72.39	18.18
ptc mm	104.60	45.41	37166.81	34612.86	206.50	96.02	>48h		93.70	51.02
ptc mr	147.48	52.81	139.05	75.66	303.64	132.39	>48h		132.42	66.25
random	90.27	33.92	2159.16	2466.38	104.21	50.85	4522.39	3543.01	25.37	8.89
hexose	7515.82	1378.46	–	–	9588.09	2296.01	–	–	10093.15	4101.18

Table 3.2: Results of 10-fold cross-validation, 100 variables, runtime in seconds

At most 400 variables per structure

We performed the same experiments with higher number of variables in one structure. This time the limit was set to 400. Table 3.3 contains the accuracy obtained by the experiments. Unfortunately the time limit 48 hours was exceeded for all experiments using SAC as x-prehomomorphism. We can see that there is again no big difference in accuracy regarding complete homomorphism testing or bounded homomorphism using GAC. Sometimes the accuracy is slightly better for complete version sometimes for bounded. The results for standard and dual formulation of the problem using bounded homomorphism with GAC give again exactly the same results.

The results in Table 3.4 show the runtime for the experiments with 400 variables. Except Hexose and CAMEL datasets, the dual formulation performs much better than standard formulation in experiments using bounded homomorphism. In most cases the version using homomorphism testing and complete element elimination seems to be the best choice

regarding the average runtime. However we should notice that for datasets PTC FM and PTC MM there is a very large standard deviation, which indicates that the average runtime is not a reliable estimate of the real runtime. A better information about the runtime behavior can be provided by Figure 3.1 which represents runtime of all folds in cross-validation. The figure shows the median, the 25th and 75th percentiles and outliers. Note that the runtime is plotted in the logarithmic scale. For comparison Figure 3.2 provides the same results for the version with bounded homomorphism using GAC and standard formulation. Note that as opposed to the other datasets the runtime of the dataset Hexose was the best for bounded version using GAC and that the complete version exceeded the time limit 48 hours.

Dataset	std GAC		std SAC		dual GAC		dual SAC		complete	
	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.
cad	0.8774	0.1274	>48h		0.8774	0.1274	>48h		0.8773	0.0895
camel	0.8600	0.1265	>48h		0.8600	0.1265	>48h		0.8300	0.1252
muta	0.7479	0.14445	>48h		0.7479	0.1445	>48h		0.7638	0.1378
ptc fm	0.5701	0.0564	>48h		0.5701	0.0564	>48h		0.6018	0.0318
ptc fr	0.6725	0.0389	>48h		0.6725	0.0389	>48h		0.6611	0.0461
ptc mm	0.6399	0.0486	>48h		0.6399	0.0486	>48h		0.6308	0.0620
ptc mr	0.5788	0.0617	>48h		0.5788	0.0617	>48h		0.5989	0.0420
random	0.835	0.0784	>48h		0.835	0.1001	>48h		0.8350	0.1107
hexose	0.6500	0.0894			0.6500	0.0894			>48h	

Table 3.3: Results of 10-fold cross-validation, 400 variables, accuracy

Dataset	std GAC		std SAC		dual GAC		dual SAC		complete	
	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.
cad	926.64	365.20	>48h		533.37	144.92	>48h		472.57	75.98
camel	232.24	112.71	>48h		287.87	187.23	>48h		99.02	69.98
muta	5441.64	2302.10	>48h		3589.22	688.38	>48h		784.34	202.74
ptc fm	3525.02	1826.70	>48h		1751.26	659.55	>48h		2929.82	6711.55
ptc fr	3228.60	1475.40	>48h		2480.85	1222.40	>48h		1644.61	637.05
ptc mm	3031.51	1684.99	>48h		1716.96	582.79	>48h		558.13	361.84
ptc mr	4962.59	2917.32	>48h		3725.07	1529.02	>48h		4532.04	10075.75
random	83.53	97.91	>48h		46.08	18.81	>48h		35.76	19.37
hexose	18391.64	6208.34			24790.33	9336.32			>48h	

Table 3.4: Results of 10-fold cross-validation, 400 variables, runtime in seconds

No limit on variables per structure

For a complete overview about the performance of the used settings we performed a set of the same experiments this time with no limit on size of the explored structures. It was not meaningful to run those experiments for the versions using SAC since they exceeded the time limit already in experiments with limited size. From this set of experiments we did not gain many results because most of them exceeded either time limit or memory limit (denoted by OOM) which was usually 16 GB. Sometimes the runtime was exceeded by only one fold so maybe higher threshold can ensure that also versions with bounded homomorphism finish. Except the dataset CAMEL we obtained only results for the version using complete homomorphism. The results of accuracy are provided in Table 3.5. We can see that there is no improvement in accuracy compared to the experiments with the size restriction. Regarding the runtime we can look into Table 3.6. We can again note that the

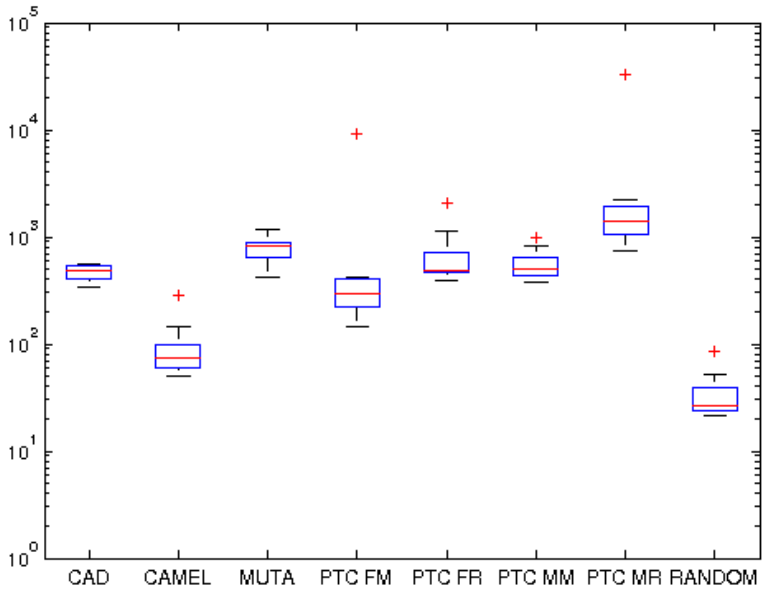


Figure 3.1: Runtime in seconds, 10 fold cross-validation, structure with 400 variables, homomorphism

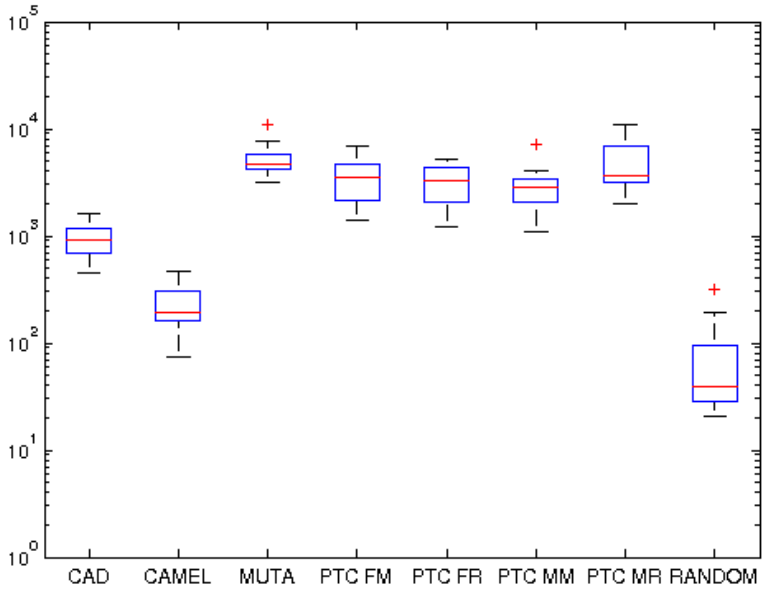


Figure 3.2: Runtime in seconds, 10 fold cross-validation, structure with 400 variables, bounded homomorphism using GAC, standard formulation

standard deviation is quite large. For this reason we again provide the box plot (Figure 3.3) of runtime of every fold which should show the runtime behavior better than mean and standard deviation.

Dataset	std GAC		dual GAC		complete	
	mean	st. dev.	mean	st. dev.	mean	st. dev.
cad	>48h		>48h		OOM	
camel	>48h		0.84	0.1075	0.85	0.1179
muta	>48h		>48h		0.7660	0.1574
ptc fm	>48h		OOM		0.5762	0.0578
ptc fr	>48h		>48h		0.6583	0.0560
ptc mm	>48h		>48h		>48h	
ptc mr	>48h		OOM		0.5843	0.0439
random	>48h		OOM		0.84	0.1022

Table 3.5: Results of 10-fold cross-validation. no variable limit. accuracy

Dataset	std GAC		dual GAC		complete	
	mean	st. dev.	mean	st. dev.	mean	st. dev.
cad	>48h		>48h		OOM	
camel	>48h		4370.96	11259.59	251.43	183.39
muta	>48h		>48h		6014.79	3094.77
ptc fm	>48h		OOM		4629.88	3114.78
ptc fr	>48h		>48h		11699.10	23772.89
ptc mm	>48h		>48h		>48h	
ptc mr	>48h		OOM		6056.39	5141.49
random	>48h		OOM		271.89	554.71

Table 3.6: Results of 10-fold cross-validation, no variable limit, time

3.1.4 Runtime analysis for random experiments

In this section we describe experiments more concerned with runtime of some procedures to compare the influence of bounded operations and different arc consistency techniques. Again we used a bounded LGG with respect to a set of structures complying with certain language bias. All experiments were performed with no limit on size of the structures.

In these experiments we always picked a random positive example from a dataset and computed its LGG with another positive example in combination with either Element elimination or Element elimination complete. We measured the runtime of the elimination algorithm. After reduction of the obtained structure we measured the runtime for homomorphism and bounded homomorphism between the obtained structure and all examples in the dataset. After that we performed LGG of this random structure with another positive example and again measured the runtime of reduction procedure and homomorphism decisions for all examples. After ten runs of LGG (LGG of ten examples) we always started with a new couple of positive examples and used their LGG for new measurements. In this way we obtained a large amount of samples on which we could measure average runtime of the operations.

The output of our measurements are two basic dependencies. The first is the dependence of runtime of an elimination algorithm on the size of a structure before and after performing it. And the second is the dependence of runtime of homomorphism (or bounded

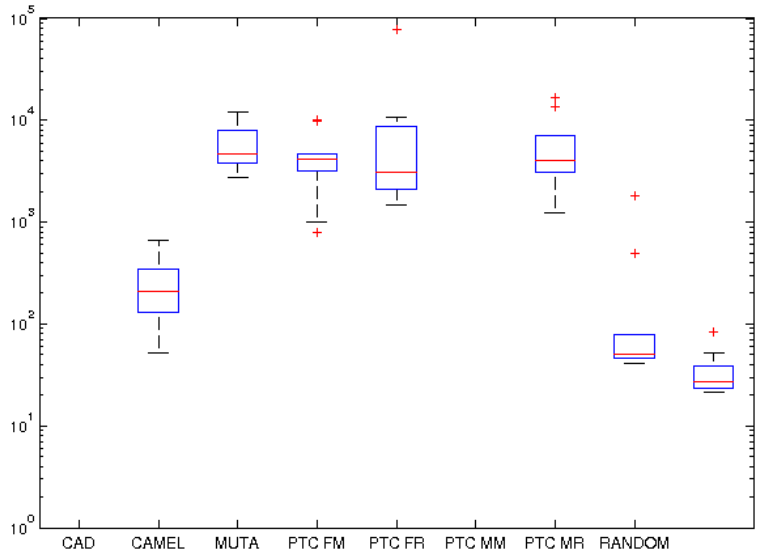


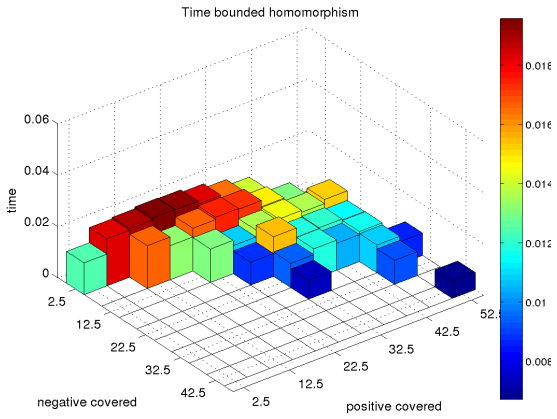
Figure 3.3: Runtime in seconds, 10 fold cross-validation, structure with 400 variables, homomorphism

homomorphism) on number of positive and negative examples covered by the explored hypothesis. We performed all experiments on one dataset with the same seed. Sometimes the experiments ran too long and therefore we did not always obtain the same number of random samples in all experiments. We usually removed a proper amount of samples in case of comparison of results of two experiments where one of them contained significantly more samples than the other to obtain exactly the same amount of samples in both sets.

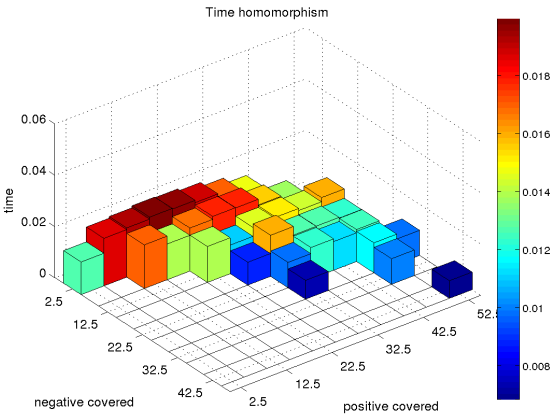
Homomorphism vs. bounded homomorphism

The most important question seems to be the runtime of homomorphism compared to bounded homomorphism. We expected the bounded homomorphism to be much faster than the complete homomorphism decision. However in cross-validation experiments it turns out that the scenario using complete homomorphism can perform better than the version using only bounded homomorphism. In the terms of CSP we basically measured the average time of GAC compared to complete solution of CSP using also GAC as its part. We always used the standard formulation in these experiments. Figures 3.4, 3.5 and 3.6 show the comparison of average runtime of homomorphism and bounded homomorphism testing between a random hypothesis and all examples in a dataset. Especially from the ratios of runtime we can see that the complete solution of a CSP is not much slower than only GAC. It means that the most of the time of CSP is taken by the GAC.

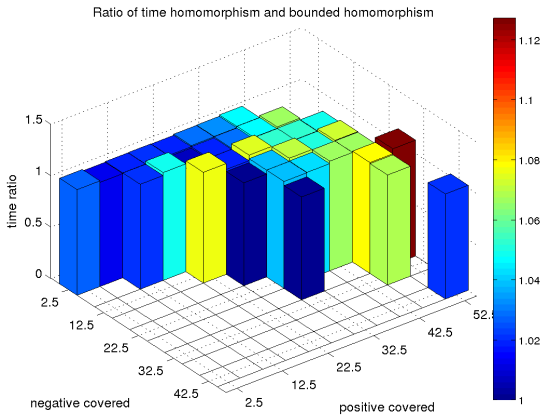
The other figures show the comparisons of runtime of Element elimination and Element elimination complete (Figures 3.7 and 3.8). Again GAC and standard formulation were used in the experiments. Note that the runtime is provided in logarithmic scale. Here we can see an interesting fact. The runtime is often much better for the complete ver-



(a) Bounded homomorphism runtime



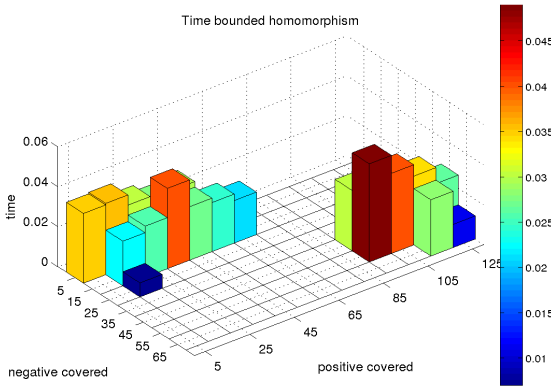
(b) Homomorphism runtime



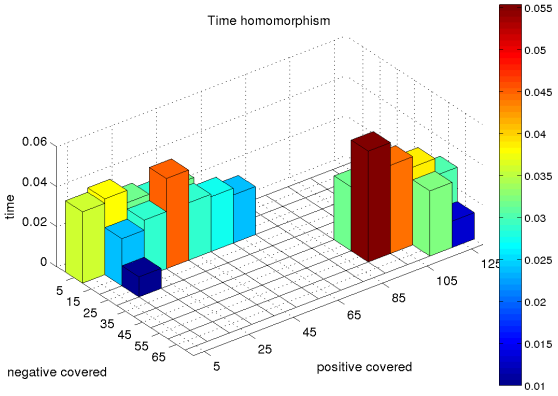
(c) Ratio of runtime: homomorphism/bounded homomorphism

Figure 3.4: Runtime in seconds dependence on number of positive and negative covered examples, CAMEL dataset, standard formulation, GAC in both prehomomorphism and homomorphism

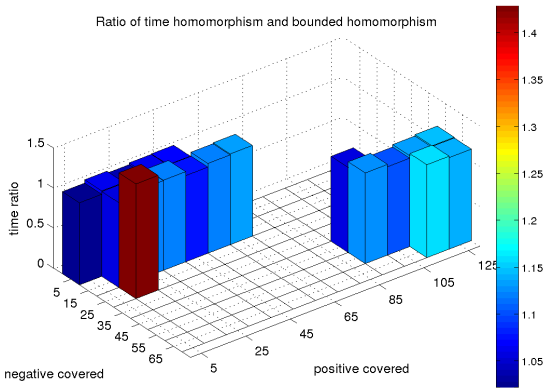
sion than for the bounded version. We could expect such a result since the cross-validation measurements were also faster in case of Element elimination complete. Such a result is surprising since the complete version performs repeatedly an exponencial-time CSP solution whereas the basic Element elimination performs repeatedly a polynomial-time procedure. However as we have seen in figures with homomorphism and bounded homomorphism comparison, the runtime is not much higher for homomorphism. In addition the reduction of CSP domains described in Section 2.1.6 is much more stronger for Element elimination complete.



(a) Bounded homomorphism runtime



(b) Homomorphism runtime

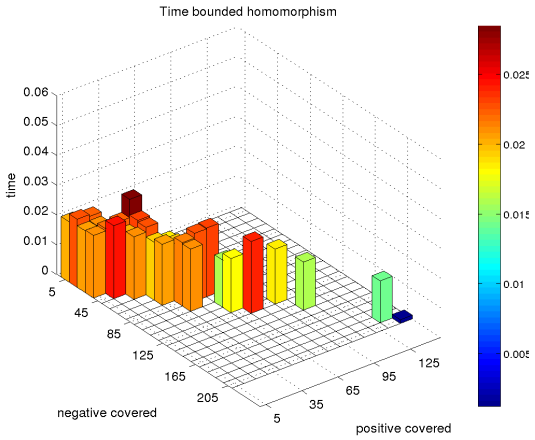


(c) Ratio of runtime: homomorphism/bounded homomorphism

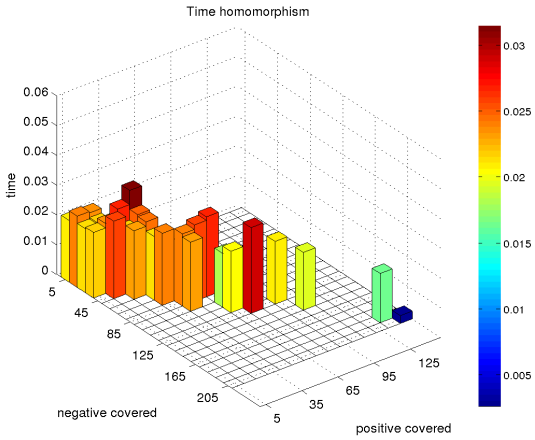
Figure 3.5: Runtime in seconds dependence on number of positive and negative covered examples, MUTA dataset, standard formulation, GAC in both prehomomorphism and homomorphism

My CSP solver vs. Choco solver

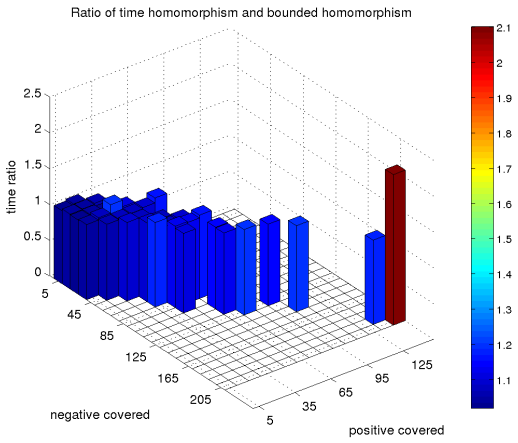
In our experiments with random hypotheses we also tested the speed of homomorphism decision using our implementation and the homomorphism decision using Choco solver, which is a widely used CSP solver. The results are shown in Figures 3.9, 3.10 and 3.11. As we can see in the figures our implementation is at least four times faster for CAMEL and PTC FM datasets and at least 2.5 times faster for MUTA dataset. These results show that our implementation of CSP solver is very effective. However our solver is designed only for CSP with table constraints as opposed to Choco which is able to solve much more general tasks.



(a) Bounded homomorphism runtime

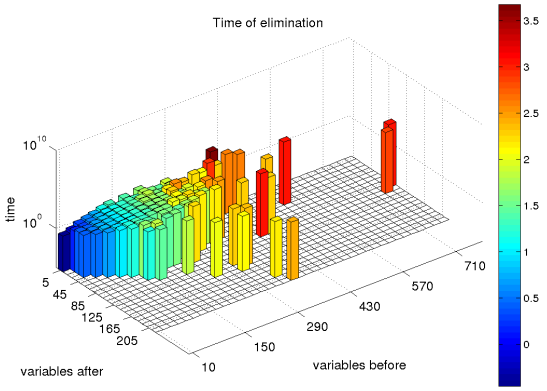


(b) Homomorphism runtime

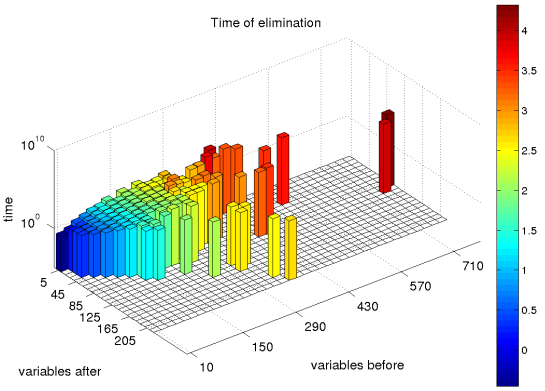


(c) Ratio of runtime: homomorphism/bounded homomorphism

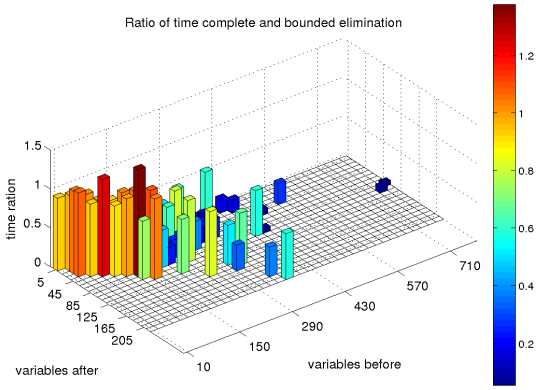
Figure 3.6: Runtime in seconds dependence on number of positive and negative covered examples, PTC FM dataset, standard formulation, GAC in both prehomomorphism and homomorphism



(a) Element elimination

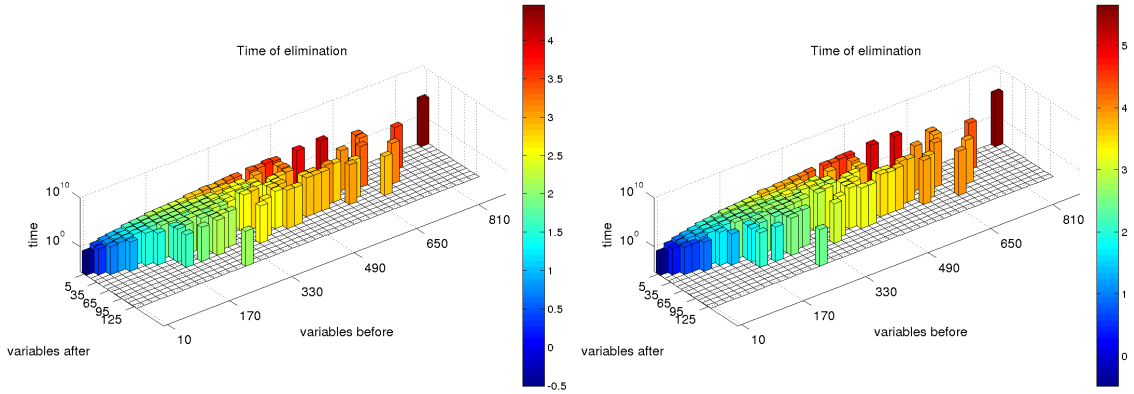


(b) Element elimination complete



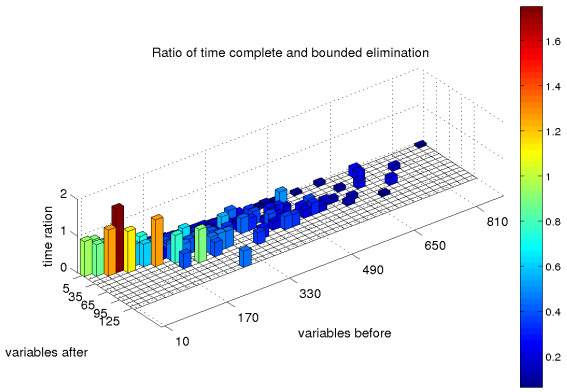
(c) Ratio of runtime: Element elimination complete/Element elimination

Figure 3.7: Runtime in seconds Element elimination complete, Element elimination, CAMEL dataset, standard formulation, GAC



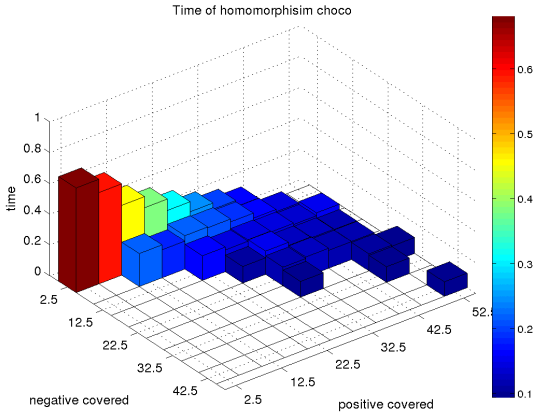
(a) Element elimination

(b) Element elimination complete

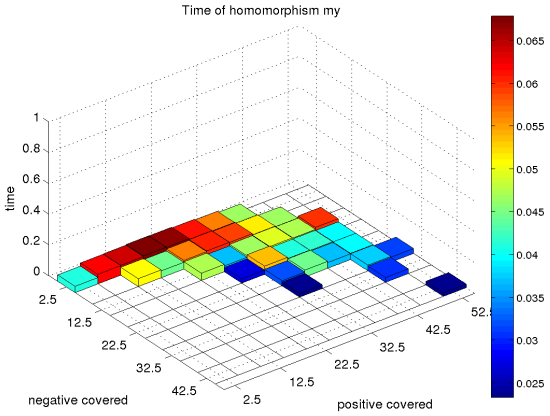


(c) Ratio of runtime: Element elimination complete/Element elimination

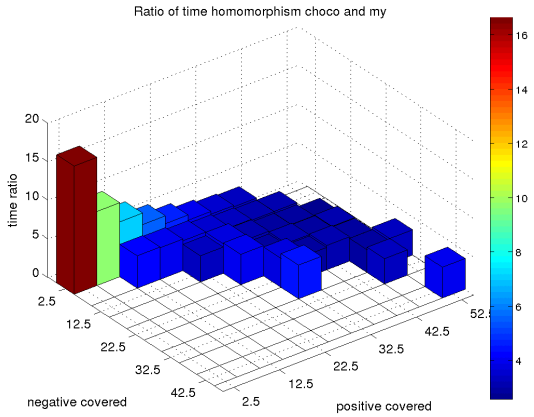
Figure 3.8: Runtime in seconds Element elimination complete, Element elimination, MUTA dataset, standard formulation, GAC



(a) Homomorphism using Choco solver

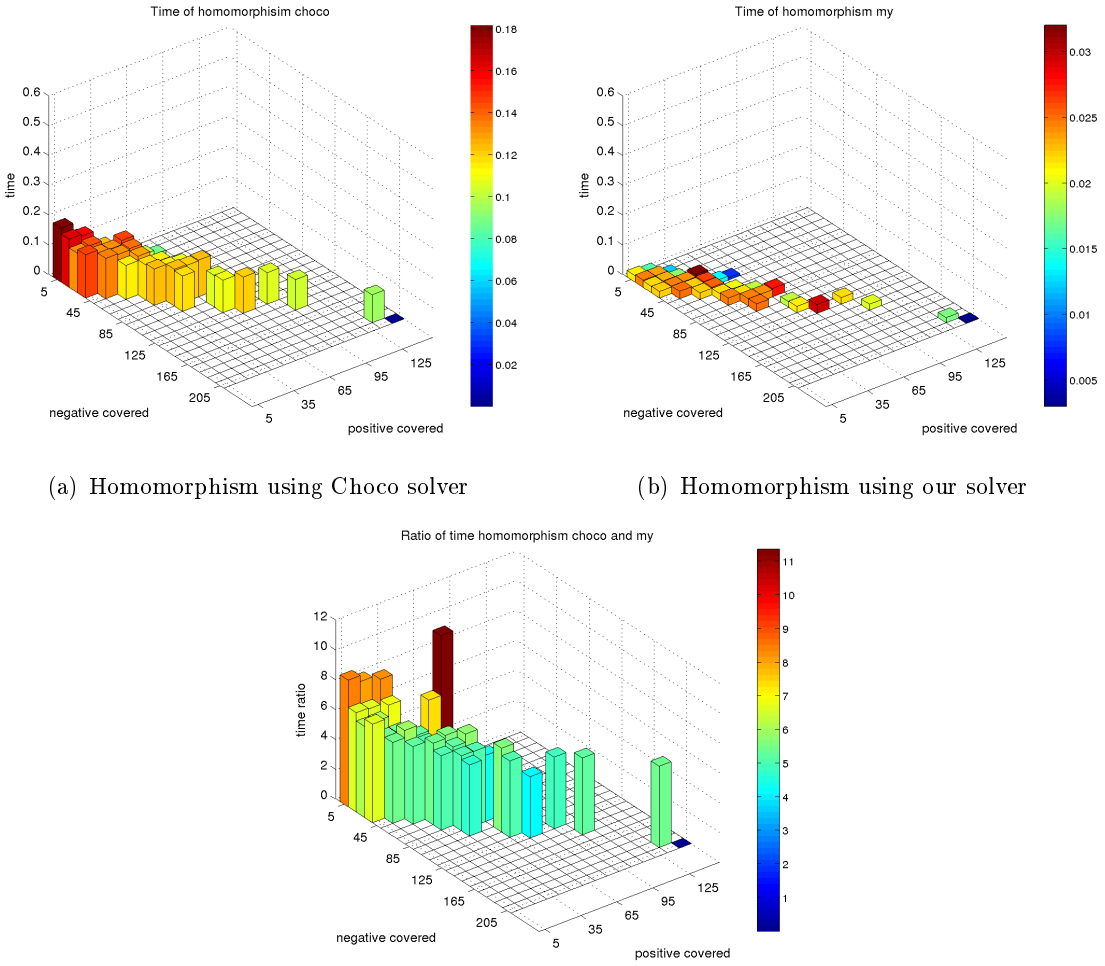


(b) Homomorphism using our CSP solver



(c) Ratio of runtime: homomorphism using choco/homomorphism using our implementation

Figure 3.9: Runtime in seconds dependence on number of positive and negative covered examples, comparison with Choco solver, CAMEL dataset, standard formulation, GAC in our solver

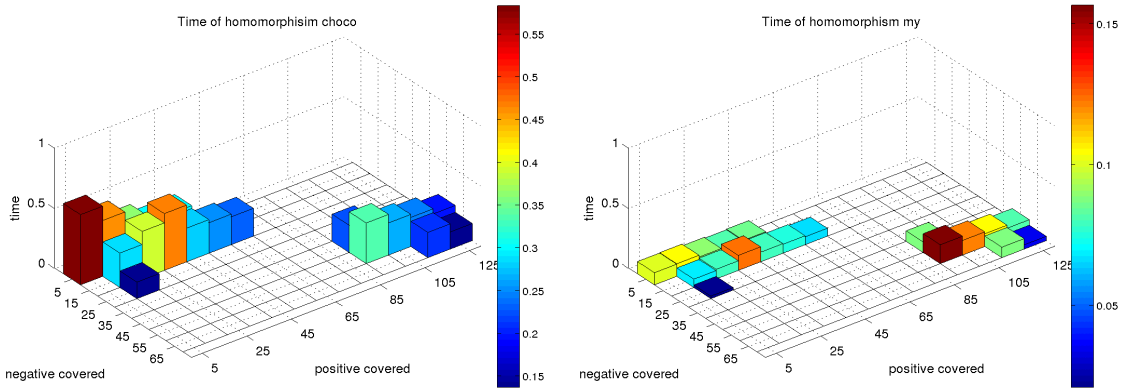


(a) Homomorphism using Choco solver

(b) Homomorphism using our solver

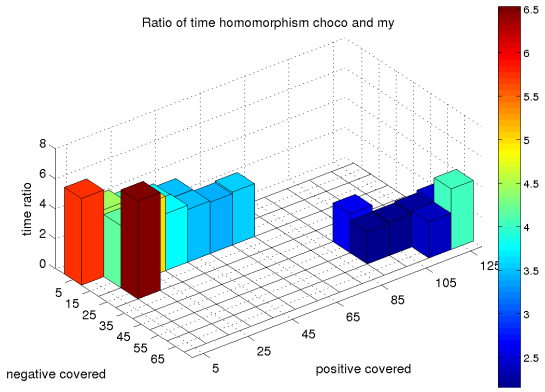
(c) Ratio of runtime: homomorphism using choco/homomorphism using our implementation

Figure 3.10: Runtime in seconds dependence on number of positive and negative covered examples, comparison with Choco solver, PTC FM dataset, standard formulation, GAC in our solver



(a) Homomorphism using Choco solver

(b) Homomorphism using our solver



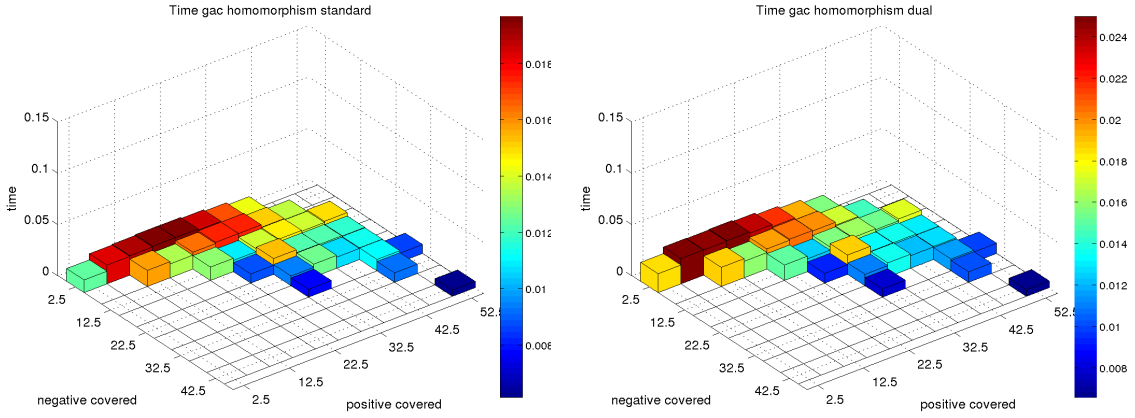
(c) Ratio of runtime: homomorphism using choco/homomorphism using our implementation

Figure 3.11: Runtime in seconds dependence on number of positive and negative covered examples, MUTA dataset, standard formulation, GAC in our solver

Standard vs. dual formulation for GAC as x-prehomomorphism

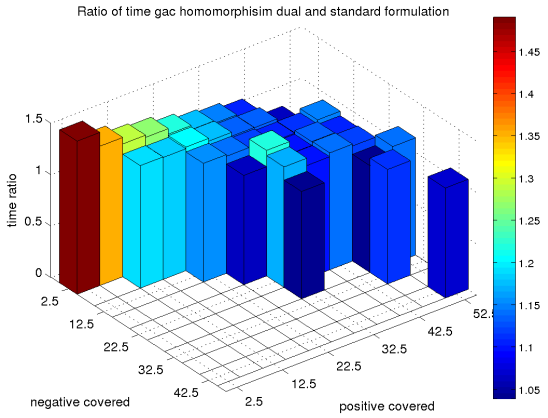
In this part we describe similar experiments for comparison of dual and standard CSP formulation in case of using GAC as x-prehomomorphism. The results for bounded homomorphism testing are in Figures 3.12 and 3.13. We can see that the dual formulation is a little bit slower for CAMEL dataset and usually equally fast or slower for PTC FM dataset.

The results for ELEMENT ELIMINATION are in Figures 3.14 and 3.15. We can see that the dual formulation is slightly slower for the CAMEL dataset, where the ratio of runtime is approximately between 1.2 and 2.2. For PTC FM dataset the results are similar with one exception where the ratio of runtime is about 30.



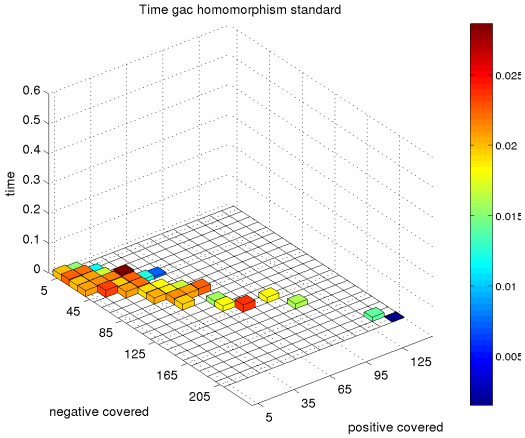
(a) Standard formulation

(b) Dual formulation

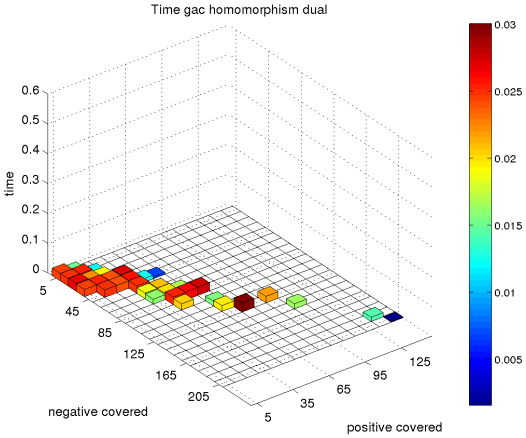


(c) dual formulation/standard formulation

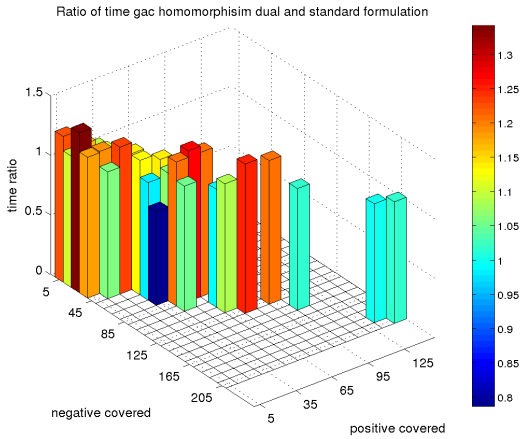
Figure 3.12: Runtime in seconds, dependence on number of positive and negative covered examples, bouned homomorphism using GAC, influence of standard and dual formulation, CAMEL dataset



(a) Standard formulation

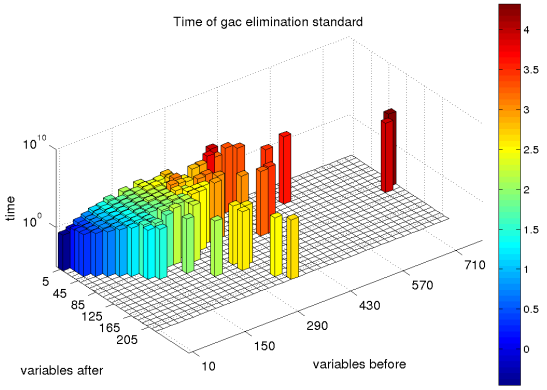


(b) Dual formulation

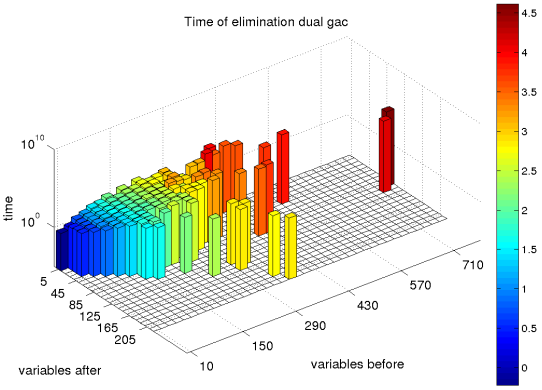


(c) Runtime ratio dual formulation/standard formulation

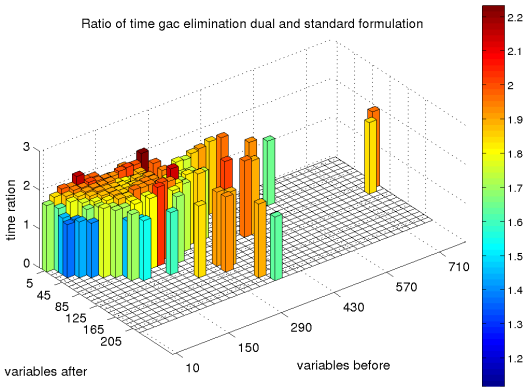
Figure 3.13: Runtime in seconds, dependence on number of positive and negative covered examples, bouned homomorphism using GAC, influence of standard and dual formulation, PTC FM dataset



(a) Standard formulation

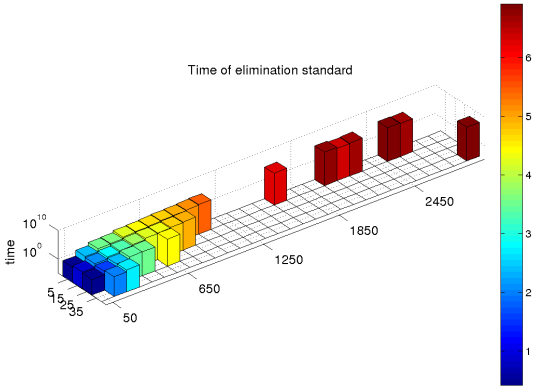


(b) Dual formulation

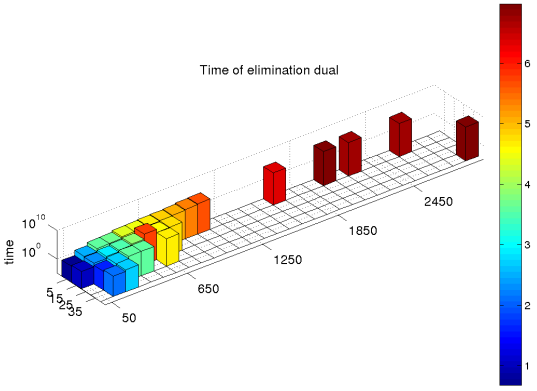


(c) Runtime ratio dual formulation/standard formulation

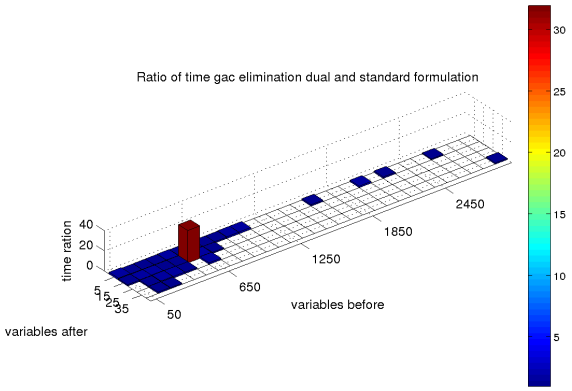
Figure 3.14: Runtime in seconds of Element elimination using GAC, dependence on number of variables before and after elimination, influence of standard and dual formulation, CAMEL dataset



(a) Standard formulation



(b) Dual formulation



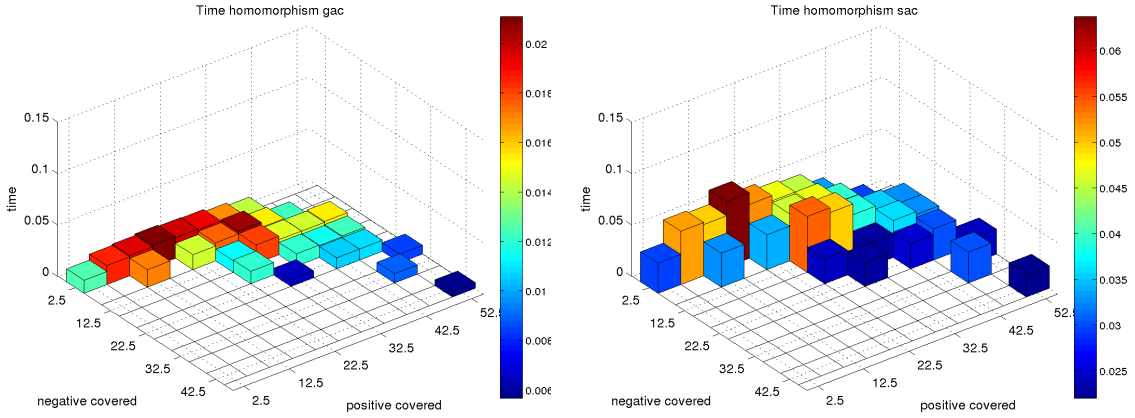
(c) Runtime ratio dual formulation/standard formulation

Figure 3.15: Runtime in seconds of Element elimination using GAC, dependence on number of variables before and after elimination, influence of standard and dual formulation, PTC FM dataset

GAC prehomomorphism vs. SAC as prehomomorphism, standard formulation

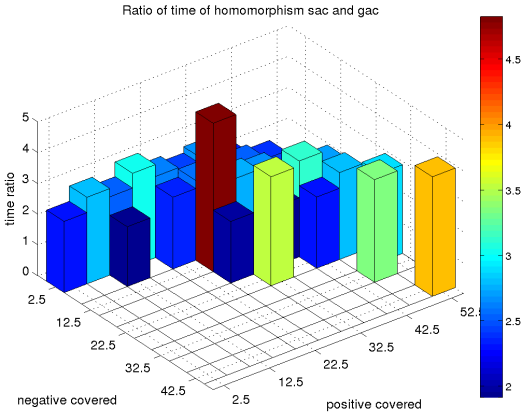
In this part we provide results of experiments for comparison of bounded homomorphism using GAC with bounded homomorphism using SAC. An example of results for CAMEL dataset is in Figure 3.16. We can see that the version with SAC is at leas two times slower than the one with GAC. The results are not surprising since SAC is based on repetitive calling of GAC.

The results for Element elimination are in Figure 3.17. We can see that in an extreme case the ratio of SAC and GAC can be up to 8000. However values approaching 2000 or 4000 are also quite often. These results can explain why we usually did not obtain results from cross-validation in reasonable time by using SAC as x-prehomomorphism.



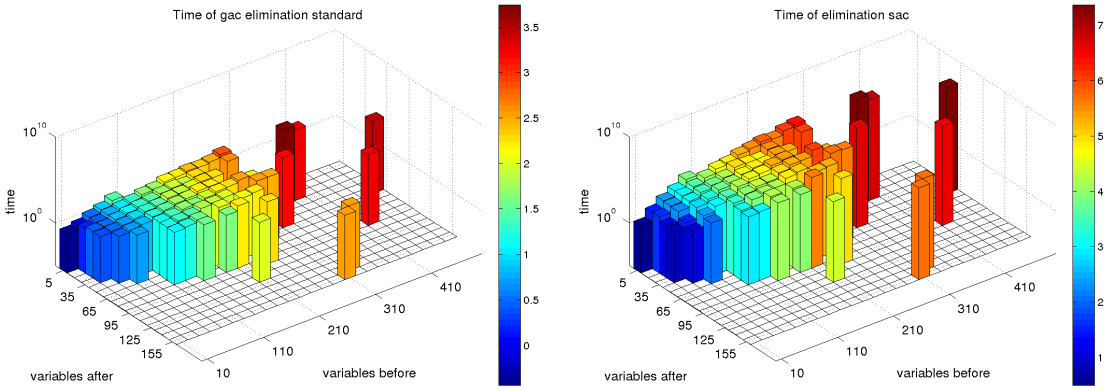
(a) Bounded homomorphism using GAC

(b) Bounded homomorphism using SAC



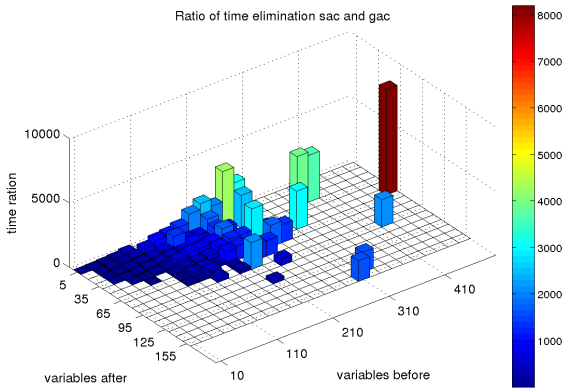
(c) Runtime ratio SAC/GAC

Figure 3.16: Bounded homomorphism runtime in seconds, dependence on number of positive and negative covered examples, standard formulation, influence of GAC and SAC, CAMEL dataset



(a) Elimination using GAC

(b) Elimination using SAC

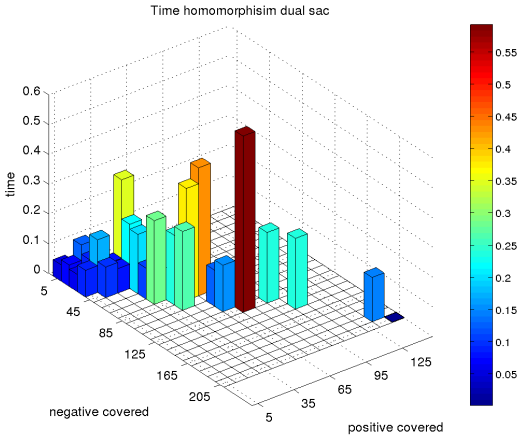


(c) Runtime ratio SAC/GAC

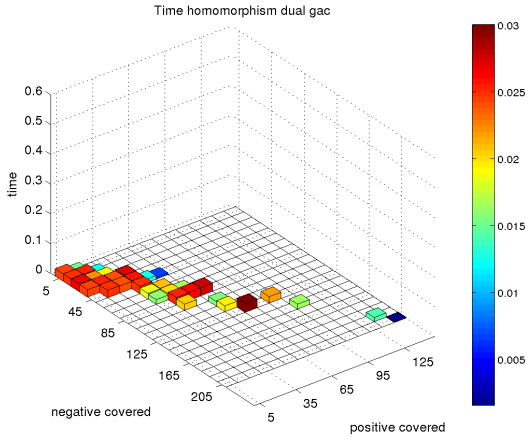
Figure 3.17: Runtime in seconds of Element elimination, dependence on number of variables before and after elimination, influence of GAC and SAC, standard formulation, CAMEL dataset

GAC prehomomorphism vs. SAC prehomomorphism, dual formulation

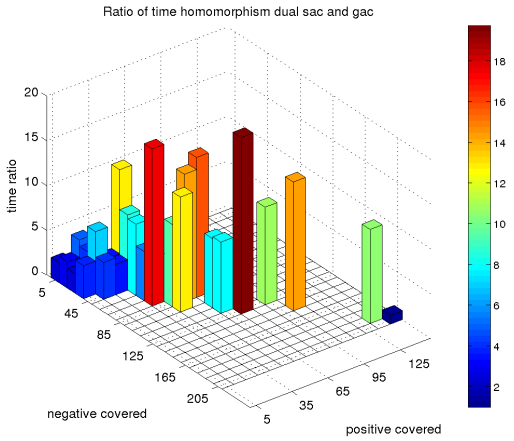
Finally we provide an example of experiment for comparison of bounded homomorphism using GAC and SAC in combination with dual CSP formulation. The results are in Figure 3.18. We can see similarly as in case of standard formulation that SAC is significantly slower than GAC.



(a) Bounded homomorphism using SAC



(b) Bounded homomorphism using GAC



(c) Bounded homomorphism using SAC/ bouned homomorphism using GAC

Figure 3.18: Runtime in seconds, dependence on number of positive and negative covered examples, dual formulation, influence of GAC and SAC, PTC FM dataset

3.1.5 Comparison with other learners

In this section we provide a comparison of accuracy of our classifier and the state-of-the-art tools for relational learning. The used learners were Aleph [25], nFOIL [12] and ProGolem [16]. We compare performance of our learner with the results from [9]. The parameters of all four systems were set so that their runtime would be about tens minutes per fold at most.

There are two columns in Table 3.7 for the learner ProGolem. In each column different value of maximum number of covered negative examples was set. In ProGolem1 the parameter was set to the default value and in ProGolem2 the parameter was set to 0.

Table 3.7 presents the results of 10-fold cross-validation. The values for our classifier are in the column denoted as My. The parameter setting was: standard formulation, bounded homomorphism using GAC and at most 100 variables in structure. These values can be also found in Table 3.1. A pairwise comparison of performance can be found in Table 3.8. The

values here mean number of wins/looses/ties of the classifier in row compared to the one in column. It can be seen from the two tables that our implementation reaches comparable accuracy to the other tools. In Table 3.2 average learning runtime per fold for every dataset for our learner can be found. We can see that the average runtime per fold is less than 200 seconds.

Dataset	My		Aleph		ProGolem1		ProGolem2		nFOIL	
	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.	mean	st. dev.
CAD	0.890	0.127	0.857	0.107	0.863	0.072	0.875	0.084	**0.969**	0.052
CAMEL	**0.861**	0.127	0.724	0.145	0.805	0.131	0.783	0.101	0.833	0.113
MUTA	0.767	0.087	0.608	0.089	0.665	0.045	**0.832**	0.070	0.766	0.096
PTC FM	0.605	0.066	**0.620**	0.033	0.591	0.071	0.619	0.069	0.602	0.087
PTC FR	0.664	0.051	**0.687**	0.039	0.650	0.083	0.658	0.047	0.670	0.065
PTC MM	0.604	0.043	0.598	0.050	0.607	0.082	0.595	0.036	**0.631**	0.088
PTC MR	0.558	0.075	**0.593**	0.045	0.549	0.054	0.567	0.074	0.573	0.071
RANDOM	0.840	0.110	0.860	0.061	**0.880**	0.063	0.810	0.094	0.830	0.059

Table 3.7: Results of 10-fold cross-validation, accuracy

	My	Aleph	progolem1	progolem2	nfoil
My	–	4/4/0	6/2/0	5/3/0	4/4/0
Aleph	4/4/0	–	3/5/0	5/3/0	4/4/0
progolem1	2/6/0	5/3/0	–	3/5/0	1/7/0
progolem2	3/5/0	3/5/0	5/3/0	–	2/6/0
nfoil	4/4/0	4/4/0	7/1/0	6/2/0	–

Table 3.8: Table of wins/looses/ties for 10-fold cross-validation

3.1.6 Hexose dataset

This dataset contains spatial structures of 80 Hexose-binding protein domains and 80 non-Hexose-binding protein domains. Positive and negative labels indicate which protein domains are capable of binding hexoses or not. For this dataset we used the language bias $d(x, \#, x, \#, \#)$. Here, the relation d has the form $d(V1, type1, V2, type2, distance)$. The first position contains an identifier of an atom, the second contains the type of the atom (reflecting also its position in the amino acid), the third position contains an identifier of another atom, the fourth its type and finally the fifth position contains the distance between the two atoms if it is less than 4 Angstroms. There are no tuples for pairs of atoms with a distance higher than 4 Angstroms in the dataset. Thank to this language bias we can easily transform the relational structures in this dataset into labelled graphs. Instead of one relational structure for every example, we obtain one labelled graph for every example. The approach is simple:

- There is one graph vertex for every atom.
- A vertex corresponding to an atom has a label with the type of the atom.
- There is an edge between two vertices if and only if the corresponding atoms belong to the same relational tuple. The edge is labelled by the distance between the two atoms.

It can be proven that the formulation using labelled graphs is equivalent to the formulation in terms of relational structures with language bias and of course also to the formulation in FOL. The exact form of the used operations can be found in [6]. Graph homomorphism is used instead of relational-structure homomorphism and graph product instead of LGG.

Previously, accuracy $67.5 \pm 10.5\%$ was reported in the original work concerned with this dataset [18]. Some results of our experiments were provided in section 3.1.3. The obtained accuracy was $65.6 \pm 90.0\%$ and $69.4 \pm 13\%$. Some of our experiments were already described in [6], where the accuracy of 10-fold cross-validation was $71.9 \pm 5.3\%$. In this experiment we used an older version of the algorithm, which was not able to set *minPositiveCovered* automatically, and slightly different settings than in Section 3.1.3. The main differences are:

- $maxHypotheses = 180$
- $maxVariables = 1500$
- $maxCandidateExamples = 4$
- *minPositiveCovered* set to 20% of positive examples
- $outerRep = 1$

There were also some little differences for example in using *innerRep* and we used homomorphism to updating score of a hypothesis. The examples in Hexose dataset are more complex (approximately 1000 edges per graph) than the examples in the other explored datasets and probably the higher values of some parameters can provide better results. However the better accuracy can be also influenced by randomness.

Figures 3.19 and 3.20 show examples of two learned graphs. Both of the two graphs covers 39 positive examples and no negative example.

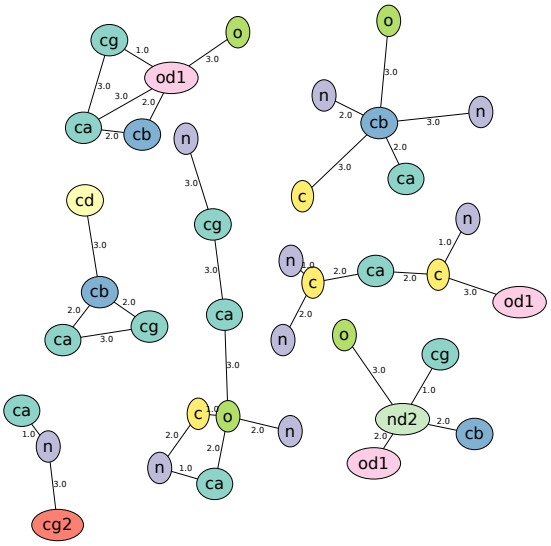


Figure 3.19: A graph structure which covers 39 positive examples and no negative example

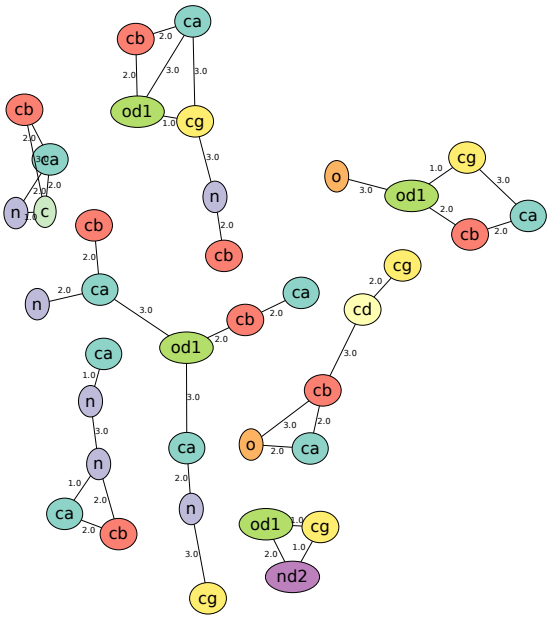


Figure 3.20: Another graph structure which covers 39 positive examples and no negative example

Conclusion

One of the goals of this thesis was to formulate methods of relational machine learning based on theoretical background from FOL into relational structures. Such formulation should be more appropriate for most scientific audience. We provided correspondence between operations and terminology of FOL and operations and terminology from the domain of relational structures. We formulated least general generalization defined by Plotkin for FOL clauses for relational structures and provided other useful equivalent transformations for example for FOL variables and constants. We also reformulated theoretical outcomes for bounded operations presented in [10] and [9] for relational structures.

The other goal of this thesis was to explore application of various techniques from constraint satisfaction problem in relational machine learning. We investigated the influence of using algorithms based on different local consistency techniques and compared them with algorithms based on complete CSP solution.

For this purpose we implemented a complex tool which can be used for relational learning. An important part of the tool is a CSP solver including two local consistency algorithms. Its implementation enables us to set many parameters of the learning process. The implementation also enables to change many parameters determining some basic principles of learning, like different local consistency techniques, different CSP formulations, turning on or off some enhancements which can contribute to better performance (Variable elimination, Component elimination, Set minimum covered, different settings of Element elimination). The implementation contains many algorithmic tricks and sophisticated data structures which contribute to its speed.

We performed several experiments to test the influence of different CSP techniques in our algorithms.

First we performed a set of cross-validation experiments. We found out that thank to effective implementation of our CSP solver the version of learning using in general exponential-time complete CSP solution is for almost all studied datasets faster that versions using only polynomial-time local consistency techniques, which is a very surprising result. However for example for the Hexose dataset, which contains more complex structures, the version using complete solver was slower than the versions using only local consistencies.

Our experiments also shown that different CSP formulations of our tasks have no influence on accuracy. The runtime can be better or worse for standard or dual formulation depending on the size of the problem and the studied dataset. We also found out that there is no significant difference in accuracy using algorithms based on GAC or SAC. However the runtime is much worse for the SAC versions. The versions using SAC usually had not finished within 48 hours even for small instances.

We also performed some experiments with randomly generated hypotheses to test the runtime of our procedures more precisely. We tested the dependence of runtime of various versions of homomorphism and bounded homomorphism, on number of covered positive and negative examples. We found out that the homomorphism testing based on CSP solution is only about approximately 1.1 to 1.4 times slower than bounded homomorphism testing based on arc consistency. We also showed that our CSP solver performs on these tasks about four time better than the widely used Choco solver. It turned out that bounded homomorphism is at least two times (but frequently more) slower using SAC than GAC.

In our experiments with random hypotheses we also tested the dependence of runtime of various version of Element elimination on the size of the processed structure before and after running it. It turned out that thanks to effective implementation the version using homomorphism testing (complete CSP solution) is very often much faster than the version using bounded homomorphism (arc consistency). The version with bounded homomorphism using dual formulation is usually slightly slower than the one using standard formulation.

The best choice for learning seems to be using version with homomorphism and Element elimination complete together with standard formulation and GAC. For some more complicated datasets like Hexose maybe bounded homomorphism and Element elimination is a better choice.

The findings presented in this thesis open many new research questions. It would be interesting to see whether the bounded operations are more suitable for datasets containing more complex examples as the experiments with the Hexose dataset suggest. It should be investigated if there exist further implementation improvements which can increase the speed of the algorithms with bounded operations. For example the actual structure of learning examples (e.g. whether they are acyclic) could be exploited. Another interesting question is how to select the hypotheses which should be used in the final classifier from all hypotheses obtained during learning. It would be also useful to find out whether higher settings of the parameters like $maxHypothesis$ contribute to better accuracy or whether it is enough to use only relatively low values, which also means faster learning. It could be also useful to investigate the best obtained hypotheses and find out whether they can be used in the scientific domain from which the data originate.

Contributions

In this work we bring several contributions. We reformulated the problems of relational learning widely studied in inductive logic programming into terminology of relational structures. Especially the definition of least general generalization for relational structures and a proposition how to deal with variables and constants from FOL in relational structures can be useful. This formulation should be more accessible for most of the scientific audience. Therefore it can contribute to use of these very useful methods in more domains.

The second contribution is the investigation of advantages and disadvantages of using bounded operations based on local consistency techniques from CSP in relational machine learning. We found out that an effective implementation of CSP solver used for homomorphism (θ-subsumption) decision, which is in general an exponential-time procedure, in a combination with an effective implementation of finding a core of a relational structure (θ-reduction) provides more effective learning than another theoretically exponentially faster approach using bounded operations. Our experiments indicate that there is no significant influence on accuracy when using different local consistency techniques as x-prehomomorphism.

Our learner is comparable in accuracy with state-of-the-art learners. Our experiments showed that the learner is able to produce good results quite fast.

We implemented an efficient CSP solver which can be used for solving tasks with general arity of constraints. This solver is even faster than the widely used Choco solver, as we showed in our experiments. However, as opposed to Choco our solver is designed only for table constraints. Thank to many enhancements our implementation performs well also in finding a core of a relational structure (θ-reduction) which is in general co-NP-complete task.

Appendix

3.2 How to use the learner

Format of the input data

The input data have to be stored in a textfile. Every row contains an example. The row starts with a label identifying whether the example is positive or negative. The labels can be for example + and −, but any strings without spaces can be also used. The label is followed by a space and the space is followed by the structure of the example. The syntax is the same as we used in some of our examples, i.e. the syntax inspired by Prolog. So one example can be written for instance in this way:

$$+\ atom(d59\_28),\ atom(d59\_34),\ bond(d59\_23, d59\_5, h\_3, c\_22, 1)$$

Parameters for learning

Various parameter settings have the general form `set(parameter_name,parameter_value)`, where every such item should be written on a unique row. The notation `set(parameter_name,'parameter_value')` is also possible. The use of the apostrophes is necessary when setting a parameter value containing commas otherwise the value will not be set properly. Comments are marked with %. Therefore any sequence on a row written after the character % is ignored.

Now we provide a description of concrete parameters:

1. `set(algorithm,'learning_one')` - Sets the type of algorithm which should be performed. Possibilities are:
   - `learning_one` - Performs one run of LEARNING ALGORITHM to obtain one hypothesis from examples of given dataset.
   - `learning_all` - Performs one complete learning procedure CLASSIFIER LEARNING to obtain a set of hypotheses.
   - `crossvalidation_one` - Part of crossvalidation, only one fold is processed. Useful for running folds separately on multiple computation sources. Script can be created automatically by using the option `create_files`.
   - `crossvalidation` - Performs a complete crossvalidation.
   - `create_files` - Creates new files where every file contains parameters for processing one crossvalidation fold. Every created file then contains `set(algorithm,learning_one)`.

2. `set(input_file,'file_name')` - Name (including path) of file with input examples. This is a compulsory parameter.

3. `set(seed_for_random,'105')` - Sets seed for random generator used in many randomized procedures.

4. `set(output_path,'name_of_output_directory')` - Sets complete path or name of a folder where outputs should be stored. Inside this directory a new directory "bull_output_*number*" is created. One script can contain multiple running instructions. The value of *number* indicates the order of run.

5. `set(filter,'dist(x,#,x,#,#), residue(x,#)')` - This filter determines the language bias. This is a due parameter in almost all running scenarios. The only exception is when `set(algorithm,'create_files')` is used. The marks # denote positions requiring elements satisfying *isConstant*. Here the apostrophes are necessary because filters always contain commas.

6. `set(positive_marker,'+')` - Indicates the label which is in the input file used for denoting positive examples. Default setting is +.

7. `set(positive_marker,'-')` - Indicates the label which is in the input file used for denoting negative examples. Default setting is −.

8. `set(consistency_type,'gac3')` - Indicates which type of consistency algorithm is used. Options are:

   - `gac3` - Default option.
   - `SAC`

9. `set(min_positive_rate,'0.01')` - Determines the parameter *minPositiveCovered* which is chosen as a rate of number of positive examples. For instance if we have 100 positive examples and set this parameter to 0.2, *minPositiveCovered* = 20. Default value is 0.1.

10. `set(parameter_learning,'random_labels')` - Indicates whether the algorithm SET MINIMUM COVERED is used for setting the parameter *minPositiveCovered* automatically. Options are:

    - `none` - Default setting. No algorithm is used for finding *minPositiveCovered*. The value for the parameter is obtained from `min_positive_rate`.
    - `random_labels` - The algorithm SET MINIMUM COVERED is used. This option turns off automatically the option `set(min_positive_rate,'0.01')`.

11. `set(max_hypotheses,30)` - Maximum hypotheses popped from the priority queue *Open* and explored in the algorithm LEARNING ALGORITHM. In text the parameter is denoted as *maxHypotheses*. Default value 150.

12. `set(max_variables_in_structure,'400')` - Maximum number of elements satisfying *isVariable* in one structure. This parameter is the input *maxVariables* of algorithm LGG REDUCTION ALGORITHM. Options:

    - An integer value.

- `no_limit` - No limit to the size, default parameter.

13. `set(max_negative_covered,'0')` - Maximum number of covered negative examples. Determines the parameter *maxNegCovered* in LEARNING ALGORITHM. Default value is 0.

14. `set(creator_type,dual)` - Determines which kind of CSP formulation is used for creating a CSP. Option:

    - `standard` - Standard formulation.
    - `dual` - Dual formulation, i.e. with only binary constraints, default value.

15. `set(elimination_type,bounded_WR)` - Determines which kind of element elimination is used. Options:

    - `bounded` - The version using only x-prehomomorphism is performed. The algorithm is used as stated in Section 2.1.6, in the basic form without additional removing of elements.
    - `bounded_WR` - This version uses x-prehomomorphism including removing more elements as described in section 2.1.6.
    - `complete` - This version uses homomorphism instead of x-prehomomorphism but without additional removing element.
    - `complete_WR` - This version uses ELEMENT-ELIMINATION COMPLETE, i.e. homomorphism testing and removing elements not present in solution of CSP.

16. `set(max_candidate_examples,'10')` - Maximum number of positive examples used for generalization of one hypothesis obtained from priority queue *Open* in LEARNING ALGORITHM. This parameter is called *maxCandidateExamples* in the text. Options:

    - An integer value.
    - `auto` - Sets the value to $\sqrt{2 \times |PositiveExamples|}$. Default value.

17. `set(component_elimination,true)` - Turns on or off the COMPONENT ELIMINATION algorithm. Can be set to `true` or `false`. Default value is `true`.

18. `set(variable_elimination,true)` - Turns on or off the VARIABLE ELIMINATION algorithm. Can be set to `true` or `false`. Default value is `true`.

19. `set(outer_repetition,'3')` - Count of repetitions of the algorithm CLASSIFIER LEARNING. Default value is 1.

20. `set(inner_repetition,'10')` - Count of repetitions of LEARNING ALGORITHM inside CLASSIFIER LEARNING and inside one outer repetition before termination if no good hypothesis is found. It is denoted as *innerRep* in CLASSIFIER LEARNING. Default value is 10.

21. `set(fold_number,10)` - Number of folds for cross-validation. This parameter has no meaning for running with `learning_one` and `learning_all`. Default value is 10.

22. `set(parameter_learning_repetition,'10')` - If `parameter_learning` is set to `random_labels` this parameter determines the number of repetitions of LEARNING ALGORITHM with fake labels. Default 10.

23. `set(learning_subsumption,bounded)` - Which type of homomorphism testing is used in LEARNING ALGORITHM. Options:
    - `bounded` - Bounded homomorphism is used. Default value.
    - `complete` - Complete homomorphism is used.
24. `set(complete_check,false)` - If this parameter is set to `true`, the homomorphism testing is used instead of LGG SCORE UPDATE even if `learning_subsumption` is set to `bounded`.
25. `set(final_selection,all_lists)` - Determines how to select final hypotheses from whole learned set. Options:
    - `all_lists` - All lists learned in every outer repetition in CLASSIFIER LEARNING are selected.
    - `best_list` - Only the best list learned in one of outer repetitions is selected. Default value.
26. `set(order_pl_random_labels,'3')` - This parameter is relevant only if `parameter_learning` is set to `random_labels`. In this case for example the third highest value of covered positive examples is used as the guess of *minPositiveCovered*. Default 1.

If we set one parameter multiple times in one file with different values, the most bottom value is used. At the end of our settings the expression `work(yes)` has to be written. It indicates that now the algorithm should start with current parameter values. This expression can be written in one file multiple times. For one run started with this expression only the settings written above it are used. If other settings are written under `work(yes)` they are used in the next run started with the next expression `work(yes)`. Not changed parameter values from the previous run are preserved.

If the option `create_files` is used, the parameters which should occur in the newly created files are set by using the word `copy` instead of `set`. So the input file can look for example this way:

```
set(algorithm,create_files)
set(seed_for_random,1202)
set(input_file,'../datasets/cad.txt')
set(output_path,'cad_source_files')

copy(outer_repetition,3)
copy(inner_repetition,10)
copy(filter,'cadFile(x), hasCADEntity(x,x), hasBody(x,x)')
copy(input_file,'../directory/datasets/cad.txt')
copy(output_path,'cad_output')
copy(elimination_type,bounded_WR)
copy(max_hypotheses,30)
copy(folds_number,10)
copy(learning_subsumption,bounded)
work(yes)
```

The parameters stated in the `copy` expressions occur in newly created files with keyword `set`. Some necessary parameters are written into the new files automatically. These parameters are:

```
set(folds_number,'10')
set(algorithm,'crossvalidation_one')
set(fold_id,4)
set(seed_for_random,922)
work(yes)
```

The file used for run of the algorithm can be found in the directory `target` in the directory `bull`. The algorithm can be run using the following command:
`java -cp bull-0.2-standalone.jar ida.bull.Main -source run_bull.txt`

3.3 Supplement

The directory `bull` contains the source codes which I implemented for purposes of this thesis (except the package `graphviz` used for drawing pictures of structures, which was provided by my colleague). Other directories: `src_treeliker`, `treeliker-cli`, `treeliker-gui`, `treeliker-lib` contain source codes of the software available at `http://ida.felk.cvut.cz/treeliker/index.html`. My implementation uses some classes from treeliker implementation. Therefore they are also a part of the supplement. The directory `diploma_thesis` contains Latex source codes of this thesis. The file `thesis_fuksova.pdf` is the basic pdf version of the thesis.

The source codes and their outputs are based on FOL terminology.

List of directories in `bull`:

- `src` - Contains source codes of the implementation.
- `target` - Contains the run file `bull-0.2-standalone.jar`.
- `examples_for_run`
  - `bull_datasets` - Directory with datasets.
  - Source scripts for run: `create_files_random.txt`, `learn_one_camel.txt`, `learn_one_hexose.txt`, `muta_crossvalidation.txt`, `random_crossvalidation.txt`. All paths in the scripts are set for the case that the run is performed from `target` directory.
  - `list of filters.txt` - List of filters which should be used for concrete datasets.

How to run from the command line:

1. Change the working directory to `bull/target`
2. Type for example: `java -cp bull-0.2-standalone.jar ida.bull.Main -source ../examples_for_run/learn_one_camel.txt`

Any of the source scripts provided can be used for run. The script `create_files_random.txt` creates source scripts for separate run of cross-validation folds.

Literature

[1] E. Alphonse and A. Osmani. On the connection between the phase transition of the covering test and the learning success rate in ilp. *Machine Learning*, 70(2-3):135–150, 2008.

[2] M. Arias, R. Khardon, and J. Maloberti. Learning horn expressions with logan-h. *Journal of Machine Learning Research*, 8:549–587, 2007.

[3] A. Atserias, A. Bulatov, and V. Dalmau. On the power of k-consistency. In *Proceedings of the 34th international conference on Automata, Languages and Programming*, ICALP'07, pages 279–290, Berlin, Heidelberg, 2007. Springer-Verlag.

[4] Ch. Bessiere, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artif. Intell.*, 172(6-7):800–822, April 2008.

[5] R. Debruyne and Ch. Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI (1)*, pages 412–417, 1997.

[6] A. Fuksová, O. Kuželka, and A Szabóová. A method for mining discriminative graph patterns. 2013.

[7] I. P. Gent, Ch. Jefferson, I. Miguel, and P. Nightingale. Data structures for generalised arc consistency for extensional constraints. In *In Proceedings of the Twenty Second Conference on Artificial Intelligence*, 2007.

[8] G. Gottlob, N. Leone, and F. Scarcello. On the complexity of some inductive logic programming problems. *New Gen. Comput.*, 17(1):53–75, 1999.

[9] O. Kuželka. *Fast Construction of Relational Features for Machine Learning*. PhD thesis, FEE, CTU in Prague, 2013.

[10] O. Kuželka, A. Szabóová, and F. Železný. Bounded least general generalization. In *ILP'12: Inductive Logic Programming*, 2013.

[11] F. Laburthe and N. Jussien. Choco constraint programming system, 2003-2005. Available at `http://choco.sourceforge.net/`.

[12] N. Landwehr, K. Kersting, and L. De Raedt. Integrating naïve bayes and FOIL. *Journal of Machine Learning Research*, 8:481–507, 2007.

[13] N. Landwehr, A. Passerini, L. De Raedt, and P. Frasconi. kFOIL: learning simple relational kernels. In *AAAI'06: Proceedings of the 21st national conference on Artificial intelligence*, pages 389–394. AAAI Press, 2006.

[14] J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.

[15] S. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, 1990.

[16] S. Muggleton, J. Santos, and A. Tamaddoni-Nezhad. Progolem: a system based on relative minimal generalisation. In *Proceedings of the 19th international conference on Inductive logic programming*, ILP'09, pages 131–148, Berlin, Heidelberg, 2010. Springer-Verlag.

[17] S. H. Muggleton and C. Feng. Efficient induction of logic programs. In *Proceedings of the First Conference on Algorithmic Learning Theory*, pages 368–381, Tokyo, Japan, 1990. Ohmsha.

[18] H. Nassif, H. Al-Ali, S. Khuri, W. Keirouz, and D. Page. An Inductive Logic Programming approach to validate hexose biochemical knowledge. In *Proceedings of the 19th International Conference on ILP*, pages 149–165, Leuven, Belgium, 2009.

[19] S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming (Lecture Notes in Computer Science)*. Springer, 1997.

[20] G. Plotkin. *A note on inductive generalization*. Edinburgh University Press, 1970.

[21] P. Prosser, K. Stegiou, and T. Walsh. Singleton consistencies. In *IN PROCEEDINGS CP'00*, pages 353–368, 2000.

[22] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[23] J. Santos and S. Muggleton. Subsumer: A prolog theta-subsumption engine. In Manuel V. Hermenegildo and Torsten Schaub, editors, *ICLP (Technical Communications)*, volume 7 of *LIPIcs*, pages 172–181. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[24] J. Sillito. Improvements to and estimating the cosf of backtracking algorithms for constraint satisfaction problems. Master's thesis, Faculty of Graduate Studies and Research, University of Alberta, 2000.

[25] A. Srinivasan. The aleph manual, 4th editions, 2007.

[26] A. Tamaddoni-Nezhad and S. Muggleton. The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. *Machine Learning*, (1):37–72.