Technische Universität Dresden

Master Thesis

θ-subsumption algorithms

October 29, 2007

Department: Computational Logic
Author: Georg Rammé
Supervisors: Prof. Steffen Hölldobler

Abstract

Since the early days of theorem proving and inductive logic programming several θ-subsumption algorithms have been developed. Recently, the focus came back to θ-subsumption due to its relevance in planning within first-order Markov Decision Processes. More than one formalism has been adopted to describe the language and the algorithms. Many experimental evaluations have been performed, but all focusing only on *some* algorithms and *a particular* domain. In this thesis, we will present the most popular θ-subsumption algorithms within a unified framework for a fair comparison. Further, we will describe the domains in which θ-subsumption is used and present a huge experimental evaluation on data from these domains. In addition, we give arguments for which algorithm is best suited depending on some basic parameters.

Acknowledgments

I would like to thank Prof Hölldobler for supervising me during the work on this thesis.

Special thanks go to Eldar Karabaev who has provided some state generators for the experimental section.

I would also like to thank Stefano Ferilli and Nicola Di Mauro for the code of FASϑ and for their help in getting it work.

Many thanks goes to Jérôme Maloberti who gave me valuable advice for the experimental evaluation and for providing the source code of Django.

I thank my brother Burkhard, for reviewing parts of this paper.

Many thanks go to Niko Baumann who reviewed parts of this paper and did not get tired of motivating me.

And many thanks to anonymous reviewers for useful comments.

Dresden, October 2007

Contents

1 List of symbols

1.1 Sets

$\in$	element
$\subseteq$	subset
$\cup$	union
$\cap$	intersection
$\emptyset$	empty set
$\vert A\vert$	cardinality (number of elements in set A)
$A \times B$	cartesian product of sets A and B
B^A	set of functions from A to B
2^A	set of all subsets of A

1.2 Logic

Σ	alphabet
Σ_V	set of variable symbols
Σ_F	set of function symbols
Σ_R	set of predicate symbols
$\mathcal{T}(\Sigma)$	set of terms based on Σ
$pred(l)$	predicate symbole of literal l
$var(C)$	set of variables occurring in clause C
$\vert C\vert$	size of the clause C
$[\,]$	the empty clause
$\wedge$	and
$\vee$	or
$\neg$	not
$\models$	logical entailment
θ, μ, σ	substitutions
ϵ	empty substitution
$\textsc{Dom}(\theta)$	domain of θ
$\textsc{Range}(\theta)$	range of θ
$\text{V}\textsc{Range}(\theta)$	variables of the range of θ
$l\lceil\pi\rceil$	argument of l at position π
$C\ \theta\text{SUBS}\ D$	C θ-subsumes D
$C\ \textsc{not}\ \theta\text{SUBS}\ D$	C does not θ-subsume D

2 Introduction

2.1 Motivation

θ-subsumption is a decidable restriction of logical implication [25]. Although being NP-complete [16], it is heavily used in domains such as

- Inductive Logic Programming ([23], [24]) as generality relation stating whether a hypothesis covers a training example;
- Frequent Pattern Discovery [5];
- Theorem Proving (like in *prover9*: a first-order automated reasoning system which is the successor to *otter*) as decidable but incomplete logical implication for remove redundant clauses; or
- Planning in a First-Order framework [14] to prune the search space by eliminating states that are more specific than others, or for calculating successor and predecessor states in the search process.

Since the early days of θ-subsumption, several algorithms have been developed to increase its computational efficiency. Nevertheless, these algorithms have not been compared exhaustively. There exists few articles that compare the older ones in a formal way on a theoretic basis. In their article [11], Gottlob and Leitsch establish a lower bound on the worst-time complexity of known algorithm upto that date (1985), namely ST and CL. Based on that study, they developed a new algorithm that does not suffer from the same weaknesses. Some experimental analysis have been carried out (e.g. in [28], [20], [9]), but they all focus their study either on particular application domains, or on particular aspects of the search domain of the θ-subsumption problems. For example, the system *django*, described in [20], is only compared to the system by Scheffer et al. [28]. Alongside, new approaches ([17], [29]) have been proposed and computational behaviour has only partially been studied.

2.2 A bit of History

θ-subsumption has first appeared in conjunction with the notion of *least general generalization* (lgg) which was introduced by Plotkin [25]. In some extend, it is the opposite of most general unification, that's why it is sometimes also called *anti-unification*. For example, given two atomic formulas $p(f(a), X)$ and $p(f(Y), b)$, unification computes the most general specialization $p(f(a), b)$ whereas anti-unification computes their most special generalization $p(f(Y), X)$. Plotkin extends this notion to clauses, and defines therefore the above mentioned notion of *θ-subsumption*: a clause C_1 generalizes a clause C_2, if C_1 θ-subsumes C_2 (The formalism and definitions will be given in the following sections). The least general generalization C of a set of clauses $\mathcal{S}$ is the *smallest* clause that generalizes every clause in $\mathcal{S}$. (*smallest* meaning here: for each clause D that generalizes every clause in $\mathcal{S}$, we have D generalizes C).

2.3 Main contributions

2.3.1 Unified framework

As θ-subsumption has been used in various domains, the formalisms used are not the same. One part of the work in this thesis, was to unify the formalisms in order to have the same representation and to be able to compare the different algorithms on a common basis.

2.3.2 Missing proofs completed

Most of the algorithms have been presented in articles in a rather short way, pointing out the main new feature, and spending few time in strict formal definitions. That's why they also only give proof sketches and informal correctness arguments, since the formal definition of e.g. the language used and assumption made are missing. In this thesis, the formalism will be clearly defined, so that we can work out the missing correctness proofs, and present them.

2.3.3 Experimental data and evaluation

The authors of the θ-subsumption algorithms have focused on only one specific domain of application each. So experimental data was often taken from *Inductive Logic Programming*. We extend the data by typical problem instances of other domain where θ-subsumption is used. Other domains are Theorem Proving and First-Order Planning.

The experimental evaluation carried out previously by other authors considered "newer" algorithms only, and neglected the existence of older algorithms. The main reason was that the implementations for older algorithms are not available. In order to make the experimental evaluation as exhaustive as possible, we implemented as much algorithms as possible for which no implementation existed.

2.4 Related work

Nearly all authors of new approaches to θ-subsumption have provided some experimental argument that showed the advantage of their method. They investigated the behaviour of their algorithm on some typical data of their focused domain.

Kietz and Lübbe [18] for instance, tested their algorithm on three aspects but staying in the ILP framework:

1. A toy domain: learning the definition of an *arch* (A curve with the ends down and the middle up, shaped like an inverted U),

2. Artificially constructed k-local Horn clauses of increasing size,

3. Testing *least general generalizations* against the example of three ILP-domains: krk [7], mesh [26], and speed [30].

Scheffer et al. [28] tested their implementation solely on the mesh domain [26]. To obtain different clauses of arbitrary size, they included only those nodes and edges that have only a certain distance from a randomly drawn starting node.

Maloberti and Sebag [20] have assessed the performance of their algorithm by a stochastic modeling. Stochastic modeling is widely used for validating CS (Constraint Satisfaction) algorithms. The modeling has been ported to θ-subsumption by Giordana and Saita [10]. They tested their implementation on artificial clauses generated such that they are in the *phase transition*, i.e., the hardest problems to solve because the probability of successful subsumption is difficult to know in advance. They also tested their implementation on the *mutagenesis* domain, which is widely used in ILP as benchmark problem. They compared they implementation again the implementation by Scheffer et al. only.

3 Preliminaries

3.1 Language

Definition 3.1 (Language) *The alphabet Σ of the language is the union of the following disjoint sets of symbols:*

- *The set Σ_V of variables, $\Sigma_V = \{Z, Y, X, ...\}$.*
- *The set Σ_F of function symbols. Each function symbol $f \in \Sigma_F$ is associated with an arity $n_f \in \mathbb{N}$, $\Sigma_F = \{f/n_f, g/n_g, ...\}$. If there is no ambiguity we write $f, g, ...$ instead of $f/n_f, g/n_g, ...$. 0-ary function symbols (function symbols with arity 0), are also called constant symbols, or constants.*
- *The set Σ_R of predicate or relations symbols. Each predicate symbol $p \in \Sigma_R$ is associated with an arity $n_p \in \mathbb{N}$, $\Sigma_R = \{p/n_p, q/n_q, ...\}$. If there is no ambiguity we write $p, q, ...$ instead of $p/n_p, q/n_q, ...$.*
- *The set of connectives $\{\neg\}$.*
- *The set of punctuation symbols "(", ",", and ")".*

Definition 3.2 (Term) *Terms are defined recursively as follows:*

1. *A variable is a term.*
2. *If f is an n-ary function symbol and t_1,...t_n are terms, then $f(t_1, ..., t_n)$ is a term.*

The set of all terms based on the alphabet Σ is noted $\mathcal{T}(\Sigma)$.

Definition 3.3 (DATALOG-term) *DATALOG-terms are defined as follows:*

1. *A constant symbol is a DATALOG-term;*
2. *A variable is a DATALOG-term.*

If there is no ambiguity, we will denote DATALOG-terms as terms.

Example $g(g(a, X), f(f(a)))$ is a term.
a and X are DATALOG-terms.

Definition 3.4 (Atom) *If p is an n-ary predicate symbol and $t_1, ..., t_n$ are terms then $p(t_1, ..., t_n)$ is an* atomic formula *or* atom*.*
An atom is called ground *if it does not contain any variable.*

Definition 3.5 (Literal) *If atom is an atom, then atom and $\neg$atom are literals, called respectively positive and negative literal.*
If l is a literal, we will write $pred(l)$ for the predicate symbol occurring in l.

Definition 3.6 (Argument position) *Let $p(a_1, ..., a_i, ..., a_n)$ be an atom. The* argument position *of a_i is i.*

If l is of the form $l = p(a_1, ..., a_i, ..., a_n)$ then $l\lceil i\rceil = a_i$. If $i > n$ then $l\lceil i\rceil$ is undefined.

The argument position of a literal is the argument position of the atom of the literal.

Example $p(X, Y)$ and $p(f(a), Z)$ are atoms, as well as (positive) literals. $\neg p(X, Y)$ is a (negative) literal. The term at argument position 2 in $p(X, Y)$ is $p(X, Y)\lceil 2 \rceil = Y$.

Definition 3.7 (Clause) *A* clause *is a set $\mathcal{C}$ of literals. The size of the clause is the cardinality of the set $\mathcal{C}$, written $|\mathcal{C}|$. The empty clause is denoted by $[\,]$.*

If $\mathcal{C}$ is a clause,

We note $var(T)$ the set of variables occurring in T, where T can be a term, a literal, a clause or a set of clauses.

Example $C = p(X, Y), p(f(a), Z)$ is a clause.

3.2 Definitions and properties

Definition 3.8 (Variable disjoint) *Let C and D be clauses. C and D are* variable disjoint *if $var(C) \cap var(D) = \emptyset$, i.e., no variable occurring in the first clause occurs in the second clause. In the literature, this is also often called* standardized apart.

In the following, we assume without loss of generality that clauses are variable disjoint.

Definition 3.9 (Substitution) *A* substitution *θ is a mapping from the set of variables into the set of terms which is equal to the identity mapping almost everywhere except for a finite set of variables, i.e., $\{X \in \Sigma_V \mid \theta(X) \neq X\}$ is finite. θ is represented as the finite set of pairs $\{X_1 \mapsto t_1, ..., X_n \mapsto t_n\}$, where $X_1, ..., X_n$ are different variables and for all $1 \leq i \leq n$ $X_i \neq t_i$. $\{X_1, ..., X_n\}$ is called the domain of θ, written $\textsc{Dom}(\theta)$ and $\{t_1, ..., t_n\}$ is the range of θ, written $\textsc{Range}(\theta)$. The set of variables of the range is written $\textsc{Vrange}(\theta)$.*

The empty substitution, i.e., the empty mapping, is denoted by ϵ.

Applying a substitution θ to a term s, denoted by $s\theta$, is the term obtained by simultaneously replacing each occurrence of a variable from $\textsc{Dom}(\theta)$ in s by the corresponding term in $\textsc{Range}(\theta)$.

Applying a substitution to an atom $p(t_1, ...t_m)$ is $p(t_1, ...t_m)\theta = p(t_1\theta, ..., t_m\theta)$. Applying a substitution to a clause $\mathcal{C} = a_1, ..., a_n$ is $\mathcal{C}\theta = a_1\theta, ..., a_n\theta$. Applying a substitution to a set $\mathcal{S} = \{l_1, ..., l_n\}$ of literal, is defined by applying θ to all the elements: $\mathcal{S}\theta = \{l_1\theta, ..., l_n\theta\}$.
We will note $\mathcal{T}(\Sigma)^{\Sigma_V}$ the set of all substitutions.

Example For example $\theta = \{X \mapsto f(Y), Y \mapsto a\}$ is a substitution. Applying this substitution to the term $t = g(X, Y)$ gives $t\theta = g(f(Y), a)$.

Definition 3.10 (Composition) *The composition $\theta\lambda$ of two substitutions θ and λ is defined as follows: for each term t, $t(\theta\lambda) = (t\theta)\lambda$.*

Proposition 3.11 (Idempotency of composition) *Let θ be a substitution. If $\textsc{Dom}(\theta) \cap \textsc{Vrange}(\theta) = \emptyset$ then the composition is idempotent, i.e., $\theta\theta = \theta$.*

Proof The proof is a proof by structural induction over terms.

- (Induction base) Let X be a variable.

- If $X \notin \textsc{Dom}(\theta)$ then $X\theta = X$. Thus $X(\theta\theta) \stackrel{def}{=} (X\theta)\theta = X\theta$.
- If $X \in \textsc{Dom}(\theta)$. Suppose $X \mapsto t$ in θ.
  Then $X(\theta\theta) \stackrel{def}{=} (X\theta)\theta = t\theta$.
  Since $t \in \textsc{Range}(\theta)$, $var(t) \in \textsc{VRange}(\theta)$ and thus $var(t) \notin \textsc{Dom}(\theta)$ by the assumption of the proposition.
  Hence, $t\theta = t$, and $X(\theta\theta) = t = X\theta$.

- (Induction step) Let f/n be a function symbol.
  Assume that the proposition holds for $t_1, ..., t_n$ (Induction hypothesis: IH).
  Then,

$$\begin{aligned} f/n(t_1, ..., t_n)(\theta\theta) &= f/n(t_1(\theta\theta), ..., t_n(\theta\theta)) \\ &\stackrel{IH}{=} f/n(t_1\theta, ..., t_n\theta) \\ &= f/n(t_1, ..., t_n)\theta \end{aligned}$$

Proposition 3.12 (Associativity of composition) *The composition of substitutions is associative, i.e., if θ, μ, σ are substitutions, then $\theta(\mu\sigma) = (\theta\mu)\sigma$.*

Proof Let t be a term and θ, μ, σ substitutions.

$$\begin{aligned} t(\theta(\mu\sigma)) &\stackrel{def}{=} (t\theta)(\mu\sigma) \stackrel{def}{=} ((t\theta)\mu)\sigma \\ t((\theta\mu)\sigma) &\stackrel{def}{=} (t(\theta\mu))\sigma \stackrel{def}{=} ((t\theta)\mu)\sigma \end{aligned}$$

Thus, $\theta(\mu\sigma) = (\theta\mu)\sigma$. ∎

Definition 3.13 (Strong compatibility) *Two substitutions θ_1 and θ_2 are called* strongly compatible *iff $\theta_1\theta_2 = \theta_2\theta_1$.*

If θ_1 and θ_2 are grounding substitutions, i.e. $\textsc{VRange}(\theta_1) = \textsc{VRange}(\theta_2) = \emptyset$ then θ_1 and θ_2 are strongly compatible iff no variable is assigned different terms in θ_1 and θ_2.

Definition 3.14 (Unifier, Most general unifier) *A substitution θ is a* unifier *of a set of terms $\mathcal{S}$ if $|\mathcal{S}\theta| = 1$.*

The definition extends in a natural way to atoms: a substitution θ is a unifier *of a set of atoms $\mathcal{S}$ if $|\mathcal{S}\theta| = 1$.*

A most general unifier *(mgu) θ is a unifier such that for all unifiers μ there exists a substitution λ such that $\mu = \theta\lambda$.*

Example Let $t_1 = f(X)$, $t_2 = f(g(Y,Z))$ then $\theta = \{X \mapsto g(Y,Z)\}$ is a most general unifier of $\{t_1, t_2\}$, but $\mu = \{X \mapsto g(a,b), Y \mapsto a, Z \mapsto b\}$ is a unifier but not a most general one.

Definition 3.15 (Matcher, Most general matcher) *Given two terms t_1 and t_1, a substitution θ is a* matcher *for t_1 over t_2 if $t_1\theta = t_2$.*

The definition extends to literals and clauses as follows:
Given two literals l and l', a matcher *for l over l' is a substitution θ such that $l\theta = l'$.*

Given two clauses C and D, a matcher *for C over D is a substitution θ such that $C\theta = D$.*

Similarly to the most general unifier, the most general matcher is defined: A most general matcher *θ is a matcher such that for all matchers μ there exists a substitution λ such that $\mu = \theta\lambda$.*

The set of all matchers for a literal $l_i \in C$ over some literal in D is denoted by $match(C, l_i, D) = \{\mu \in \mathcal{T}(\Sigma)^{\Sigma_V} \mid l_i \in C, l_i\mu \in D\}$

Example Let $C = p(X, Y), p(Y, Z), q(Z)$, $D = p(a, b), p(b, c), p(c, d), q(d)$. Then $match(C, p(X, Y), D) = \{\theta_1, \theta_2, \theta_3\}$, with

- $\theta_1 = \{X \mapsto a, Y \mapsto b\}$ matcher for $p(X, Y)$ over $p(a, b)$.
- $\theta_2 = \{X \mapsto b, Y \mapsto c\}$ matcher for $p(X, Y)$ over $p(b, c)$.
- $\theta_3 = \{X \mapsto c, Y \mapsto d\}$ matcher for $p(X, Y)$ over $p(c, d)$.

The next definition is the central definition of this thesis.

3.3 θ-subsumption

Definition 3.16 (θ-subsumption [25]) *Let C and D be clauses. C θ-subsumes D, written $C\ \theta SUBS\ D$ iff there exists a substitution θ such that $C\theta \subseteq D$.*

The set of all substitutions θ such that $C\ \theta SUBS\ D$ is called the answerset *of $C\ \theta SUBS\ D$.*

There exists also a stronger version of θ-subsumption, written $C\ \theta SUBS_{\leq} D$, defined as follows: $C\ \theta SUBS_{\leq} D$ iff $C\ \theta SUBS\ D$ and $|C| \leq |D|$.

Example Let $C = p(X, Y), p(Y, Z), q(Z)$, $D_1 = p(a, b), p(b, c), p(c, d), q(d)$ and $D_2 = p(a, b), p(b, c)$. Then $C\ \theta\mathrm{SUBS}\ D_1$ by substitution $\theta = \{X \mapsto b, Y \mapsto c, Z \mapsto d\}$, and not $C\ \theta\mathrm{SUBS}\ D_2$.

There are three different problems that will be considered in this thesis:

- Decision Problem: Given two clauses C and D, check whether $C\ \theta\mathrm{SUBS}\ D$. A problem instance of this kind will be written in short "$\exists?\theta.C\ \theta\mathrm{SUBS}\ D$".
- One-Solution Problem: Given two clauses C and D, find one substitution θ such that $C\ \theta\mathrm{SUBS}\ D$, if it exists. A problem instance of this kind will be noted "$One\theta?.C\ \theta\mathrm{SUBS}\ D$"
- Function Problem: Given two clauses C and D, find all substitutions (the answerset) $\theta \in \mathcal{T}(\Sigma)^{var(C)}$ such that $C\ \theta\mathrm{SUBS}\ D$. A problem instance of this kind will be noted "$All\theta?.C\ \theta\mathrm{SUBS}\ D$".

Example Consider the clauses

- $C = p(X, Y), p(Y, Z), q(T)$ and
- $D = p(a, a), p(a, c), p(c, d), q(a), q(b)$.

The answer to the decision problem "$\exists?\theta.C\ \theta\mathrm{SUBS}\ D$" is YES.

The answer to the one-solution problem "$One\theta?.C\ \theta\mathrm{SUBS}\ D$" is for example $\theta_1 = \{X \mapsto a, Y \mapsto a, Z \mapsto c, T \mapsto a\}$ or $\theta_2 = \{X \mapsto a, Y \mapsto c, Z \mapsto d, T \mapsto a\}$...

The answer to the function problem "$All\theta?.C\ \theta\mathrm{SUBS}\ D$" is the set of substitutions containing:

- $\theta_1 = \{X \mapsto a, Y \mapsto a, Z \mapsto c, T \mapsto a\}$
- $\theta_2 = \{X \mapsto a, Y \mapsto a, Z \mapsto c, T \mapsto b\}$
- $\theta_3 = \{X \mapsto a, Y \mapsto c, Z \mapsto c, T \mapsto a\}$
- $\theta_4 = \{X \mapsto a, Y \mapsto c, Z \mapsto c, T \mapsto b\}$

θ-subsumption is defined syntactically. But for understanding why it has been defined, we have to take a short look at the semantics of clauses, and how it is related to θ-subsumption.

The semantics of clauses are defined through interpretations.

Definition 3.17 (Interpretation) *An interpretation $\mathcal{I}$ is a set of ground atoms.*

Example The following are interpretations:

- $\mathcal{I}_0 = \emptyset$, the empty interpretation
- $\mathcal{I}_1 = \{p(a, b)\}$
- $\mathcal{I}_2 = \{p(a, b), q(b, c)\}$

Definition 3.18 (Model) *Let M be an interpretation. M is a model for an atom a, written $M \models a$, if $a \in M$, otherwise M is not a model for a, written $M \not\models a$.*
M is a model for negative literal $\neg a$, written $M \models \neg a$, iff $M \not\models a$.
M is a model for clause $C = a_1, ..., a_n$, written $M \models C$, iff $M \models a_1$ or ... or $M \models a_n$.

Definition 3.19 (Logical entailement) *Let C and D be clauses. C entails D (or equivalently D is a logical consequence of C), written $C \models D$ iff $M \models C$ implies $M \models D$, i.e., every model for C is a model for D.*

Proposition 3.20 ([12]) *Let C and D be clauses. If $C \models D$ then $C\ \theta SUBS\ D$.*

The converse is not true, as shown be the following popular example from [24].

Example Let $C = p(f(X)), \neg p(X)$ and $D = p(f(f(X))), \neg p(X)$.
Then C NOT θSUBS D but $C \models D$.

This shows that θ-subsumption is a sound but incomplete restriction of logical entailment.

Proposition 3.21 *Let C and D be clauses. Let D' be the clause D where all variables have been skolemized, i.e., all variables have been replaced by new (occurring nowhere else) constant symbols. Then, $C\ \theta SUBS\ D$ iff $C\ \theta SUBS\ D'$ and the set of all solutions $\mathcal{S}'$ for $C\ \theta SUBS\ D'$ can be obtained from the set of all solutions $\mathcal{S}$ for $C\ \theta SUBS\ D$ by skolemizing all the variables from the set $var(S) \cap var(D)$.*

Definition 3.22 (Subsumption algorithm) *A* subsumption algorithm *takes as input two clauses C and D and, depending on the type of problem that the algorithm is intended to solve, it produces as answer:*

- *"YES" or "NO", if it solves the Decision Problem;*
- *A substitution θ such that $C\ \theta SUBS\ D$ by θ or "NO" if it solves the One-Solution Problem;*
- *A set $\mathcal{S}$ such that for each $\theta \in \mathcal{S}$ we have $C\ \theta SUBS\ D$ by θ ($\mathcal{S}$ is empty, if no such θ exists), if it solves the Function Problem.*

Proposition 3.23 (Eisinger [8]) *A clause C θ subsumes a clause D iff there is an n-tuple $(\theta_1, ..., \theta_n) \in \times_{i=1}^{n} match(C, l_i, D)$, where $n = |C|$, such that all θ_i are pairwise strongly compatible.*

Example Consider the clauses

- $C = p(X, Y), p(Y, Z)$ and
- $D = p(a, c), p(c, d)$.

Then the carthesian product $\times_{i=1}^{n} match(C, l_i, D) =$

$$\underbrace{\left\{ \begin{array}{l} \theta_{1,1} = \{X \mapsto a, Y \mapsto c\} \\ \theta_{1,2} = \{X \mapsto c, Y \mapsto d\} \end{array} \right\}}_{match(C,p(X,Y),D)} \times \underbrace{\left\{ \begin{array}{l} \theta_{2,1} = \{Y \mapsto a, Z \mapsto c\} \\ \theta_{2,2} = \{Y \mapsto c, Z \mapsto d\} \end{array} \right\}}_{match(C,p(Y,Z),D)}$$

The substitutions of the tuple $(\theta_{1,1}, \theta_{2,2})$ are pairwise strongly compatible (since in both Y is mapped to c), so C θSUBS D.

Definition 3.24 (Deterministic Subsumption [18]) *Let C and D be clauses. C deterministically θ-subsumes D, written $C\ \theta SUBS\ _{DET} D$, by $\theta = \theta_1\theta_2...\theta_n$ iff there exists an ordering $C' = l_1, ..., l_n$ of C such that for all $i \in \{1..n\}$, there exists exactly one θ_i such that $\{l_1, ..., l_i\}\theta_1...\theta_i \subseteq D$.*

Theorem 3.25 ([18]) *Given two clauses C and D, solving the problem $C\ \theta SUBS\ _{DET} D$ can be computed with $O(|C|^2 \cdot |D|)$ basic unification attempts.*

Most of the algorithms can only solve θ-subsumption problems on DATALOG-clauses. This is motivated by the fact that non-DATALOG-clauses can be transformed (*flattened*) into DATALOG-clauses such that θ-subsumption is preserved.

The idea is that for every function symbol f of arity n, a new predicate f_p of arity $n + 1$ is introduced, where the first n arguments are the same as for the function and the last is the result of the function. Formally, we have the following definition and proposition.

Definition 3.26 (Flattening predicate (adapted from [27])) *The* flattening predicate *f_p associated with the function symbol f of arity $n \geq 1$* [1] *is the predicate of arity $n + 1$ defined by:*

$$f_p(X_1, ..., X_n, X) \equiv X = f(X_1, ..., X_n)$$

The variable X is called the output argument *of f.*

[1] In the original version, constant symbols where not treated separately, thus removing the additional condition $n \geq 1$ for the arity of the function symbol.

Definition 3.27 (Flattened clause [27]) *Let C be a clause. The flattened clause flat(C) is obtained from C by exhaustively applying the following rule: Replace any occurrence of $f/n(t_1, ..., t_n)$ by a new variable X and add the literal $f_p/n+1(t_1, ..., t_n, X)$ to the clause.*

Example Let $C = p(f(f(X))), q(f(X))$ be a clause. The application of the transformation rule yields:

$$\frac{\dfrac{p(\underline{f(f(X))}), q(f(X)).}{p(X_1), q(\underline{f(X)}), f_p(\underline{f(X)}, X_1).}}{p(X_1), q(X_2), f_p(X_2, X_1), f_p(X, X_2).}$$

So, flat$(C) = p(X_1), q(X_2), f_p(X_2, X_1), f_p(X, X_2)$.

Proposition 3.28 ([27]) *Let C and D be clauses.*
C $\theta SUBS$ D iff flat(C) $\theta SUBS$ flat(D).

Proposition 3.28 allows us to flatten every clause before testing it for θ-subsumption. That's why, in the rest of this thesis, we will only talk about DATALOG-clauses.

4 State-of-the-art

4.1 Overview of the algorithms

In Table 1, the different algorithms under study are presented. The first part shows which problem the algorithm can solve. The second part states the restrictions on the language or on the definition of θ-subsumption. In the third part, the feature used by the algorithms are shown.

	Problem			Restrictions				Feature											
	"$\exists?\theta.C$ θSUBS D"	"$All\theta?.C$ θSUBS D"	"$One\theta?.C$ θSUBS D"	General terms	DATALOG-terms only	θSUBS $\leq$	θSUBS	Binary resolution	Successive θ_i	Components	1-local	GC_1 / 1-SIG	GC_n	2-SIG	Object Context (OC)	Substitution Graph (SG)	Layered SG	Progressive SG construction	Binary CSP
CL	x			x		x		x											
ST	x			x		x			x										
DC	x			x		x				x									
KL	x			x			x			x	x								
GC		x	x		x		x					x	x			x			
Django	x				x		x					x		x					x
AllTheta	x	x	x		x		x					x	x				x	x	
FASϑ		x			x		x			(1)									
OC		x			x		x								x				

Table 1: Overview of the algorithms. (1) means that the algorithm assumes that the input clauses have been preprocessed with that feature

4.2 CL [3]

The CL algorithm was one of the first θ-subsumption algorithms developed by C.-L. Chang and R. C.-T. Lee in 1971. θ-subsumption was essentially use as a technique for reducing the search space in theorem provers, due to the combinatorial explosion of generated clauses.

The CL algorithm uses a special resolution strategy to solve the θ-subsumption problem.

Let $D_{gr} = l_1, ...l_n$, then $\neg D_{gr} = \neg l_1, ..., \neg l_n$. In the case C θSUBS D holds, then CL derives a contradiction from $C \wedge \neg D$; in the other case, it does not.

Definition 4.1 (Binary resolvent) *Let $C = l_1, l_2, ...l_n$ and $D = l'_1, l'_2, ..., l'_m$, and let there be i and j ($1 \leq i \leq n$, $1 \leq j \leq m$) such that l_i is a positive literal and l_j is a negative literal and l_i and l'_j are unifiable with unifier θ, then $R = l_1\theta, ..., l_{i-1}\theta, l_{i+1}\theta, ..., l_n\theta, l'_1\theta, ...l'_{j-1}\theta, l'_{j+1}\theta, ...l'_m\theta$ is a* binary resolvent *of C and D. Note that R can be the empty clause [ ].*

Algorithm 1: CL

Input: C, D_{gr}: two clauses
Output: **true** if C θSUBS D, **false** otherwise
begin
  $U \leftarrow \{C\}$;
  while $[] \notin U$ **and** $U \neq \emptyset$ **do**
    $U \leftarrow \{$ binary resolvents of C_1 and C_2 $|C_1 \in U, C_2 \in \neg D_{gr}\}$;
    if $[] \in U$ **then**
      SUBS $\leftarrow$ **true** ;
    end
    SUBS $\leftarrow$ **false** ;
  end
end

Theorem 4.2 *([11]) If $n \geq m$ then $u_{CL}(n, m, k) \geq k(k+1)^{m-2}(m+k)$, otherwise if $n < m$ it holds $u_{CL}(n, m, k) \geq k(k+1)^{n-1}(k(m-n)+m)$,*

Remark: the algorithm here is a slightly modified version of the original one presented in [3] since binary resolution was used instead of full resolution. In [11], it is shown that the worst-case complexity is reduced when using only binary resolution.

4.3 ST [32]

ST has been developed by Rona B. Stillman in 1973. It was used in theorem proving like the CL algorithm. Although faster than CL in numerous cases, it suffers from being dependent on the order of literals in the clauses.

The ST algorithm is based on the generation of successive substitutions θ_i.

Let $C = l_1, ..., l_n$. If θ_i has been generated, we have for all $j \leq i$, $l_j\theta_i \in D$. Then, ST tries to find a $\theta_{i+1} = \theta_i\mu$ such that for all $j \leq i+1$, $l_j\theta_{i+1} \in D$. If such a θ_{i+1} cannot be found, backtracking is applied. The run time of ST is essentially dependent on the ordering of literals in C.

Let $C = l_1, ..., l_n$ and $D = l'_1, ..., l'_m$. We define a Boolean function $unify(L, M)$, which returns *true* iff L and M are unifiable; if l and l' are unifiable, then a variable, mgu(l,l'), which represents the most general unifier of l and l' is defined.

The Stillman algorithm then consists in the application of the boolean function ST (see below) on the input $(1, 1, \epsilon)$; $C = l_1, ..., l_n$ and $D = l'_1, ..., l'_m$ are to be considered as global to ST.

Algorithm 2: ST

```
function ST(i, j, θ);
begin
    let a be a variable local to ST ;
    if j > |D| then
        return false ;
    else
        a ← j;
        while not unify(l_i θ, l'_a) and a ≤ |D| do
            a ← a + 1
        end
        if a > |D| then
            return false ;
        else
            μ_i ← mgu(l_i θ, l'_a);
            if i = |C| or ST(i + 1, 1, θμ_i) then return true ;
            else return ST(i, a + 1, θ);
        end
    end
end
```

Theorem 4.3 ([11]) *The worst-case unification complexity of ST is:*

- $u_{ST}(n, m, k) = \frac{k(k^m-1)}{k-1}$, *if* $n \geq m$ *and* $k > 1$
- $u_{ST}(n, m, k) = \frac{k(k^{n+1}-1)}{k-1} + k^{n+1}s$, *if* $n < m$ *and* $k > 1$, *with* $s = min(m - n - 1, k - 1)$

4.4 DC [11]

In 1985, G. Gottlob and A. Leitsch analysed the complexity of existing algorithms and developed a new one with a better worse case complexity. The main feature was that θ-subsumption problems were decomposed in independent smaller problems, thus reducing the overall complexity.

The formal definition of the algorithm follows.

Definition 4.4 *([11]) The* literal graph $L(C) = (V_{L(C)}, E_{L(C)})$ *of a clause* $C = l_1, \ldots, l_n$ *is defined by the set of vertices* $V_{L(C)} = \{l_1, \ldots, l_n\}$ *the set of literals in* C *and the set of edges* $E_{L(C)}$ *such that* $(l, l') \in E_{L(C)}$ *iff* $l, l' \in C, l \neq l'$ *and there is a variable occurring both in* l *and* l'.

Example Let

$$
\begin{aligned}
C = & p(X, a), p(b, X), q(X), \\
& p(Z_1, Z_2), \\
& p(U, V), p(V, W), p(W, T), p(T, U)
\end{aligned}
$$

The literal graph of C is $L(C)$ shown in figure Fig. 1.

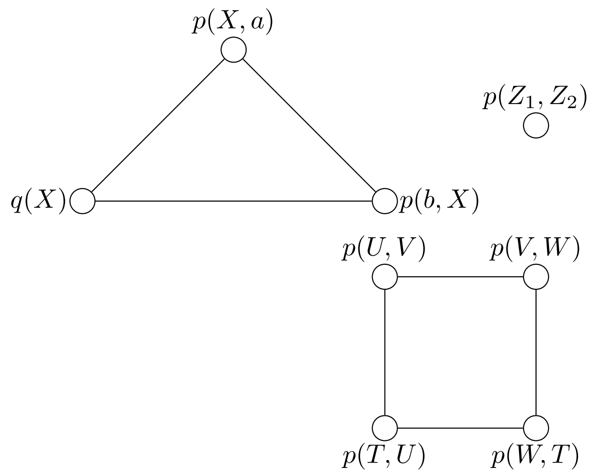

Figure 1: Literal graph of the clause C.

Definition 4.5 (connected components (graph theory)) *In an undirected graph, a* connected component *or* component *is a maximal connected subgraph.*

In an undirected graph G, two vertices u and v are called connected if G contains a path from u to v.

Generally, the clause C is decomposed into different independent groups of subclauses. These groups can be identified with the connected components of the graph $L(C)$.

In figure Fig. 1, there are 3 components, namely:

- $p(X,a), p(b,X), q(X)$
- $p(Z_1, Z_2)$
- $p(U,V), p(V,W), p(W,T), p(T,U)$

In [11] it is proved that if C is a clauses where $L(C)$ has the connected components $G_1, ..., G_k$ then C θSUBS D iff for all $i = 1, ..., k : V_{G_i}$ θSUBS D.

Definition 4.6 *A clause is called* simple *if for all components G_i of $L(C)$ it holds that $|var(V_{G_i})| \leq 1$ or $|V_{(G_i)}| = 1$.*

In other words, a clause is simple if each of the components of its literal graph has either only one variable or consist of only one vertex.

The algorithm DC (division into components) works as follows. First we test C_{simp} θSUBS D for the maximal subclause C_{simp} of C, which is simple (the graph $L(C_{simp})$ itself may consist of several components) because this test is polynomial. If G_i is a component such that V_{G_i} is not simple, we select a literal $L_{top} \in V_{G_i}$ having the maximal number of variables that also occur in other literals. (By storing the clauses appropriately, L_{top} can be selected in linear time with respect to $|V_{G_i}|$.) More exactly, we define L_{top} as the first literal L in C such that L has a maximal number of variables that also occur in other literals and $var(\{L\}) \geq 2$ (L exists because V_{G_i} is not simple).

After selection of L_{top} from V_{G_i}, we proceed as follows: If a substitution θ is found such that $L_{top}\theta \in D$, we process the clause $V_{G_i}\theta - \{L_{top}\}\theta$, subjecting it to the same analysis as defined above (recursion).

Algorithm 3: DC

```
function DC(C, D)// Tests the Predicate C θSUBS D
Input: C, D: two clauses
Output: true if C θSUBS D, false otherwise
begin
    construct L(C);
    identify C_simp and let G_simp be the connected components of
    L(C_simp);
    if not C_simp θSUBS D then
        return false // we apply a polynomial decision algorithm
            e.g.  ST
    else if For All G ∉ G_simp TC(V_G, D) then  return true ;
    else  return false ;
end
function TC(E, D) // TC means "test components"
Input: E: a set of literals
D: a clause
Output: true if E θSUBS D, false otherwise
begin
    select L_top from E;
    a ← 1;
    repeat
        while a ≤ |D| and not unify(L_top, K_a) do a ← a + 1;
        if a > |D| then sub ← false ;
        else
            if |E| = 1 then sub ← true ;
            else
                μ ←mgu(L_top, K_a);
                if DC(Eμ − {L_top}μ, D) then sub ← true ;
                else sub ← false ;
            end
        end
        a ← a + 1;
    until a > |D| or sub ;
    return sub;
end
```

4.5 KL [18]

In 1994, Kietz and Lübbe brought the idea by Gottlob and Leitsch to a further level. They studied the complexity of θ-subsumption by looking at *loosely connected* componentes, which they call *k-locals*. They show that θ-subsumption "is efficiently computable for some resonably small k" [18]. Formally, k-locals are defined as follows:

Definition 4.7 (based on [18], corrected) *Let* $C = C_{DET}, C_{NONDET}$ *and*

D be two clauses, with C_{DET} the maximal subclause of C which deterministically θ-subsumes C and $C_{NONDET} = C \setminus C_{DET}$. $LOC_i \subseteq C_{NONDET}$ is a non-determinate local of C iff $(var(LOC_i) \setminus var(C_{DET})) \cap var(C_{NONDET} \setminus LOC_i) = \emptyset$ and there does not exist a $LOC_j \subset LOC_i$ (*) which is also a non-determinate local of C. A non-determinate local LOC_i is a k-local for a constant k iff $k \geq \min(|(var(LOC_i) \setminus var(C_{DET}))|, |LOC_i|)$. A clause is k-local iff every non-determinate local is a k-local.

In other words, a non-determinate local is a subclause which do not share variables with the rest of the clause, except with literals that can be matched deterministically.

In the original version of definition 4.7, the set inclusion marked by *(*)* was $LOC_j \supset LOC_i$, which is obviously wrong since the one and only nondeterminate local of D_{NONDET} would then be D_{NONDET} itself.

Example Let $C = r(X,Y), p(Y,Z), p(Z,T), (T,Z), p(U,V), q(V,W), p(M,N)$ and $D = r(a,b), p(b,c), p(b,d), q(a,b), q(b,d)$.

Then $C_{DET} = p(X,Y)$ and
$C_{NONDET} = p(Y,Z), p(Z,T), p(T,Z), p(U,V), q(V,W), p(M,N)$.
The non-determinate locals are:

$$LOC_1 = p(Y,Z), p(Z,T), p(T,Z) \quad (1)$$
$$LOC_2 = p(U,V), q(V,W) \quad (2)$$
$$LOC_3 = p(M,N) \quad (3)$$

LOC_1 is a 2-local, but not a 1-local, since $|var(LOC_1) \setminus C_{DET}| = 2$; LOC_2 is a 2-local, but not a 1-local, since $|LOC_2| = 2$; and LOC_3 is a 1-local and also a 2-local.

Basically the only improvement to the algorithm by Gottlob and Leitsch [11] is that the k-locals are the connected components of the literal-graph (defined by Gottlob and Leitsch) after the deterministic subsumption step and not the connected components of the literal-graph of the initial clause.

For example, with the same clause C of the previous example, the connected components would be

$$G_1 = r(X,Y), p(Y,Z), p(Z,T), p(T,Z) \quad (4)$$
$$G_2 = p(U,V), q(V,W) \quad (5)$$
$$G_3 = p(M,N) \quad (6)$$

4.6 GC [28]

In 1996, Scheffer et al. proposed a new method for solving the θ-subsumption problem. Their work is twofold.

First, they borrow the idea of reducing matching candidates from the similar problem of graph isomorphism. They give a new characterisation of the set of clauses that can be tested for subsumption in polynomial time.

Second, they map the subsumption problem to the clique problem (i.e., finding the maximal clique in a graph). they define a highly optimized version of an algorithm for the clique problem, tuning it by knowing in advance which is the size of the clique to be found (which is the size of the first clause of the θ-subsumption problem, as we will see).

4.6.1 Contexts

The context of a literal is described by occurrences of identical variables (or constants) or chains of such occurrences. It is defined for a fixed depth. The intuition is that, for each literal f, we look at the literals that are linked to f, by chains of variables. The idea is that a literal l_1 can only be matched to a literal l_2 if its context is included in the context of l_1.

For example $C = on(X, Y), on(Y, Z)$ cannot subsume $D = on(a, b), on(c, d)$ because the literal $on(X, Y)$ shares the variable Y with the literal $on(Y, Z)$ whereas $on(a, b)$ does not share anything with $on(c, d)$.

We will now define the context of literals formally based on [28].

Definition 4.8 (Occurrence Graph [28]) *The* occurrence graph *of a clause* $C = l_1, \ldots, l_n$ *is the directed edge-labeled graph* (V_C, E_C) *with*

- *vertices* $V_C = \{l_1, \ldots, l_n\}$ *and*
- *labeled edges* E_C *such that* $(l_i, (\pi_i, \pi_j), l_j) \in E_C$ *iff* $l_i\lceil\pi_i\rceil = l_j\lceil\pi_j\rceil$, *with* $l_i \neq l_j$ *or* $\pi_i \neq \pi_j$. *In other words, there is a term t that occurs in literal* l_i *at argument position* π_i *and in literal* l_j *at argument position* π_j. l_i *and* l_j *can refer to the same literal. The term t must occur twice, i.e., if* $i = j$, *we cannot have* $\pi_i = \pi_j$.

Example Let $C = p(X, Y), q(Y, Z), r(Z)$. The occurrence graph of C is represented in Figure 2.

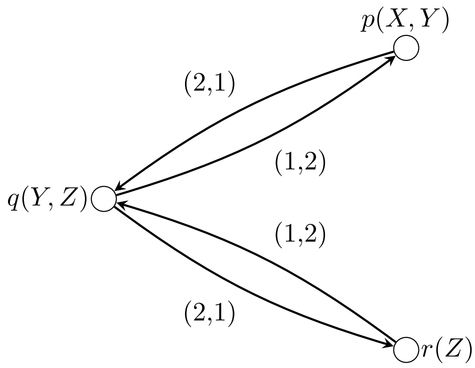

Figure 2: Occurrence graph of the clause C.

Definition 4.9 (Context [28]) *The* context *of depth d of a literal* l_{i_0} *in a clause C, written* $con(l_{i_0}, d, C)$, *is the set of alternating sequenceses of predicate symbols and position pairs such that*

$$\left[p_{i_0}, (\pi_{i_0}, \pi'_{i_1}), p_{i_1}, (\pi_{i_1}, \pi'_{i_2}), p_{i_2}, ..., p_{i_{d-1}}, (\pi_{i_{d-1}}, \pi'_{i_d}), p_{i_d}\right] \in con(l_{i_0}, d, C)$$

iff the following walk

$$l_{i_0}, (\pi_{i_0}, \pi'_{i_1}), l_{i_1}, (\pi_{i_1}, \pi'_{i_2}), ...l_{i_{d-1}}, (\pi_{i_{d-1}}, \pi'_{i_d}), l_{i_d}$$

exists in the occurrence graph of C, and each p_j *is the predicate symbol of literal* l_j. *(Note that* $l_i = l_{i'}$ *is possible too.)*

Example Let $C = p(X,Y), q(Y,Z), r(Z)$. Then the context of depth 2 of literal $p(X,Y)$ is

$$con(p(X,Y), 2, C) = \{[(q,2,1),(r,2,1)], [(q,2,1),(p,1,2)]\}$$

The first sequence means that literal $p(X,Y)$ is linked by a variable at position 2 in $p(X,Y)$ (namely Y) to a variable at position 1 in some literal l_q with predicate symbol q; l_q in turn is linked by a variable at position 2 (namely Z) to a variable at position 1 in some literal with predicate symbol r.

The second sequence is the round trip from $p(X,Y)$ over $q(Y,Z)$ back to $p(X,Y)$.

Proposition 4.10 *Let $l_1 \in C, l'_1 \in D$ be literals, let the depth d be any natural number. Let μ be a matcher of l_1 and l'_1: $l_1\mu = l_2$. If $con(l_1,d,C) \not\subseteq con(l'_1,d,D)$, then there is no substitution θ such that $C\mu\theta \subseteq D$.*

In other words, a literal need not be matched against another literal if its context cannot be embedded in the other literal's context. In [28], there is a proof-sketch of Proposition 4.10, but to the best of our knowledge, a complete proof does not exist yet. For completeness, we provide here a proof. This proof is not based on the proof-sketch since our formalism is not exactly the same.

Proof Let C and D be clauses, $l_1 \in C$ and $l'_1 \in D$ be literals such that there is a μ so that $l_1\mu = l'_1$. Assume $con(l_1,d,C) \not\subseteq con(l'_1,d,D)$ and assume (ad absurbum) that there is a substitution θ such that $C\mu\theta \subseteq D$.
Let $\mathcal{P} = [p_1,(\pi_1,\pi'_2),p_2,(\pi_2,\pi'_3),...,p_{d-1},(\pi_{d-1},\pi'_d),p_d] \in con(l_1,d,C)$.
Then, there is a sequence of literals $l_1,...,l_d$ in C such that for all $j \in \{2..d\}$ $pred(l_j) = p_j$ and $l_{j-1}\lceil\pi_{j-1}\rceil = l_j\lceil\pi'_j\rceil$.
Thus, for all $j \in \{2..d\}$, we have $(l_{j-1}\lceil\pi_{j-1}\rceil)\mu\theta = (l_j\lceil\pi'_j\rceil)\mu\theta$.
Thus, $(l_{j-1}\mu\theta)\lceil\pi_{j-1}\rceil = (l_j\mu\theta)\lceil\pi'_j\rceil$ for $j \in \{2..d\}$.
Since $C\mu\theta \subseteq D$, for all $j \in \{1..d\}$: $l_j\mu\theta \in D$ (and $l_1\mu\theta = l'_1$).
The sequence of literals $l_1\mu\theta,...,l_d\mu\theta$ is in D and such that $(l_{j-1}\mu\theta)\lceil\pi_{j-1}\rceil = (l_j\mu\theta)\lceil\pi'_j\rceil$ for $j \in \{2..d\}$ and $l_1\mu\theta = l'_1$,
thus the walk $l_1\mu\theta,(\pi_1,\pi'_2),l_2\mu\theta,(\pi_2,\pi'_3),l_3\mu\theta,...,l_{d-1}\mu\theta,(\pi_{d-1},\pi'_d),l_d\mu\theta$ is in the occurrence graph of D, starting at $l_1\mu\theta = l'_1$.
And thus $[p_1,(\pi_1,\pi'_2),p_2,(\pi_2,\pi'_3),...,p_{d-1},(\pi_{d-1},\pi'_d),p_d] \in con(l_1\mu\theta,d,D) = con(l_2,d,D)$.
Thus, $con(l_1,d,C) \subseteq con(d_2,d,D)$.
Which contradicts with the assumption. ∎

Definition 4.11 (Substitution Graph) *The* substitution graph *of two clauses $C = l_1,l_2,...,l_n$ and $D = l'_1,l'_2,...,l'_m$ is an undirected graph (V,E) such that*

- *The vertices $V = \{(\theta,i) \in \mathcal{T}(\Sigma)^{\Sigma_V} \times \mathbb{N} \mid 1 \le i \le n \wedge l_i \in C \wedge \exists l' \in D \cdot l_i\theta = l'\}$ are all matching substitutions from any term of C to some term in D. The substitutions are augmented with the number of the originating literal in C (called* layer*) because each clique must contain only one matching substitution for each literal of C.*
- *The edges $E = \{((\theta_1,i_1),(\theta_2,i_2)) \in V \times V \mid \theta_1\theta_2 = \theta_2\theta_1\}$ are the compatibility of the substitutions.*

Definition 4.12 (Clique) *A* clique *in an undirected graph G is a set of vertices V such that for every two vertices in V, there exists an edge connecting the two.*

4.6.2 The algorithm

Algorithm 4: ONETHETA

function `OneTheta`(C, D)
input : C, D two clauses
output: **true** if C θSUBS D
false otherwise

1 **begin**
Match as many literals of C deterministically to literals of D;
Substitute C with the substitution found;
if *some literal of* C *does not match any literal of* D **then**
return false
end
end

2 **begin**
Match as many literals of C context based deterministically to literals of D;
Substitute D with the substitution found;
if *some literals of* C *does not match any literal of* D **then**
return false
end
end

3 **begin**
Set up the substitution graph $(V_{C,D}, E_{C,D})$;
Delete all nodes $(\mu, i) \in V_{C,D}$ with $l_i\mu = l'$ $(l_i \in C, l' \in D)$ and $con(l_i, d, C) \nsubseteq con(l', d, D)$;
return `OneClique` $(V_{C,D}, E_{C,D}, |C|)$;
end

The Algorithm 4 combines the benefit of deterministic matching and the clique search. Clauses that can be tested context based deterministically are tested in polynomial time without any search. In other cases, the algorithm matches as many literals deterministically as possible, and sets up the remaining substitution graph. It follows from proposition 4.10 that we can delete nodes that match literals of C to literal of D which do not possess at least the same context.The remaining graph is searched for a clique with a highly optimized algorithm presented in Algorithm 5.

Algorithm 5: ONECLIQUE(V, E, k)

input : (V, E): a graph with vertices of the form (θ, i)
k: the size of the clique to be found
output: **true** if a clique of size k has been found
false otherwise

1 **begin**
2   **if** $k = 0$ **then return true** ;
3   **if** $|\{i \mid (\theta_j, i) \in V\}| < k$ **then return false** ;
4   Let $v \in V$ be any node;
5   **if** ONECLIQUE($V \cap neighbors_E(v), E, k-1$) **then**
6   **return true** ;
7   **else if** ONECLIQUE($V \backslash \{v\}, E, k$) **then**
8   **return true** ;
9   **else**
10   **return false** ;
11   **end**
12 **end**

The algorithm works essentially like this: pick the first node, search its neighbors for a clique including it, and on failure, search for a clique without it.

The main difference to a normal clique search algorithm is line 3. Additional knowledge about the subsumption problem is used to identify a class of regions of the search space that cannot contain solutions. No substitution can contain two matching substitutions that match the same literal in C to different literals in D (each variable X in C has to match one $X\theta$ in D by definition). Thus, no clique can contain two nodes augmented with equal numbers of originating literals. Therefore the algorithm can be stopped if the number of different augmented numbers in the current V is less than the number of nodes that are missing to make up a clique of the desired size.

4.6.3 Bugs of the implementation

Scheffer et al. have kindly provided the implementation of their algorithm, as well as the source code. Unfortunately, they do not maintain the source code anymore, and it was not possible to fix the bugs that the implementation contains by ourselves.

Here are two examples showing the problem, where θSUBS^{GC} denotes the GC implementation of θ-subsumption.

- Not correct: $p(X, X), p(Y, Y)$. NOT $\theta\text{SUBS}^{GC} p(a, a)$. Since the first clause subsumes the other by $\theta = \{X \mapsto a, Y \mapsto a\}$.
- Correct: $p(X, X), p(Y, X)$. $\theta\text{SUBS}^{GC} p(a, a)$. Surprisingly, on this example the implementation works correctly.

Nonetheless, we keep GC in our experimental evaluation. It must be treated specially since it is not complete. None of the example tested showed that it is not sound, so we believe that the implementation is sound but incomplete.

4.7 Django [20]

Django is a system developed by Maloberti et al. It maps the θ-subsumption problem into a binary CSP and solves the CSP using different CS methods.

4.7.1 CSP

Definition 4.13 (Binary CSP) *A binary CSP is a tuple* (χ, ξ)*, where*

- $\chi = \{X_1, ..., X_n\}$ *is a set of variables. Each* X_i *is associated with a domain* $dom(X_i) = \{a_{i_1}, ...a_{i_{n_i}}\}$*, which is the set of all possible values of* X_i
- $\xi = \{C_1, ..., C_N\}$ *is a set of binary constraints. Each constraint is associated with a set* $arg(C_i) = \{X_j, X_k\} \subseteq \chi$*, which are the two variables involve in the constraint* C_i*.*

  Constraints can be seen as a set of possible assignment to the variables. As we deal with binary constraints only (involving only 2 variables), a constraint is a set of pairs: $C_i = \{(a_{1,1}, a_{1,2}), ..., (a_{N,1}, a_{N,2})\}$*. Further, we restrict this set to be finite.*

Example Consider the problem of having 3 variables which can take the values a or b, and the constraint that each variable must have a different value. The binary CSP for that problem is (χ, ξ), where

- $\chi = \{X_1, X_2, X_3\}$, with $dom(X_1) = dom(X_2) = dom(X_3) = \{a, b\}$.
- $\xi = \{C_1, C_2, C_3\}$ with
  - $arg(C_1) = \{X_1, X_2\}$, $C_1 = \{(a, b), (b, a)\}$,
  - $arg(C_2) = \{X_2, X_3\}$, $C_2 = \{(a, b), (b, a)\}$,
  - $arg(C_3) = \{X_3, X_1\}$, $C_3 = \{(a, b), (b, a)\}$,

4.7.2 Transformation of the θ-subsumption-problem into a CS-problem

Let C and D be clauses, with $C = l_1, l_2, ..., l_n$ where $l_i = p_i(a_{i_1}, ..., a_{i_{k_i}})$, a_{i_j} DATALOG-Terms and $D = l'_1, l'_2, ..., l'_m$ where $l'_i = p'_i(a'_{i_1}, ..., a'_{i_{k_i}})$, a'_{i_j} constant DATALOG-Terms.

Now, the following CSP $(\chi, \xi)_{C,D}$ is defined:

- The set of variables: $\chi = \{Y_{l_i}\}_{l_i \in C}$. Each literal l_i occurring in C give raise to a constrained variable Y_{l_i} termed *dual variable*, as opposite to the variables in C referred to as *primal* variables.
- $dom(Y_{l_i}) = \{l'_j \in D \mid \exists\theta \cdot l_i\theta = l'_j\}$. The domain of a dual variable Y_{l_i} is the set of all literals in D that matches the literal l_i.
- The set of constraints:

  $$\xi = \{C_{(Y_{l_i}, Y_{l_j})}\}_{Y_{l_i}, Y_{l_j} \in \chi, \exists \pi_i, \pi_j \cdot l_i \lceil \pi_i \rceil = l_j \lceil \pi_j \rceil \ \wedge \ l_i \lceil \pi_i \rceil \in \Sigma_V}$$

  For each pair (Y_{l_i}, Y_{l_j}) (possibly $i = j$) such that l_i and l_j share at least 1 variable, a constraint $C_{(Y_{l_i}, Y_{l_j})}$ is defined:

  $$C_{(Y_{l_i}, Y_{l_j})} = \{(l'_{k_1}, l'_{k_2}) \in D \times D \mid \exists\theta.(l_i\theta = l'_{k_1} \wedge l_j\theta = l'_{k_2})\}$$

with $arg(C_{Y_{l_i},Y_{l_j}}) = (Y_{l_i}, Y_{l_j})$. The constraints in the dual CSP, termed dual constraints, are constructed in such a way that literals in C must be mapped onto literals in D so that each (primal) variable in C is mapped onto a single constant or variable in D. The constaint enforces that literals in C that share a variable are mapped to literals in D such that the shared variable is mapped to the same term in D.

Example Let

- $C = t(X_0), p(X_0, X_1), q(X_0, X_2, X_3)$ and
- $D = t(a_0), p(a_0, a_1), p(a_1, a_2), q(a_0, a_2, a_3), q(a_0, a_1, a_3)$.

Then, for the subsumption problem instance "$\exists?\theta.C\ \theta$SUBS D", the following CSP is defined:

- Set of variables: $\chi = \{Y_{t(X_0)}, Y_{p(X_0,X_1)}, Y_{q(X_0,X_2,X_3)}\}$
- With domains:
  - $dom(Y_{t\cdot 1}) = \{t(a_0)\}$
  - $dom(Y_{p\cdot 2}) = \{p(a_0, a_1), p(a_1, a_2)\}$
  - $dom(Y_{q\cdot 3}) = \{q(a_0, a_2, a_3), q(a_0, a_1, a_3)\}$
- Three constraints are defined:
  - $C_1 = \{(t(a_0), p(a_0, a_1))\}$, with $arg(C_1) = (Y_{t(X_0)}, Y_{p(X_0,X_1)})$
  - $C_2 = \{(t(a_0), q(a_0, a_2, a_3)), (t(a_0), q(a_0, a_1, a_3))\}$, with $arg(C_2) = (Y_{t(X_0)}, Y_{q(X_0,X_2,X_3)})$
  - $C_3 = \{(p(a_0, a_1), q(a_0, a_2, a_3)), (p(a_0, a_1), q(a_0, a_1, a_3))\}$, with $arg(C_3) = (Y_{p(X_0,X_1)}, Y_{q(X_0,X_2,X_3)})$

Theorem 4.14 *"$\exists?\theta \cdot C\ \theta SUBS\ D$" admits a solution iff $(\chi, \xi)_{C,D}$ is satisfiable.*

Proof ($\Rightarrow$) Let C θSUBS D by the substitution $\theta = \{X_i \mapsto a_i\}$. Then, $\exists(\theta_1, ..., \theta_n) \in \times_{i=1}^{n} match(C, l_i, D)$, $n = |C|$ such that all θ_i are pairwise strongly compatible (Eisinger). Let $\mathcal{S} = (Y_{l_i} = l_i\theta_i)_{i=1...n}$. We will prove that $\mathcal{S}$ is a solution for $(\chi, \xi)_{C,D}$.

- Trivially, $l_i\theta_i \in dom(Y_{l_i})$.
- For each constraint $C_{(Y_{l_i},Y_{l_j})}$ in ξ, we must prove that $(l_i\theta_i, l_j\theta_j) \in C_{(Y_{l_i},Y_{l_j})}$, i.e., we must prove that $\exists\mu \cdot l_i\mu = l_i\theta_i \wedge l_j\mu = l_j\theta_j$. Let's take $\mu = \theta_i\theta_j$.
  We have $l_i = p_i(t_{i,1}, ..., t_{i,n_i})$,
  then $l_i\theta_i = p_i(t_{i,1}\theta_i, ..., t_{i,n_i}\theta_i)$
  and $l_i\theta_i\theta_j = p_i(t_{i,1}\theta_i\theta_j, ..., t_{i,n_i}\theta_i\theta_j)$.
  If $t_{i,k}$ is a constant, $t_{i,k}\theta_i = t_{i,k}\theta_i\theta_j$.
  If $t_{i,k}$ is a variable, then $t_{i,k} \in dom(\theta_i)$.
  Since θ_i and θ_j are strongly compatible,
  we have $\forall X \in dom(\theta_i) \cdot X\theta_i = X\theta_j$,
  thus $\forall X \in dom(\theta_i) \cdot X\theta_i\theta_i = X\theta_j\theta_i$,
  thus $\forall X \in dom(\theta_i) \cdot X\theta_i = X\theta_j\theta_i$ (idempotency)
  thus $\forall X \in dom(\theta_i) \cdot X\theta_i = X\theta_i\theta_j$ (strong compatibility)
  Thus, $l_i\theta_i\theta_j = l_i\theta_i$.
  Analogously, $l_j\theta_i\theta_j = l_j\theta_j$. Thus, $(l_i\theta_i, l_j\theta_j) \in C_{(Y_{l_i},Y_{l_j})}$. Thus, $\mathcal{S}$ satisfies each constraint in ξ.

Thus $\mathcal{S}$ is a solution for $(\chi, \xi)_{C,D}$.

($\Leftarrow$) Let $(\chi, \xi)_{C,D}$ be satisfiable. Then there exists a solution and let $\mathcal{S} = (Y_{l_i} = l_i^*)_{i \in \{1..n\}}$ be a solution for $(\chi, \xi)_{C,D}$. The aim is to construct a substitution μ such that $C\mu \subseteq D$. Let $\mu_1, ..., \mu_n$ be substitutions defined as follows: each μ_i is the most general matcher of l_i and l_i^* (it must exist since $l_i^* \in Dom(Y_{l_i})$): $l_i\mu_i = l_i^*$. Each μ_i is of the form $\mu_i = \{X_{i,k}^\bullet \mapsto a_{i,k}^\bullet\}_{k \in \{1..n_i\}}$. For each two substitutions μ_i and μ_j $(i, j \in \{1..n\})$, two cases arises:

- Case 1: $\forall k \in \{1..n_i\}, k' \in \{1..n_j\} \cdot X_{i,k}^\bullet \neq X_{j,k'}^\bullet$. Then μ_i and μ_j are strongly compatible.
- Case 2: $\exists k \in \{1..n_i\}, k' \in \{1..n_j\} \cdot X_{i,k}^\bullet = X_{j,k'}^\bullet$. Thus, l_i and l_j share (at least) a variable. Thus there is a constraint $C_{(Y_{l_i}, Y_{l_j})} \in \xi$. Since $\mathcal{S}$ is a solution of $(\chi, \xi)_{C,D}$, the constraint is satisfied and $(l_i^*, l_j^*) \in C_{(Y_{l_i}, Y_{l_j})}$, and thus $\exists \theta^* \cdot l_i\theta^* = l_i^* \wedge l_j\theta^* = l_j^*$. θ^* is of the form $\theta^* = \{X_k^*\}_{k \in \{1..n^*\}}$. W.l.o.g. let $X_{i,1}^\bullet = X_{j,1}^\bullet = X_1^*, ..., X_{i,m}^\bullet = X_{j,m}^\bullet = X_m^*$ be the variables shared by l_i and l_j $(m \geq 1, m \leq n_i, m \leq n_j, m \leq n^*)$. Then, for each $k \in \{1..m\}$: $X_k^*\mu_i = X_k^*\theta^*$ since $l_i\mu_i = l_i\theta^*$ and $X_k^*\mu_j = X_k^*\theta^*$ since $l_j\mu_j = l_j\theta^*$. Thus all variables shared by l_i and l_j are mapped to the same term. Thus μ_i and μ_j are strongly compatible: $\mu_i\mu_j = \mu_j\mu_i$ (since μ_i and μ_j are most general matchers and C and D are variable disjoint).

Thus, $C\mu \subseteq D$, with $\mu = \mu_1...\mu_n$ (Eisinger). ∎

4.7.3 Additional feature of Django

Another data structure is defined to restrict the search space. To each literal in C and D is associated a signature (1-SIG and 2-SIG). 1-SIG is the same as graph context with depth 1 (see Section 4.6).

Definition 4.15 (2-SIG) *Let l_1 be a literal of a clause C. The* 2-signature *(2-SIG) associated with l_1 is the set of tuples $(p, (\pi_{1,1}, \pi_{1,2}), (\pi_{2,1}, \pi_{2,2}))$ such that there exist a literal l_2 in C so that $l_1\lceil\pi_{1,1}\rceil = l_2\lceil\pi_{2,1}\rceil$ and $l_1\lceil\pi_{1,2}\rceil = l_2\lceil\pi_{2,2}\rceil$.*

In other words, the first literal shares 2 variables (or constants) with another literal, and the positions of these variables are encoded in the 2-SIG.

Example Let $C = p(X,Y), q(Y,X)$ be a clause. Then the 2-SIG of $p(X,Y)$ is $\{(q, (1,2), (2,1))\}$, meaning that $p(X,Y)$ shares two variables with a literal l_q with predicate symbol q such that the first variable of $p(X,Y)$ is the second of l_q and the second variable of $p(X,Y)$ is the first of l_q.

A necessary condition for a literal l in C to be mapped onto a literal l' in D is that the 2-signature associated with l is included in that of l'.

Django uses standard well known reduction and search procedures to solve the CSP problem obtained from the θ-subsumption problem. We will give a brief overview of that procedure. A detailed description can be found in [33].

4.7.4 Reduction Procedures

Reduction procedures reduce the variable domains, thereby transforming a CSP into an equivalent one of lower complexity.

Definition 4.16 (2-Consistency) *Let X be a constraint variable and let v denote a value in $dom(X)$. The value v is 2-consistent if, for every variable Y such that there exists a constraint C with $X, Y \in arg(C)$, there exists* some *value w in $dom(Y)$ such that $(v, w) \in C$. w is called the* support *of v wrt. C.*

Example Consider the binary CSP (χ, ξ), where

- The set of variables is $\chi = \{X, Y, Z\}$, with $dom(X) = dom(Y) = dom(Z) = \{a, b, c, d, e, f, g\}$.
- The set of constraints is $\xi = \{C_1, C_2\}$ with
  - $arg(C_1) = \{X, Y\}$, $C_1 = \{(a, b), (c, d)\}$,
  - $arg(C_2) = \{X, Z\}$, $C_2 = \{(a, e), (f, g)\}$

Value a is consistent for X and is supported by b for Y and e for Z. Value c is not consistent.

All non 2-consistent values for a variable X can soundly be removed from $dom(X)$.

Definition 4.17 (Consistency of partial assignments) *A partial assignment θ assigning a value to a subset χ_θ of the constraint variables χ, is consistent if it violates no constraint, i.e., it satisfies all constraints defined on (a subset of) χ_θ.*

Definition 4.18 (k-Consistency) *A CSP is k-consistent iff for any consistent assignment θ over $k-1$ variables, for any variables X not in χ_θ, there exists a value a_X such that $\theta' = \theta \cup \{X \mapsto a_X\}$ is consistent.*

If a consistent partial assignment over some $k-1$ variables in a CSP cannot be extended to another variable, then this partial assignment can soundly be removed from the assignment search space. This pruning technique is termed k-consistency.

4.7.5 Search Procedures

CSP algorithms construct an assignment solution $\{X_i/a_i\}$ through a depth first exploration of the assignment space (the substitution tree). The nodes are variables X_i with edges corresponding to the candidate value a_i tentatively assigned to X_i. On each assignment consistency is checked; on failure, another candidate value for the current node is considered; if no other value is available, the search is backtracked.

Approaches to improve the backtracking procedure are *look-back* algorithms like:

- Conflict directed BackJumping

Look-back algorithms try to avoid the repeated exploration of the same substitution subtree on backtracking.

Approaches to improve the choice of the next variable and candidate value to consider are *look-ahead* like:

- Constraint propagation
- Forward checking
- Maintaining Arc Consistency

Look-ahead algorithms try to minimize the number of assignments considered.

Dynamic or static variable ordering can be used to. Dynamic variable ordering is generally based on the first fail principle, favoring the variable with the smallest domain.

4.7.6 The implementation

The authors of Django have implemented their algorithm in C.

4.7.7 Bugs in the implementation

During the experimental evaluation, the implementation shows some strange behaviour on some example. After reducing the examples to a minimal example, and with correspondence with the authors of Django, it shows up, that there are 2 bugs in their implementation.

The first bug deals with 2-Signatures. The following examples show the problem ($\theta\text{SUBS}^{\,django}$ stands for the implementation of *django*):

- Not correct: $p(X,Y), p(X,X)$ NOT $\theta\text{SUBS}^{\,Django} p(a,a)$. But obviously the first clause subsumes the second by $\theta = \{X \mapsto a, Y \mapsto a\}$.
- Correct: $p(X,Y), p(X,Z)\ \theta\text{SUBS}^{\,Django} p(a,a)$.

So, Django with 2-Signatures is not soundly implemented. Thus, we used a version of Django without 2-Signatures.

The second bug has to do with the arity of the predicates. If the arity is 3 or greater, then Django does not work properly, and the soundness is lost. Consider the following examples:

- Not correct: $ac(A0, parked, S4), ac(A1, airborne, S4)\ \ \theta\text{SUBS}^{\,Django}$ $ac(a0, airborne, s1), ac(a1, parked, s4)$. But the first clause does not subsume the second.
- Correct: $ac_{parked}(A0, S4), ac_{airborne}(A1, S4)$ NOT $\theta\text{SUBS}^{\,Django}$ $ac_{airborne}(a0, s1), ac_{parked}(a1, s4)$. On this example, the implementation works correctly.

Especially in the AIRPORT domain, there are predicates with arity greater than 2. So the comparison is not 100% fair, and the greater speed of execution of Django on some examples may partly due to this bug that renders the implementation of Django not sound.

4.8 Fastheta [6], [9]

Ferilli et al. developed an algorithm that searches all solutions to a given θ-subsumption problem. They define a new concept termed *multi-substitution*, to overcome the need of backtracking. Basically, a multi-substitution is a space-efficient way to store multiple substitutions in a single structure.

In essence, their search is close to a breadth-first search, with the difference of better storage of partial solutions.

4.8.1 Definitions

Definition 4.19 (Multibinding, Multi-substitution) *([9])*
A mutlibinding *is denoted by $X \rightarrow T$, where X is a variable and T is a non-empty set of constants. A* multi-substitution *is a non-empty set of multibindings $\Theta = \{X_1 \rightarrow T_1, ..., X_n \rightarrow T_n\}$, where $\forall i \neq j : X_i \neq X_j$.*

Multi-substitutions will be noted Θ, Ξ, or Σ.

Definition 4.20 (Split) *([9])*
Given a multi-substitution $\Theta = \{X_1 \rightarrow T_1, ..., X_n \rightarrow T_n\}$, split($\Theta$) *is the set of all substitutions represented by: split$(\Theta) = \{\{X_1 \rightarrow c_{i_a}, ..., X_n \rightarrow c_{i_n}\} \mid \forall k = 1...n : c_{i_k} \in T_k \wedge i = 1...|T_k|\}$.*

Definition 4.21 (Union of multi-substitutions) *([9])*
The union *of two multi-substitutions $\Theta' = \{\overline{X} \rightarrow T', X_1 \rightarrow T_1, ..., X_n \rightarrow T_n\}$ and $\Theta'' = \{\overline{X} \rightarrow T'', X_1 \rightarrow T_1, ..., X_n \rightarrow T_n\}$ is the multi-substitution defined as*

$$\Theta' \sqcup \Theta'' = \{\overline{X} \rightarrow T' \cup T''\} \cup \{X_i \rightarrow T_i\}_{1 \leq i \leq n}$$

Note that the two input multi-substitutions must be defined on the same set of variables and must differ in at most one multibinding; otherwise $\Theta' \sqcup \Theta''$ is undefined.

4.8.2 Algorithms

The next definitions are based on algorithms that follow.

Definition 4.22 (Merge) *([9])*
Given a set S of multi-substitutions on the same variables, merge(S) *is the set of multi-substitutions obtained according to Algorithm 6.*

Algorithm 6: merge($\mathcal{S}$)

input : $\mathcal{S}$: a set of multi-substitutions
output: ...
begin
  while $\exists \Theta', \Theta'' \in \mathcal{S}$ *such that* $\Theta' \neq \Theta''$ *and* $\Theta' \sqcup \Theta'' = \Xi$ *is defined* **do**
    $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{\Theta', \Theta''\}) \cup \{\Xi\}$;
  end
  return $\mathcal{S}$;
end

Definition 4.23 *([9])*

The intersection *of two multi-substitutions* $\Sigma = \{X_1 \to S_1, ..., X_n \to S_n, Y_1 \to S_{n+1}, ..., Y_m \to S_{n+m}\}$ *and* $\Theta = \{X_1 \to T_1, ..., X_n \to T_n, Z_1 \to Z_{n+1}, ..., Z_l \to T_{n+l}\}$*, where* $n, m, l \geq 0$ *and* $\forall j, k : Y_j \neq Z_k$*, is the multi-substitution defined as:*

$$\Sigma \sqcap \Theta = \{X_i \to S_i \cap T_i\}_{i=1...n} \cup \{Y_j \to S_{n+j}\}_{j=1...m} \cup \{Z_k \to T_{n+k}\}_{k=1...l}$$

iff $\forall i = 1...n : S_i \cap T_i \neq \emptyset$*; otherwise it is undefined.*

The $\sqcap$ operator can be extended to the case of sets of multi-substitutions. Specifically, given two sets of multi-substitutions $\mathcal{S}$ and $\mathcal{T}$, their intersection is defined as the set of multi-substitutions obtained as follows:

$$\mathcal{S} \sqcap \mathcal{T} = \{\Sigma \sqcap \Theta \mid \Sigma \in \mathcal{S}, \Theta \in \mathcal{T}\}$$

Proposition 4.24 *([9])*

Let $C = l_1, ..., l_n$ *and* $\forall i = 1...n : \mathcal{T}_i = merge(match(C, l_i, D))$*; let* $\mathcal{S}_1 = \mathcal{T}_1$ *and* $\forall i = 2...n : \mathcal{S}_i = \mathcal{S}_{i-1} \sqcap \mathcal{T}_i$*.* C *θ-subsumes* D *iff* $\mathcal{S}_n \neq \emptyset$*.*

This leads to the θ-subsumption procedure reported in Algorithm 7.

Algorithm 7: matching(C, D)

Input: $C = l_1, l_2, ..., l_n$;$D : l'_1, l'_2, ..., l'_m$: two clauses
begin
  Let $\mathcal{S}_1 = merge(match(C, l_1, D))$;
  for $i \leftarrow 2$ **to** n **do**
    $\mathcal{S}_i \leftarrow \mathcal{S}_{i-1} \sqcap merge(match(C, l_i, D))$;
  end
  return $(S_n \neq \emptyset)$;
end

4.8.3 Adapting clauses with constants to work with FASϑ

The algorithm developed by Ferilli et al. put some constraints on the input clauses. Firstly, the first clause must only contain variables as arguments of the predicates. Secondly, the second clause must only contain constants as arguments of the predicates.

The second constraint does not restrain the number of clauses that the algorithm can deal with.

The first constraint is more restrictive. But Ferilli et al. have proposed a workaround. If the first clause contains a constant c, one has to replace all the occurrences of c by a fresh variable C, and add a literal $c(C)$ to the first and $c(c)$ ($c/1$ is a new predicate) to the second clause. By doing this, it is guaranteed that the variable C is always bound to the constant c.

For example if the first clause is

$$C = p(X, a), q(b, Y)$$

and the second

$$D = p(a, X), q(b, c)$$

then C becomes

$$C' = p(X, A), q(B, Y), a(A), b(B)$$

and D becomes

$$D' = p(a, x), q(b, c), a(a), b(b)$$

Complexity issues: For each constant in the first clause, one new variable and one new predicate is added to each clause. The new predicate can only be matched to the corresponding predicate in the other clause. Thus, in the best case, already only a logarithmic increase in the time complexity would arise (since searching an element in a list is done in $O(\log n)$).

5 New approaches

5.1 AllTheta [17]

The general algorithm stays the same as in Algorithm 4 (Algorithm of GC) except that the clique search is replaced by the AllClique algorithm.

The search of all the cliques is done by a specialized depth-first-search. Recall Definition 4.11 of a substitution graph: The vertices are all matching substitutions from any term of the first clause C to some term in second clause D. Each vertice is augmented with the number of the originating literal, which is called the layer.

The graph can thus been represented as successive layers of vertices, where we call the top of the graph the first layer, and the bottom the last layer.

The depth-first search works as follows. Starting from layer 1, the graph is traversed from top to bottom by visiting each layer successively. At each node, the path which has been taken from the first layer to that node is memorised. Each node is check for validity if visited. We say that a node is *valid* if following condition is satisfied:

- The node must have at least one outgoing edge going into each layer, except the layer in which it is;

If a node is not valid when it is visited, it is completely removed from the graph as well as all the edges involving that node, and the search continues with the previously visited node.

We will now present the algorithm in a formal way.

Algorithm 8: AllClique(V, E)

input : V, E the vertices and edges of the substitution graph for clauses Z_1 and Z_2
output: The set of all cliques of size $|Z_1|$ in the graph (V, E)

begin
  $paths \leftarrow \emptyset$;
  $currPath \leftarrow \emptyset$;
  foreach $v = (\mu, 1) \in V$ **do**
    $paths \leftarrow$ `findPath(` *V, E, paths, v, currPath, 1* `)`;
  end
  return paths;
end

The algorithm AllCliques initializes the search. It selects the nodes from the first layer and performs the path search from that node through the graph to the last layer.

Algorithm 9: findPath(V, E, $paths$, v, $currPath$, i)

```
input : [inout] (V, E): The graph
        [inout] paths: paths from first to last layer forming a clique
        v: the currently visited node
        currPath: the current path from first layer to current layer
        i: current layer
output: Paths from first to last layer forming a clique
begin
    if valid(v) then
        currPath ← currPath ∪ {v};
        if i = |Z_1| then
            paths ← paths ∪ {currPath};
        else
            foreach u = (μ', i + 1) ∈ V with (u, v) ∈ E do
                if clique(u,currPath) then
                    findPath(V,E,paths,u,currPath,i+1);
                end
            end
        end
    else
        V ← V \ {v};
    end
    return paths;
end
```

Experimental evaluations have shown that the construction of the substitution graph is costly. To overcome this problem, the graph can be constructed progressively layer by layer. At each step in this construction process, the nodes are checked for validity, where the validity for a node is slightly changed:

- A valid node must have at least one outgoing edge going into each *previously constructed* layer, except the layer in which it is;

5.2 Object-contexts

In the following section, we denote as objects the variables and constants of clauses.

This approach is a similar approach as graph-context. The difference is that the vertives of the created occurrence graph are the *objects* and not the *literals*.

Definition 5.1 (Object Occurrence Graph) *The* object occurrence graph *for a clause C is a labelled directed graph $GC = (V, E, \ell)$, where*

- *the vertices V are objects of C, denoted as $Obj(C)$;*
- *the labeled edges E are such that $(o_1, o_2, (\pi_1, \pi_2, f)) \in E$ iff $(o_1, o_2) \in V \times V$ and there is a literal l in C based on function symbol f such that $l\lceil\pi_1\rceil = l\lceil\pi_2\rceil$;*

- *the labeling function* $\ell : V \to 2^{\Sigma_F}$ *such that* $\ell(o) = \{f/1 \in \Sigma_F \mid f/1(o) \in C\}$*, i.e.,* ℓ *associates each object* o *with the set of unary function symbols* $f/1$ *this object belongs to.*

Definition 5.2 (Object Context) *Let* C *be a clause,* $o \in Obj(C)$ *and* $d \in \mathbb{N}, d > 0$*. The* object context *of depth* d *of an object* o *in* C*, written* $objcon(o, C, d)$*, is a set of alternatinsequenceses of labels and tuples of the form* (π, f, π') *such that*

$$\left[l(o_{i_1}), (\pi_{i_1}, f_{i_1}, \pi'_{i_1}), l(o_{i_2}), (\pi_{i_2}, f_{i_2}, \pi'_{i_2}), ..., l(o_{i_{d-1}}), (\pi_{i_d}, f_{i_d}, \pi'_{i_d}), l(o_{i_d})\right] \in objcon(o, C, d)$$

iff the following walk exists in GC

$$o, (\pi_{i_1}, f_{i_1}, \pi'_{i_1}), o_{i_1}, (\pi_{i_2}, f_{i_2}, \pi'_{i_2}), ..., o_{i_{d-1}}, (\pi_{i_d}, f_{i_d}, \pi'_{i_d}), o_{i_d}$$

Example Let $C = p(X, Y), p(Y, t), r(X), s(X), q(Y), u(Y)$. Then the object context of variable X of depth 1 is

$$objcon(X, 1, C) = \{[\{r, s\}, (1, p, 2), \{q, u\}]\}$$

and the object context of depth 2 of X is

$$objcon(X, 2, C) = \{[\{r, s\}, (1, p, 2), \{q, u\}, (1, p, 2), \{\}], [\{r, s\}, (1, p, 2), \{q, u\}, (2, p, 1), \{r, s\}]\}$$

Proposition 5.3 *Let* C *and* D *be clauses,* $X \in var(C)$*,* $o \in Obj(D)$*, and* $d \in N, d > 0$*. Let there be a matching substitution* μ *such that* $X\mu = o$*. If* $objcon(X, C, d) \nsubseteq objcon(o, D, d)$ *then there exists no substitution* θ *such that* $C\mu\theta \subseteq D$*.*

In other words, a variable X in C need not be matched against an object o in D if the variable's context cannot be embedded in the object's context.

The algorithm for finding the substitutions is based on the idea of [28], and is presented in Algorithm 10.

Algorithm 10: OBJCON-ALLTHETA

function `OneTheta`(C, D)
input : C, D two clauses
output: **true** if C θSUBS D
false otherwise
begin
  Match as many literals of C deterministically to literals of D;
  Substitute C with the substitution found;
  if *some literal of* C *does not match any literal of* D **then**
    return false
  end
end
begin
  Match as many literals of C object context based deterministically to literals of D;
  Substitute D with the substitution found;
  if *some literals of* C *does not match any literal of* D **then**
    return false
  end
end
begin
  Set up the substitution graph $(V_{C,D}, E_{C,D})$;
  Delete all nodes $(\mu, i) \in V_{C,D}$ with $X\mu = o$ (for some $X \in var(C), o \in Obj(D))$ such that $objcon(X, d, C) \nsubseteq objcon(o, d, D)$;
  return `OneClique` $(V_{C,D}, E_{C,D}, |C|)$;
end

5.2.1 Bugs in the implementation

The implementation has kindly be given to us by Eldar Karabaev. The experimental evaluation shows that on some rare examples, the implementation is not sound, meaning that the implementation detects that a clause θ-subsumes another while they do not.

The following examples show the problem (θSUBS ^{objcon} stands for the implementation of *objcon*):

- Not correct: $p(X, X)$ θSUBS $^{objcon} p(a, b), p(b, a)$. Obviously, the first clause does not subsume the second.
- Correct: $p(X, X)$ NOT θSUBS $^{objcon} p(c, b), p(b, a)$.

This shows that the implementation is not sound. The experimental evaluation did not show any case in which *objcon* was not complete, so we believe that it is complete.

6 Application domains

6.1 Planning

We adopt here a formalism near to the one presented in [15] which is based on the conjunctive fluent calculus. A more general, and widely used representation are

Markov Decision Processes[2]. We adopted the simpler representation here for the sake of understandability. It would be of little use to overlaod this section with a full description of MDPs, dealing with probabilities and complex algorithms for finding optimal policies[3]. For the experimental evaluationn it is no difference, since at the stage were θ-subsumption comes into play, either representation deals with literals or equivalent entities (e.g. *fluents* as in [14]). The aim is here to give a general understanding of planning, and how θ-subsumption is used in that process.

Planning problem. A planning problem is a tuple $(\mathcal{Z}, Z_{\mathcal{I}}, Z_{\mathcal{G}}, \mathcal{A})$, where

- $\mathcal{Z}$ is a set of *(abstract) states* (described below);
- $Z_{\mathcal{I}}$ is the *initial state*;
- $Z_{\mathcal{G}}$ is the *goal state*;
- $\mathcal{A}$ is a finite set of actions of the form $A : \text{Pre} \Rightarrow \text{Eff}$, where $\text{Pre} = \{c_1, ..., c_l\}$ and $\text{Eff} = \{e_1, ..., e_k\}$ are sets of atoms and are called *preconditions* and *effects* respectively.

An action $A : \text{Pre} \Rightarrow \text{Eff}$ is *applicable* in a state Z iff Pre θSUBS Z, i.e., there is a substitution θ such that $\text{Pre}\theta \subseteq Z$.

The *application of an action* in a state Z gives rise to a new state Z' defined as follows: $Z' = (Z \setminus \text{Pre}\theta) \cup \text{Eff}\theta$. A sequence of actions $[A_1, ..., A_n]$ is called a *plan*. A plan is a *solution* of a planning problem iff the successive application of the actions in the plan transforms the initial state $Z_{\mathcal{I}}$ to the goal state $Z_{\mathcal{G}}$.

In the following, we adapt the representation from [14], where states were represented using the Probabilistic Fluent Calculus, to our formalism of clauses.

Abstract states are represented by clauses containing only positive literals. The difference to the Fluent Calculus representation is that literals cannot occur multiple times in clauses (since clauses are sets of literals). This can by bypassed by the following transformation

Transforming clauses with multiple occurrences of literals to clauses without multiple occurrences but preserving θ-subsumption: Let $\underline{C}$ be a clause were literals can occur multiple times.

For each literal $p(a_1, \ldots, a_n)$ occurring multiple times (say k times) in $\underline{C}$, replace all

$$p(a_1, \ldots, a_n)$$

by all the elements of

$$\{p(a_1, \ldots, a_n, Z_i) | i \in \{1, \cdots, k\}\}$$

[2]Markov Decision Processes (MDPs) have become the representational and computational standard for planning problems and more generally for Decision Theoretic Planning (DTP) [1]. DTP is an extension of the classical AI planning paradigm. It allows to model actions with uncertain effects, as well as incomplete knowledge about the world. Furthermore, resource consumption can be represented, i.e., an action consumes a certain amount of resources. In addition, the goal may be given by a goal specification, i.e., the goal is not fully described, only the relevant data is given.

[3]A policy is a function from the set of states to the set of action. So a policy gives for each state an action that can be applied

and add the literals of the following set

$$\{diff(Z_i, Z_j) | i, j \in \{1, \cdots, k\}, i \neq j\}$$

For example, the clause with multiple occurrences

$$\underline{C} = p(X), p(X), q(X)$$

would become

$$C = p(X, Z_1), p(X, Z_2), q(X), diff(Z_1, Z_2), diff(Z_2, Z_1)$$

Complexity issues: Let n be the size of the clause $\underline{C}$, and let k be the maximum of the number of multiple occurrences of the literals. Then, the size of the new clause C is smaller than or equal to n' and

$$n' = n + \underbrace{(k-1)}_{p(a_1, \cdots, a_n, Z_i)} + \underbrace{(k-1)^2}_{diff(Z_i, Z_j)}$$

which is bound by $O(n + k^2)$.

Search for a plan In the following, we recast the algorithm described in [14] to the search of a plan within our planning framework.

A forward search strategy is applied. The initial state is the root of the search tree. A node is then chosen from the tree's fringe, i.e., the set of all leaf nodes, and all applicable actions are applied. Each action application extends the plan by one step and generates a new state. The search ends when a state is subsumed by the goal state. A solution plan can then be extracted from the search tree. Forward search aims at finding a solution from the beginning to the end by adding actions to the end of the current sequence od actions. Forward search only considers states that can be reached from the initial state $Z_{\mathcal{I}}$.

Normalization The new set of states can then be pruned be removing redundant state, this phase is called normalization ([14], [2]). As in [14], we deal with abstract states. In essence, the normalization can be seen as the exhaustive application of the following rule:

$$\frac{Z_1 \quad Z_2}{Z_1} \; Z_1 \; \theta\text{SUBS} \; Z_2$$

Example Let $Z_1 = on(X_2, a)$ and $Z_2 = on(X_1, a), on(a, table)$, then we have Z_1 θSUBS Z_2, so that the state Z_1 can be pruned.

In [14], it is shown that the normalization drastically shrinks the computational effort during the iterations of the forward search. For example, on a simple planning problem, they demonstrate that after the seventh iteration, the search space after the normalization is 11.6 times smaller than before. That means that over ninety percent of the initial search space is redundant. They show also that the computation time is orders of magnitude faster with the normalization.

6.2 ILP

Inductive logic programming (ILP) is a subfield of machine learning which uses logic programming as a uniform representation for examples, background knowledge and hypotheses. Given an encoding of the known background knowledge and a set of examples represented as a logical database of facts, an ILP system will derive an hypothesised logic program which entails all the positive and none of the negative examples.

ILP systems commonly use θ-subsumption as generality relation. The generality relation is used as the covering test to test whether a hypothesis covers a training example.

6.3 Theorem proving (*prover9*)

Automated theorem proving, currently the most well-developed subfield of automated reasoning, is the proving of mathematical theorems by a computer program. First-order theorem proving is one of the most mature subfields of automated theorem proving. The logic is expressive enough to allow the specification of arbitrary problems, often in a reasonably natural and intuitive way. On the other hand, it is still semi-decidable, and a number of sound and complete calculi have been developed, enabling fully automated systems.

prover9 is one of these automated theorem provers for first-order and equational logic developed by William McCune, it is the successor of the system OTTER.

prover9 is applicable to statements in classical first-order logic with equality. It accepts as input either clauses or quantified formulas. Quantified formulas are transformed to clauses by usual normal form conversion and skolemisation.

prover9's main inference rules are based on resolution or paramodulation.

The procedure of *prover9* for processing a newly inferred clause *new_cl* follows; steps marked with $*$ are optional. The details of the algorithm are beyond the scope of this thesis. It is shown here to have an idea of where θ-subsumption is used in a theorem-proving engine. The interested reader is referred to [21] for a comprehensive description.

	1.	Renumber variables.	
*	2.	Output *new_cl*.	
	3.	Demodulate *new_cl*.	
*	4.	Orient equalities.	
*	5.	Apply unit deletion.	
	6.	Merge identical literals (leftmost copy is kept).	
*	7.	Apply factor-simplification.	
*	8.	Discard *new_cl* and exit if too many literals or variables.	
	9.	Discard *new_cl* and exit if *new_cl* is a tautology.	
*	10.	Discard *new_cl* and exit if *new_cl* is too ?heavy?.	
*	11.	Sort literals.	
*	12.	Discard *new_cl* and exit if *new_cl* is subsumed by any clause in usable, sos, or passive (forward subsumption).	Steps
	13.	Integrate *new_cl* and append it to sos.	
*	14.	Output kept clause.	
	15.	If *new_cl* has 0 literals, a refutation has been found.	
	16.	If *new_cl* has 1 literal, then search usable, sos, and passive for unit conflict (refutation) with *new_cl*.	
*	17.	Print the proof if a refutation has been found.	
*	18.	Try to make *new_cl* into a demodulator.	
*	19.	Back demodulate if Step 18 made *new_cl* into a demodulator.	
*	20.	Discard each clause in usable or sos that is subsumed by *new_cl* (back subsumption).	
*	21.	Factor *new_cl* and process factors.	

19-21 are delayed until steps 1-18 have been applied to all clauses inferred from the active given clause.

7 Experimental evaluation

7.1 Experimental settings

All experiments are done on a 1,4GHz Pentium M running under Linux Debian ("Etch"). The results are presented in the form depicted in Table 2. The parameters will depend on the problem instance at hand. Not all subsumers are applicable to all problems. The timing results μ and σ are the average time needed for one subsumption attempt and the standard deviation, expressed in seconds.

Parameters				Subsumer				
n	v	c	....	ST	DC	GC	Dj	...
...	...	...	...	$\mu_{ST}(\sigma_{ST})$	$\mu_{DC}(\sigma_{DC})$	$\mu_{GC}(\sigma_{GC})$	$\mu_{Dj}(\sigma_{Dj})$	...

Table 2: Format of the experimental results.

7.2 Datasets

7.2.1 Random

The first dataset referred to as RANDOM is generated with the help of the random generator.

For the experimental evaluation, we generate 100 clauses (which will be enough for 10000 subsumption attempts) with following variable parameters

- n: Number of literals in each clause
- a, b: Minimal and maximal arity for predicates
- v: Number of different variables
- c: Number of different constants
- p: Number of different predicates

In this part, we analyse the influence of the parameters described above, without any other particular structure. Only random literals are generated and put together to obtain a clause of the desired size.

In the following we will shortly describe the combinations that have been tested.

Varying size of the clauses The first series is obtained by varying the size of the clauses, all other parameters staying the same. We fixed the number of predicates to 5, the arity is set to two, no constants are allowed and the number of variables is set to 5. We set the number of predicates and variables to a small number for having many possible matches for each literal.

The size of the clauses varies from 5 to 2000 with increasing steps.

Tables 3 and 4 show the results.

Parameters			*Django*	*GC*	Allθ	*DC*	*ST*	Res	Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	κ
5	5	5	$0,003$	$0,006$	$0,008$	$0,081$	$0,043$	38	1
10	5	5	$0,005$	$0,018$	$0,012$	$0,492$	$0,235$	0	0,57
15	5	5	$0,008$	$0,041$	$0,016$	$0,858$	$1,286$	4	0,32
20	5	5	$0,010$	$0,090$	$0,019$	$2,947$	$1,155$	2	0,14
30	5	5	$0,018$	$1,052$	$0,053$	$5,255$	$2,408$	0	-0,11
50	5	5	$0,074$	$3,567$	$0,284$	$34,754$	$10,280$	100	-0,43
70	5	5	$0,356$		$11,659$		$42,144$	500	-0,64
100	5	5	$2,779$					2400	-0,86
150	5	5	$22,966$					5500	-1,11
200	5	5							-1,29
500	5	5							-1,86
700	5	5							-2,07
1000	5	5							-2,29
2000	5	5							-2,72

Table 3: Dataset: RANDOM, Problem: YesNo, Variing: size of clauses *size*. 10000 subsumption attempts.

The configuration of the clause renders a small probability of successful subsumption for small clauses (about 0.1%). Django gives the best results with a gain factor ranging from 2 to 32 to the second best. The second best algorithm is ALLTHETA.

The variation of the size only does change the behaviour of the algorithms as the clauses do not have a special structure due to the random generation.

Parameters			FASϑ	*ObjCon*	AllTheta	Results		Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	pos	nbSubst	κ
5	5	5	0,036	0,307	0,020	38	45	1
10	5	5	0,031	0,080	0,017	0		0,57
15	5	5	0,043	0,091	0,024	4	5	0,32
20	5	5	0,054	0,103	0,029	2	2	0,14
30	5	5	0,092	0,129	0,090	0		-0,11
50	5	5	0,234	0,293	0,519	100	100	-0,43
70	5	5	0,438	2,125	16,729	500	514	-0,64
100	5	5	0,933			2400	2804	-0,86
150	5	5	2,455			5500	10876	-1,11
200	5	5	6,452			8600	46276	-1,29
500	5	5				0		-1,86
700	5	5				0		-2,07
1000	5	5				0		-2,29
2000	5	5				0		-2,72

Table 4: Dataset: Random, Problem: ALL, Variing: size of clauses *size*. 10000 subsumption attempts.

GC has to be treated specially as it is not complete and overlooks some of the positive subsumption tests. This can clearly be seen here, as it gives the wrong number of positive results most of the time. In the following we will not consider GC in the comparison, since it would not be a fair comparison.

Varying number of predicate In this series, the influence of the number of predicates is investigated. The number of predicates varies from 2 to 100 with increasing steps. Two different settings for the other parameters are taken: the first with clauses of size 100, and the second with clauses of size 500. The number of constants and variables is fixed to zero and five respectively. The arity of the predicate is set to two.

Tables 5 and 6 show the results

Parameters			*Django*	*GC*	Allθ	*DC*	*ST*	Res	Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	κ
50	2	5	7,045				46,267	9300	-0,17
50	5	5	0,073	3,565	0,286	34,774	10,299	100	-0,43
50	7	5	0,038	1,274	0,061	14,907	4,857	0	-0,6
50	10	5	0,037	0,259	0,032	7,663	2,214	0	-0,86
50	15	5	0,037	0,131	0,028	4,354	0,992	0	-1,29
50	20	5	0,042	0,104	0,028	3,090	0,716	0	-1,72
50	30	5	0,045	0,059	0,027	1,811	0,387	0	-2,58
50	50	5	0,044	0,033	0,027	1,121	0,244	0	-4,31
50	100	5	0,044	0,021	0,023	0,693	0,166	0	-8,61
500	2	5							-0,74
500	5	5							-1,86
500	7	5							-2,61
500	10	5							-3,72
500	15	5	22,538					600	-5,58
500	20	5	5,364					0	-7,45
500	30	5	3,676		2,016			0	-11,17
500	50	5	3,324	23,231	0,273			0	-18,61
500	70	5	3,218	7,701	0,279		43,759	0	-26,06
500	100	5	3,696	4,815	0,697		26,143	0	-37,23
500	150	5	3,635	2,132	0,314		11,829	0	-55,84
500	200	5	3,838	1,406	0,346		8,353	0	-74,45
500	300	5	4,179	0,912	0,433		5,026	0	-111,68
500	500	5	4,415	0,514	0,312	40,202	3,469	0	-186,14
500	1000	5	5,628	0,476	0,399	26,318	2,821	0	-372,27

Table 5: Dataset: Random, Problem: YesNo, Variing: number of predicates *pred*. 10000 subsumption attempts.

Parameters			FASϑ	*ObjCon*	AllTheta	Results		Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	pos	nbSubst	κ
50	2	5	1,978			9300	80839	-0,17
50	5	5	0,243	0,337	0,573	100	100	-0,43
50	7	5	0,158	0,169	0,103	0		-0,6
50	10	5	0,135	0,150	0,046	0		-0,86
50	15	5	0,146	0,143	0,038	0		-1,29
50	20	5	0,150	0,146	0,036	0		-1,72
50	30	5	0,158	0,138	0,033	0		-2,58
50	50	5	0,170	0,163	0,034	0		-4,31
50	100	5	0,218	0,156	0,028	0		-8,61
500	2	5				0		-0,74
500	5	5				0		-1,86
500	7	5	48,923			10000	976100	-2,61
500	10	5	14,547			7600	21889	-3,72
500	15	5	4,235			600	600	-5,58
500	20	5	3,429			0		-7,45
500	30	5	2,242		1,819	0		-11,17
500	50	5	1,831		0,318	0		-18,61
500	70	5	2,094		0,325	0		-26,06
500	100	5	2,488		0,423	0		-37,23
500	150	5	3,199		0,333	0		-55,84
500	200	5	3,854		0,331	0		-74,45
500	300	5	4,884		0,345	0		-111,68
500	500	5	6,338		0,343	0		-186,14
500	1000	5	9,018		0,395	0		-372,27

Table 6: Dataset: Random, Problem: ALL, Variing: number of predicates *pred*. 10000 subsumption attempts.

For few different predicates, Django outperforms the others. With increasing number of different predicates, AllTheta becomes the best algorithm, Django still been the second most of the time.

Surprisingly, the execution speed of ST faster and faster for large clauses with many different predicates. This could be explained by the fact that the probability that a literal in the first clause matches a literal in the second one goes near to zero, so that the order of the literals, on which ST heavily depends on, matters less and less.

The low probability of matching seem to be a key factor for the superior efficiency of the graph-contexts in AllTheta over arc consistency in Django. Even though theoretically both are equivalent, graph-contexts are faster when it comes to detect that a clause does not subsume another.

Varying ratio size of clauses and number of predicates In this series, the ratio $\frac{size}{nbPrediate}$ varies. A ratio greater than one means that the size is greater than the number of predicates. The probability of successful subsumption will increase, as there are more possible matches for each literal. The results show that the factor must be 10.0 for having an effect. On the other hand, a ratio less than one means more different predicates, reducing the successful subsumption probability.

The other parameters are set as follows: arity to 2, number of constants to zero, number of variables to 5.

Table 7 and 8 present the obtained results.

The two best competitors are AllTheta and Django. The observation from the previous paragraph seem to be valid here too. Django outperforms AllTheta when there are more positive subsumption results, and vice versa.

Parameters			*Django*	*GC*	Allθ	*DC*	*ST*	Res	Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	κ
5	5	5	0,003	0,006	0,008	0,081	0,042	38	1
7	7	5	0,004	0,008	0,008	0,129	0,059	2	1,11
10	10	5	0,005	0,009	0,011	0,227	0,089	0	1,14
15	15	5	0,007	0,012	0,011	1,030	0,244	0	0,95
20	20	5	0,009	0,016	0,014	0,543	0,146	0	0,55
30	30	5	0,017	0,021	0,018	0,553	0,156	0	-0,68
50	50	5	0,044	0,033	0,027	1,100	0,242	0	-4,31
70	70	5	0,074	0,049	0,056	2,040	0,365	0	-8,96
100	100	5	0,159	0,066	0,050	2,580	0,581	0	-17,23
10	5	5	0,005	0,018	0,012	0,490	0,233	0	0,57
14	7	5	0,007	0,023	0,013	0,908	0,685	0	0,5
20	10	5	0,009	0,030	0,014	1,192	0,327	0	0,28
30	15	5	0,016	0,045	0,020	1,182	0,317	0	-0,34
40	20	5	0,025	0,055	0,024	1,610	0,423	0	-1,17
60	30	5	0,051	0,078	0,034	3,014	0,562	0	-3,26
100	50	5	0,135	0,137	0,049	4,885	1,174	0	-8,61
140	70	5	0,283	0,266	0,069	8,641	1,339	0	-14,99
200	100	5	0,547	0,364	0,103	14,144	2,044	0	-25,84
25	5	5	0,014	0,182	0,032	3,343	1,485	1	0
35	7	5	0,022	0,182	0,029	4,542	1,597	0	-0,29
50	10	5	0,037	0,258	0,031	7,667	2,219	0	-0,86
75	15	5	0,074	0,379	0,051	14,514	3,191	0	-2,05
100	20	5	0,127	0,515	0,051	16,135	4,914	0	-3,45
150	30	5	0,277	1,041	0,081	32,351	5,653	0	-6,68
250	50	5	0,807	1,682	0,122		10,808	0	-14,31
350	70	5	1,588	2,340	0,194		13,766	0	-22,96
500	100	5	3,277	3,903	0,378		23,331	0	-37,23
50	5	5	0,073	3,575	0,283	34,815	10,301	100	-0,43
70	7	5	0,101	2,111	0,267	38,874	14,699	100	-0,9
100	10	5	0,174	3,374	0,197		25,975	100	-1,72
150	15	5	0,285	5,928	0,125		29,513	0	-3,34
200	20	5	0,481	7,846	0,125		40,277	0	-5,17
300	30	5	1,110	10,454	0,159		53,467	0	-9,26
500	50	5	3,162	18,979	0,267			0	-18,61
700	70	5	6,380	25,958	0,430			0	-28,99
1000	100	5		32,969	0,740			0	-45,84
5	10	5	0,008	0,005	0,007	0,058	0,027	0	2
7	14	5	0,004	0,005	0,008	0,080	0,034	0	2,21
10	20	5	0,005	0,006	0,011	0,139	0,052	0	2,28
15	30	5	0,007	0,008	0,011	0,516	0,115	0	1,9
20	40	5	0,010	0,010	0,013	0,334	0,086	0	1,11
30	60	5	0,018	0,014	0,019	0,355	0,104	0	-1,36
50	100	5	0,046	0,021	0,023	0,693	0,166	0	-8,61
70	140	5	0,092	0,030	0,082	1,177	0,240	0	-17,91
100	200	5	0,186	0,043	0,051	2,130	0,375	0	-34,45
5	25	5	0,003	0,004	0,007	0,042	0,019	0	5
7	35	5	0,004	0,004	0,007	0,057	0,024	0	5,54
10	50	5	0,005	0,005	0,011	0,092	0,033	0	5,69
15	75	5	0,008	0,006	0,010	0,219	0,060	0	4,76
20	100	5	0,011	0,007	0,012	0,195	0,059	0	2,77
30	150	5	0,020	0,010	0,018	0,237	0,078	0	-3,4
50	250	5	0,049	0,016	0,027	0,455	0,127	0	-21,53
70	350	5	0,091	0,024	0,041	0,757	0,184	0	-44,78
100	500	5	0,183	0,040	0,057	1,213	0,291	0	-86,14

Table 7: Dataset: Random, Problem: YesNo, Variing: ratio *size*/*pred*. 10000 subsumption attempts.

Parameters			FASϑ	*ObjCon*	AllTheta	Results		Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	pos	nbSubst	κ
5	5	5	$0,030$	$0,311$	$0,023$	38	45	1
7	7	5	$0,023$	$0,075$	$0,011$	2	2	1,11
10	10	5	$0,032$	$0,079$	$0,014$	0		1,14
15	15	5	$0,047$	$0,087$	$0,014$	0		0,95
20	20	5	$0,062$	$0,098$	$0,017$	0		0,55
30	30	5	$0,095$	$0,116$	$0,021$	0		-0,68
50	50	5	$0,172$	$0,150$	$0,034$	0		-4,31
70	70	5	$0,249$	$0,182$	$0,061$	0		-8,96
100	100	5	$0,390$	$0,221$	$0,058$	0		-17,23
10	5	5	$0,033$	$0,085$	$0,019$	0		0,57
14	7	5	$0,041$	$0,090$	$0,017$	0		0,5
20	10	5	$0,058$	$0,096$	$0,019$	0		0,28
30	15	5	$0,088$	$0,117$	$0,025$	0		-0,34
40	20	5	$0,119$	$0,126$	$0,029$	0		-1,17
60	30	5	$0,186$	$0,159$	$0,041$	0		-3,26
100	50	5	$0,337$	$0,217$	$0,058$	0		-8,61
140	70	5	$0,513$	$0,281$	$0,082$	0		-14,99
200	100	5	$0,846$	$0,365$	$0,119$	0		-25,84
25	5	5	$0,072$	$0,114$	$0,052$	1	1	0
35	7	5	$0,090$	$0,132$	$0,041$	0		-0,29
50	10	5	$0,137$	$0,148$	$0,047$	0		-0,86
75	15	5	$0,209$	$0,182$	$0,066$	0		-2,05
100	20	5	$0,289$	$0,227$	$0,067$	0		-3,45
150	30	5	$0,464$	$0,295$	$0,100$	0		-6,68
250	50	5	$0,881$	$0,467$	$0,149$	0		-14,31
350	70	5	$1,418$	$0,574$	$0,228$	0		-22,96
500	100	5	$2,503$		$0,440$	0		-37,23
50	5	5	$0,237$	$0,292$	$0,528$	100	100	-0,43
70	7	5	$0,277$	$0,308$	$0,488$	100	100	-0,9
100	10	5	$0,318$	$0,333$	$0,337$	100	100	-1,72
150	15	5	$0,438$	$0,380$	$0,190$	0		-3,34
200	20	5	$0,579$	$0,427$	$0,172$	0		-5,17
300	30	5	$0,940$	$0,586$	$0,206$	0		-9,26
500	50	5	$1,836$		$0,337$	0		-18,61
700	70	5	$3,036$		$0,490$	0		-28,99
1000	100	5	$5,414$		$0,823$	0		-45,84
5	10	5	$0,019$	$0,070$	$0,008$	0		2
7	14	5	$0,025$	$0,076$	$0,009$	0		2,21
10	20	5	$0,034$	$0,085$	$0,012$	0		2,28
15	30	5	$0,050$	$0,087$	$0,012$	0		1,9
20	40	5	$0,067$	$0,104$	$0,015$	0		1,11
30	60	5	$0,104$	$0,116$	$0,021$	0		-1,36
50	100	5	$0,193$	$0,154$	$0,027$	0		-8,61
70	140	5	$0,309$	$0,176$	$0,088$	0		-17,91
100	200	5	$0,527$	$0,225$	$0,057$	0		-34,45
5	25	5	$0,020$	$0,073$	$0,007$	0		5
7	35	5	$0,027$	$0,075$	$0,008$	0		5,54
10	50	5	$0,037$	$0,085$	$0,012$	0		5,69
15	75	5	$0,055$	$0,090$	$0,011$	0		4,76
20	100	5	$0,076$	$0,104$	$0,014$	0		2,77
30	150	5	$0,132$	$0,120$	$0,020$	0		-3,4
50	250	5	$0,273$	$0,155$	$0,030$	0		-21,53
70	350	5	$0,458$	$0,198$	$0,045$	0		-44,78
100	500	5	$0,849$	$0,239$	$0,063$	0		-86,14

Table 8: Dataset: Random, Problem: ALL, Variing: ratio *size*/*pred*. 10000 subsumption attempts.

Varying number of variables This time, the influence of the number of different variables is analysed. The number of variables varies from 5 to 100. The arity is set to 2. The number of constants is zero. The tests are done for various combinations for the size of the clauses and the number of predicates.

Table 9 shows the results.

Parameters			*Django*	*GC*	Allθ	*DC*	*ST*	Res	Metric
size	*pred*	*vars*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	κ
5	5	2	0,004	0,007	0,008	0,085	0,036	203	-0,8
5	5	5	0,003	0,006	0,008	0,081	0,043	38	1
5	5	7	0,003	0,007	0,009	0,080	0,055	22	0,84
5	5	10	0,003	0,008	0,010	0,079	0,057	33	0,65
5	5	15	0,003	0,009	0,012	0,075	0,071	108	0,47
5	5	20	0,003	0,010	0,017	0,087	0,083	285	0,37
5	5	30	0,003	0,011	0,019	0,089	0,087	345	0,25
5	5	50	0,004	0,012	0,028	0,092	0,089	689	0,16
5	5	70	0,004	0,013	0,050	0,106	0,094	1316	0,12
5	5	100	0,004	0,014	0,060	0,113	0,099	1338	0,08
50	5	2	3,927		1,101	3,234	0,823	8909	-9,11
50	5	5	0,074	3,569	0,284	34,830	10,315	100	-0,43
50	5	7	0,039	2,345	0,145	34,291	39,472	0	-0,01
50	5	10	0,039	4,420	0,101	38,042		0	0,15
50	5	15	0,037	26,489	0,056	29,948		0	0,19
50	5	20	0,037		0,055	18,849		0	0,17
50	5	30	0,038		0,100	11,555		0	0,14
50	5	50	0,038		0,072	5,144		0	0,1
50	5	70	0,035		0,077			0	0,08
50	5	100	0,033		0,100			0	0,06
500	5	2			2,604	2,881	0,827	10000	-17,41
500	5	5							-1,86
500	5	7							-0,85
500	5	10							-0,35
500	5	15	48,514					1100	-0,1
500	5	20	27,237					300	-0,02
500	5	30	13,240					0	0,03
500	5	50	5,433					0	0,04
500	5	70	4,162					0	0,04
500	5	100	3,629					0	0,03
50	50	2	0,060	0,030	0,023	0,861	0,185	0	-91,1
50	50	5	0,046	0,034	0,027	1,101	0,242	0	-4,31
50	50	7	0,036	0,037	0,028	1,094	0,274	0	-0,07
50	50	10	0,034	0,046	0,030	1,146	0,345	0	1,51
50	50	15	0,032	0,055	0,029	1,163	0,397	0	1,85
50	50	20	0,032	0,063	0,033	1,142	0,492	0	1,74
50	50	30	0,032	0,075	0,037	0,954	0,602	0	1,42
50	50	50	0,032	0,116	0,041	0,603	0,877	0	1
50	50	70	0,032	0,125	0,042	0,681	0,869	0	0,77
50	50	100	0,032	0,138	0,045	1,252	0,992	0	0,58
500	50	2			5,421		17,455	200	-174,14
500	50	5	3,402	20,048	0,270			0	-18,61
500	50	7	2,862	27,355	0,317			0	-8,53
500	50	10	2,501		0,325			0	-3,49
500	50	15	2,351		0,289			0	-0,98
500	50	20	2,418		0,287			0	-0,19
500	50	30	2,486		0,347			0	0,29

Table 9: Dataset: RANDOM, Problem: YesNo, Variing: number of variables *vars*. 10000 subsumption attempts.

With small clauses, having few or many variables, Django is the fastest. With increasing size of the clauses, the gain of Django becomes less important. This is again due to the fact that fewer subsumption tests succeeds. ALLTHETA detects earlier that a clause does not subsume another and is thus faster in that case.

We can observe some interesting cases in which ST outperforms all other systems. This time it is not due to the negative case, but on the contrary to an extremely high probability of success for the subsumption test. This suggests that in those cases, no matter which literal is selected first and matched against another, there is no need to backtrack since the match would result in a solution with high probability.

Due to bugs and limitations of the implementations, we did not test out all possibilities. In particular, we set the arity of the predicates to two in all dataset generated randomly since Django do not work correctly if the arity is 3 or greater.

Due to the limitation of FASϑ in respect to constants in the first clauses, we did not introduce constants in the first clause and handled only the case where variables occur.

The experimentation on randomly generated clauses shows that when the clauses are generated randomly according to a uniform distribution, the simple look at the basic parameters (number of variables, number of predicate, size of the clause) can give us a clue on which algorithm is the best one.

In their work on Django, Maloberti and Sebag [20], have ported the κ parameter from CSP to θ-subsumption:

$$\kappa = \frac{m \cdot (2 \log_2 L - \log_2 N)}{n \cdot \log_2 L}$$

where n denotes the number of variables in C, L is the number of constants and variables in D, and m is the number of different predicate symbols.

In the θ-subsumption problems from the Random domain, the κ parameter can easily be calculated. We can make the following observation in 99% of the cases:

- In case $\kappa > 2$, Django is best
- In case $\kappa < -2$, AllTheta is best (nearly: in some special cases discussed above, ST may outperform AllTheta)
- In case $-2 \leq \kappa \leq 2$, it is not known *a priori* which algorithm is better suited.

The the cases in which all solution have to be found, the best competitors were FASϑ and AllTheta. One could observe that AllTheta outperform the others in most of the cases. This is due to the fact that randomly generated clauses ofter do not subsume each other. As stated earlier, AllTheta detects very soon, that one clause cannot subsume another due to the graph-context build for the clauses.

The κ-parameter allows us to state the following in 99% of the cases:

- If $\kappa > 2$ or $\kappa < 0$, then AllTheta is the best in the random case.
- if $0 \leq \kappa \leq 2$, then we cannot tell in advance which is better on the presented experimental results.

7.2.2 Blocksworld

The blocksworld domain is the typical toy example when dealing with planning problems. It consists of a table on which various cubic blocks are placed. A block can be on top of another block. For moving the block, there is a robotic arm that can grasp one block at a time, move it around, and place it on the table or on top another block. Given an initial situation (an initial state), the aim is to move the blocks around to get to a given final situation (a final state).

A state is represented by a clause. There are several predicates:

- *on*/2: Given two blocks b_1 and b_2, the meaning of $on(b_1, b_2)$ is that block b_1 is situated on block b_2.
- *ontable*/1: Given a block b, the meaning of $ontable(b)$ is that block b is on the table.
- *clear*/1: Given a block b, $clear(b)$ indicates that there is no block ontop of b.
- *holding*/1: Given a block b, $holding(b)$ means that the robotic arm is holding the block b.
- *empty*/0: Means that the robotic arm is not holding anything.

Four actions can be performed on the states, defined as follows

- Action $pickup : clear(V), ontable(V), empty \Rightarrow holding(V)$
- Action $unstack : clear(V), on(V, W), empty \Rightarrow holding(V), clear(W)$
- Action $putdown : holding(V) \Rightarrow clear(V), ontable(V), empty$
- Action $stack : holding(V), clear(W) \Rightarrow on(V, W), clear(V), empty$

Example The state defined by

$$ontable(a), on(b, a), clear(b), empty$$

would be transformed into the state

$$ontable(a), holding(b), clear(a)$$

by the application of the action *unstack*.

7.2.3 Pipesworld

The Pipesworld domain models the flow of oil-derivative liquids through *pipeline segments* connecting *areas*, and is derived by applications in the oil industry [22].

Batches of a certain size models the liquids. A segment must allways be full, i.e., they contain allways the same number of batches. Batches can be pushed into pipelines from either side, which has as effect that the batch at the other side of the segment will be pushed out of the segment and fall into the incident area. Batches have associated *product types*. Certain type of liquid must never be adjacent in a pipeline. Areas may have constraints on how many batches of a certain product type they can hold.

Formally, the Pipesworld is defined as follows.

Definition 7.1 (Pipesworld [13]) *Let $P = \{lco, gasoline, rata, oca1, oc1b\}$ be a set of* products. *Two products $p, p' \in P$ are called* compatible *unless $p = rata$ and $p' \in \{oca1, oc1b\}$ or vice versa.*

A PIPESWORLD *task is given by:*

- *finite sets of* areas *A and* pipeline segments *S,*
- *a finite set ob* batches *B, each with a* product type *$b^P \in P$,*
- *for each pipeline segment $s \in S$, a* start area *$s^- \in A$ and an* end area *$s^+ \in A$ and a* segment length *$|s| \in \mathbb{N}_1$,*
- *an* area capacity function *$c : A \times P \to \mathbb{N}_0$,*
- *a* goal contents function *$C_G : A \to 2^B$ such that for each batch $b \in B$, we have $b \in C_G(a)$ for at most one area $a \in A$, and*
- *an initial state: a state is defined by an* area contents *function $C_A : A \to 2^B$ and a* pipeline segment contents *function $C_S : S \to B^*$ such that*
  - *for each batch $b \in B$ either $b \in C_A(a)$ for exactly one area $a \in A$, or $b \in C_S(s)$ for exactly one segment $s \in S$ (meaning that a batch exists exactly once, and must be localized either in an area or in a segment),*
  - *for all areas $a \in A$ and products $p \in P$, $C_A(a)$ contains at most $c(a, p)$ batches of product type p, and*
  - *for all pipeline segments $s \in S$, $|C_S(s)| = |s|$ (the segments are allways completely full with some products) and any two adjacent batches in $C_S(s)$ can interface, i.e., have compatible product types.*

A state is a goal state iff $C_G(a) \subseteq C_A(a)$ for all $a \in A$.

The only action in the task is the push *action. If $s \in S$ is a pipeline segment with contents $b_1 ... b_{|s|}$ and $b \in C_A(s^-)$ is a batch that can interface with b_1, then b can be pushed into s. This results in a state where the new contents of segment s are $bb_1 ... b_{|s|-1}$, b is no longer in $C_A(s^-)$, and $b_{|s|}$ is in $C_A(s^+)$. Similarly, $b \in C_A(s^+)$ can be pushed into s if it can interface with $b_{|s|}$, leading to a state where the contents of s are $b_2 ... b_{|s|}b$, b is no longer in $C_A(s^+)$, and b_1 is in $C_A(s^-)$. Pushing a batch into a pipeline segment is only allowed if the state obtained after the action application would not violate the area capacity constaints.*

Formally, the PIPESWORLD can be represented based on [22] in clausal form as follows.

- The positioning of batches is done with 5 predicates:
  - $first/2$, $last/2$, $follow/2$: these predicates are used to define the content of the pipelines. $first(batch, pipe)$ means that *batch* is situated at the first position is the pipeline *pipe*. Similarly, $last(batch, pipe)$ indicates the last *batch* in the pipeline. $follow(batch_1, batch_2)$ is used to construct a list of batches that are located one after the other in the pipes.

  - $on/2$: $on(batch, tank)$ indicates that the batch *batch* is on the tank *tank*
  - $isProduct/2$: $isProduct(batch, product)$ indicates the product of a batch.

- Areas and tanks are defined through the predicates $tankInArea/2$ and $tankProduct/2$: $tankInArea(tank, area)$ means that *tank* is located in the area *area*, and $tankProduct(tank, product)$ means that *tank* can hold only products of type *product*.
- Segments are defined through the predicate *connect*: $connect(area_1, area_2, pipe)$ means that the area $area_1$ is connected to area $area_2$ by a pipeline segment *pipe*.
- Finally, the compatibility of products is done with the predicate $mayInterface/2$: $mayInterface(product_1, product_2)$ mean that $product_1$ and $product_2$ are compatible and can thus interface.
- In our representation, the capacity constraints are not modeled.

Example Consider the following Pipesworld problem

- There are two areas a_1 and a_2,
- There is one segment s of length 3 connecting a_1 and a_2,
- There are two tank in each area for product type *lco* and *rata*
- The batches are called $b_1, b_2, ...$ and are of type *lco*

A state of this problem could be:

$$\begin{array}{l}
connect(a_1, a_2, s),\\
first(b_1, s), follow(b_1, b_2), follow(b_2, b_3), last(b_3, s),\\
tankInArea(t_{11}, a_1), tankProduct(t_{11}, lco),\\
tankInArea(t_{12}, a_1), tankProduct(t_{12}, rata),\\
tankInArea(t_{21}, a_2), tankProduct(t_{21}, lco),\\
tankInArea(t_{22}, a_2), tankProduct(t_{22}, rata),\\
isProduct(b_1, lco), isProduct(b_2, lco), isProduct(b_3, lco).
\end{array}$$

Planning in the Pipesworld domain is NP-hard [13].

7.2.4 Airport

The Airportdomain is another real world planning problem.

Planning in the Airport domain is PSPACE-equivalent [13].

The results of the experimental evaluation in the Airport domain is reported in Tables 11 and 12 for the problem of finding all solutions for a given subsumption problem. Tables 13 and 14 reports the result for the problem of finding out whether the subsumption test holds or not.

Parameters			fastheta		objcon	alltheta		Results		
a	p	n	μ (ms)	σ (ms)	μ (ms)	μ (ms)	σ (ms)	pos	total	subst
2	2	10	0,418	0,650	4,000	0,614	1,077	3	10	10
3	2	10	0,979	0,733	10,000	40,879	40,528	8	10	56
4	2	10	0,783	0,340	6,000	74,547	115,101	6	10	44
5	2	10	1,223	0,772	14,000	459,250	536,665	7	10	46
6	2	10	1,496	0,615	26,000	1045,079	910,516	10	10	86
7	2	10	5,246	2,952	57,000	9868,951	6765,917	10	10	197
8	2	10	6,674	5,811	8388,000	14992,975	16477,516	10	10	202
9	2	10	68,030	85,750	52075,000			10	10	361
10	2	10	68,013	78,929	10087,000			10	10	311
11	2	10	116,261	181,514				10	10	394
12	2	10	480,642	481,925				10	10	529
13	2	10	4438,062	7488,458				10	10	707
2	3	10	0,469	0,611	5,000	0,407	0,441	2	10	6
3	3	10	0,767	0,381	5,000	20,233	20,883	6	10	39
4	3	10	0,846	0,185	6,000	122,624	92,700	7	10	51
5	3	10	1,752	0,697	18,000	1647,827	1095,073	9	10	139
6	3	10	3,162	1,241	109,000	10419,058	8222,374	10	10	183
7	3	10	4,584	1,907	90,000	24929,049	15535,528	10	10	319
8	3	10	21,757	12,632	168,000			10	10	446
9	3	10	29,734	24,524	231,000			10	10	485
10	3	10	121,037	179,257	47240,000			10	10	546
11	3	10	191,012	202,041				10	10	610
12	3	10	1020,407	918,470				10	10	1008
13	3	10	2651,414	2244,462				10	10	1181
2	4	10	0,531	0,121	4,000	5,273	8,323	4	10	30
3	4	10	0,823	0,484	8,000	60,393	104,345	4	10	29
4	4	10	0,979	0,197	9,000	405,741	425,469	8	10	71
5	4	10	1,572	0,635	9,000	2135,726	2241,159	8	10	117
6	4	10	3,738	1,907	140,000	24688,936	14969,441	10	10	271
7	4	10	7,336	5,449	152,000	57867,035	24896,689	10	10	297
8	4	10	26,311	27,850	402,000			10	10	455
9	4	10	130,094	101,810	16065,000			10	10	575
10	4	10	269,114	221,963	830,000			10	10	876
11	4	10	1552,104	2113,759				10	10	965
12	4	10	1843,696	3563,665	2758,000			10	10	1116
13	4	10	3865,117	3168,594				10	10	1375
2	5	10	0,345	0,284	5,000	1,036	1,812	2	10	8
3	5	10	0,858	0,275	6,000	150,781	127,769	5	10	104
4	5	10	1,281	0,469	7,000	1039,189	934,671	8	10	143
5	5	10	1,844	0,506	12,000	6638,105	4877,151	9	10	155
6	5	10	4,319	3,188	55,000	28419,917	21922,619	10	10	370
7	5	10	9,263	9,005	174,000			10	10	485
8	5	10	26,818	22,216	564,000			10	10	539
9	5	10	43,623	48,804	10659,000			10	10	518
10	5	10	291,444	298,725	4472,000			10	10	907
11	5	10	934,835	1302,319	5371,000			10	10	1145
12	5	10	3284,922	8180,932				10	10	959

Table 10: Timing results for the PIPESWORLD domain. Actions-States configuration. The parameters are: a for the number of areas, p for the number of products, n for the number of clauses

Parameters					FASϑ	$ObjCon$	AllTheta	Results		
ac	s	t	p	b	μ (ms)	μ (ms)	μ (ms)	pos	total	nbSubst
2	2	1	0	1	0,015	4,833	0,021	19	60	21
2	2	1	1	0	0,015	5,111	0,017	13	45	15
2	2	1	1	1	0,014	3,500	0,017	23	80	25
2	3	1	0	0	0,015	4,667	0,018	19	60	21
2	3	1	1	1	0,014	4,833	0,019	18	60	20
2	3	2	0	0	0,015	4,667	0,017	19	60	21
2	3	2	0	1	0,015	4,667	0,017	19	60	21
2	3	2	0	2	0,013	2,833	0,019	34	120	42
2	3	2	1	0	0,015	5,000	0,019	19	60	21
2	3	2	1	1	0,012	2,833	0,019	30	120	37
2	3	2	1	2	0,015	3,238	0,058	34	105	39
2	3	2	2	0	0,015	6,444	0,018	13	45	15
2	3	2	2	1	0,016	6,444	0,037	13	45	15
2	3	2	2	2	0,013	3,500	0,037	29	100	32
2	4	1	1	1	0,002	11,500	0,015	0	20	0
2	4	2	2	0	0,011	6,000	0,016	13	65	15
2	4	2	2	1	0,015	3,030	0,019	53	165	60
2	4	2	2	2	0,513	7,556	0,018	13	45	15
2	4	3	0	0	0,014	3,889	0,040	26	90	33
2	4	3	0	1	0,013	3,167	0,019	34	120	42
2	4	3	0	2	0,015	2,917	0,019	38	120	46
2	4	3	0	3	0,015	6,000	0,018	13	45	15
2	4	3	1	0	0,030	4,083	0,020	39	120	45
2	4	3	1	1	0,015	4,667	0,036	19	60	21
2	4	3	1	2	0,013	5,000	0,020	15	60	20
2	4	3	2	0	0,001	20,000	0,015	0	10	0
2	4	3	2	1	0,015	4,833	0,017	19	60	21
2	4	3	2	2	0,719	4,000	0,020	38	120	46
2	4	3	2	3	0,021	3,000	0,020	61	220	69
2	4	3	3	0	0,014	3,125	0,025	50	160	56
2	4	3	3	1	0,014	3,778	0,048	26	90	30
2	4	3	3	2	0,015	7,333	0,020	13	45	15
2	4	3	3	3	0,015	3,333	0,019	40	120	45
2	5	1	1	0	0,016	5,111	0,017	13	45	15
2	5	2	0	0	0,015	4,667	0,017	19	60	21
2	5	2	1	0	0,015	4,833	0,019	19	60	21
2	5	2	2	0	0,015	5,778	0,019	13	45	15
2	5	2	2	2	0,011	3,400	0,015	24	100	27
2	5	3	0	0	0,015	6,444	0,018	13	45	15
2	5	3	0	1	0,015	6,444	0,018	13	45	15
2	5	3	0	2	0,016	4,000	0,018	30	85	33
2	5	3	0	3	0,015	5,778	0,018	13	45	15
2	5	3	1	2	0,015	6,444	0,020	13	45	15
2	5	3	1	3	0,010	4,462	0,022	12	65	12
2	5	3	2	0	0,006	6,571	0,015	2	35	2
2	5	3	2	3	0,015	5,800	0,019	15	50	18
2	5	3	3	1	0,015	3,083	0,022	36	120	42
2	5	3	3	3	0,013	3,077	0,021	33	130	42
2	5	4	0	0	0,008	5,600	0,014	8	50	8
2	5	4	0	1	0,015	6,667	0,019	13	45	15
2	5	4	1	0	0,011	7,667	0,018	6	30	6
2	5	4	1	1	0,015	4,833	0,024	19	60	21
2	5	4	1	3	0,017	4,667	0,021	18	60	21
2	5	4	1	4	0,013	3,778	0,020	23	90	30
2	5	4	2	0	0,012	3,778	0,019	23	90	30
2	5	4	2	1	0,015	1,875	0,022	78	240	94
2	5	4	2	3	0,002	23,000	0,015	0	10	0
2	5	4	2	4	0,016	1,855	0,022	94	275	119
2	5	4	3	0	0,015	2,963	0,021	41	135	51
2	5	4	3	1	0,008	2,500	0,016	26	160	28
2	5	4	3	3	0,015	2,222	0,020	57	180	67
2	5	4	3	4	0,015	6,222	0,018	13	45	15
2	5	4	4	0	0,013	1,962	0,019	70	260	87
2	5	4	4	1	0,015	5,231	0,019	21	65	24
2	5	4	4	3	0,012	2,318	0,017	58	220	73
2	5	4	4	4	0,013	2,250	0,018	59	200	67
3	3	1	0	0	0,018	1,889	0,025	69	180	87
3	3	1	1	0	0,007	6,000	0,023	4	50	4
3	3	1	1	1	0,002	11,500	0,021	0	20	0
3	3	2	0	0	0,017	1,667	0,029	84	240	116

Table 11: Dataset: Airport(1), Configuration: Precondition/States.

Parameters					FASϑ	*ObjCon*	AllTheta	Results		
ac	s	t	p	b	μ (ms)	μ (ms)	μ (ms)	pos	total	nbSubst
3	3	2	0	1	0,025	2,222	0,028	57	180	84
3	3	2	1	0	0,011	2,909	0,023	22	110	26
3	3	2	1	1	0,016	1,667	0,029	84	240	116
3	3	2	2	0	0,019	1,387	0,042	148	375	180
3	4	1	0	1	0,001	7,000	0,021	0	40	0
3	4	2	0	0	0,030	2,370	0,026	47	135	63
3	4	2	0	1	0,019	2,222	0,027	69	180	87
3	4	2	0	2			0,042	174	420	219
3	4	2	1	0	0,018	1,324	0,027	129	340	165
3	4	2	1	1	0,001	11,500	0,020	0	20	0
3	4	2	1	2	0,017	1,789	0,037	103	285	140
3	4	2	2	0	0,016	4,250	0,049	26	80	30
3	4	2	2	1	0,002	13,000	0,021	0	20	0
3	4	2	2	2	0,017	1,204	0,026	176	465	211
3	4	3	0	0	0,019	1,778	0,028	88	225	111
3	4	3	0	1	0,018	2,429	0,029	56	140	65
3	4	3	0	2	0,019	1,911	0,028	88	225	111
3	4	3	0	3	0,026	1,875	0,027	97	240	120
3	4	3	1	0	0,015	2,424	0,069	52	165	63
3	4	3	1	1	0,001	23,000	0,021	0	10	0
3	4	3	1	2	0,017	1,789	0,030	102	285	127
3	4	3	2	0	0,023	1,358	0,049	159	405	213
3	4	3	2	1	0,017	2,581	0,026	56	155	62
3	4	3	2	2	0,020	1,425	0,030	168	400	219
3	4	3	3	0	0,019	1,700	0,036	126	300	156
3	4	3	3	1	0,023	0,685	0,033	532	1065	678
3	4	3	3	2	0,020	0,793	0,033	367	870	465
3	5	1	1	0	0,001	5,750	0,021	0	40	0
3	5	1	1	1	0,001	11,500	0,021	0	20	0
3	5	2	0	1	0,002	11,500	0,020	0	20	0
3	5	2	0	2	0,013	2,581	0,026	40	155	42
3	5	2	1	0	0,018	1,342	0,037	154	380	216
3	5	2	1	1	0,019	1,778	0,028	88	225	111
3	5	2	1	2	0,002	11,500	0,021	0	20	0
3	5	2	2	0	0,020	0,795	0,029	326	780	449
3	5	2	2	1	0,037	1,729	0,025	94	295	110
3	5	2	2	2	0,009	1,818	0,020	48	220	48
3	5	3	0	0	0,012	2,051	0,023	48	195	50
3	5	3	0	2	0,014	1,667	0,024	73	270	87
3	5	3	1	0	0,014	4,000	0,025	24	85	28
3	5	3	1	1	0,025	1,545	0,041	141	330	180
3	5	3	1	2	0,009	4,250	0,023	16	80	16
3	5	3	1	3	0,019	1,700	0,029	124	300	153
3	5	3	2	0	0,012	3,579	0,023	22	95	24
3	5	3	2	2	0,021	1,109	0,031	231	505	294
3	5	3	2	3	0,018	1,292	0,031	185	480	246
3	5	3	3	0	0,020	1,821	0,028	124	280	153
3	5	3	3	1	0,019	0,482	0,047	669	1700	913
3	5	3	3	2	0,015	2,250	0,024	58	200	62
3	5	3	3	3	0,008	1,729	0,027	44	295	44
3	5	4	0	0	0,022	1,226	0,032	186	465	246
3	5	4	0	1	0,016	1,731	0,036	83	260	112
3	5	4	0	3	0,021	0,838	0,052	295	800	424
3	5	4	0	4	0,011	1,683	0,033	74	315	78
3	5	4	1	0	0,017	1,594	0,029	112	320	153
3	5	4	1	1	0,019	1,120	0,029	198	500	258
3	5	4	1	3	0,024	0,954	0,033	294	650	381
3	5	4	1	4	0,021	1,583	0,032	154	360	198
3	5	4	2	1	0,019	1,700	0,028	120	300	148
3	5	4	2	3	0,019	1,889	0,030	99	270	132
3	5	4	2	4	0,020	1,556	0,030	148	360	189
3	5	4	3	0	0,012	2,045	0,022	52	220	54
3	5	4	3	1	0,019	2,222	0,029	66	180	87
3	5	4	3	3	0,026	0,771	0,033	408	1025	547
3	5	4	3	4	0,019	1,048	0,033	236	620	322
3	5	4	4	0	0,025	0,492	0,038	963	1850	1286
3	5	4	4	1	0,018	2,963	0,027	47	135	63
3	5	4	4	3	0,019	1,825	0,043	102	285	137

Table 12: Dataset: Airport(2), Configuration: Precondition/States.

Parameters					*Django*	*GC*	AllTheta	*DC*	*ST*	Results	
ac	*s*	*t*	*p*	*b*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	total
2	2	1	1	1	0,005	0,007	0,035	0,097	0,039	9	81
2	3	1	1	0	0,003	0,005	0,011	0,094	0,029	12	144
2	3	2	0	0	0,007	0,006	0,019	0,054	0,057	15	81
2	3	2	0	1	0,005	0,006	0,015	0,103	0,037	32	225
2	3	2	0	2	0,004	0,006	0,026	0,143	0,052	12	144
2	3	2	1	2	0,006	0,011	0,014	0,110	0,061	9	81
2	3	2	2	0	0,007	0,007	0,019	0,184	0,043	52	400
2	3	2	2	1	0,005	0,007	0,018	0,152	0,051	12	144
2	3	2	2	2	0,023	0,027	0,038	0,334	0,123	6	16
2	4	1	1	0	0,003	0,005	0,019	0,050	0,033	33	441
2	4	2	0	1	0,007	0,007	0,020	0,119	0,037	39	225
2	4	2	1	1	0,006	0,007	0,016	0,079	0,041	21	144
2	4	3	0	0	0,005	0,012	0,029	0,090	0,071	15	81
2	4	3	0	1	0,004	0,012	0,017	0,127	0,090	18	324
2	4	3	0	3	0,006	0,007	0,020	0,176	0,065	35	729
2	4	3	1	0	0,003	0,008	0,016	0,070	0,052	84	1296
2	4	3	1	1	0,007	0,012	0,018	0,118	0,089	11	81
2	4	3	1	3	0,009	0,012	0,032	0,309	0,067	16	144
2	4	3	2	0	0,033	0,042	0,024	0,371	0,161	22	100
2	4	3	3	0	0,008	0,015	0,018	0,322	0,083	9	81
2	4	3	3	1	0,004	0,010	0,019	0,121	0,063	31	961
2	4	3	3	2	0,020	0,027	0,061	0,722	0,179	5	25
2	4	3	3	3	0,009	0,017	0,018	0,278	0,092	18	196
2	5	2	2	0	0,010	0,020	0,022	0,156	0,116	13	49
2	5	2	2	1	0,005	0,008	0,015	0,162	0,056	32	784
2	5	2	2	2	0,017	0,024	0,037	0,450	0,117	10	36
2	5	3	0	0	0,003	0,006	0,028	0,051	0,033	21	144
2	5	3	0	1	0,005	0,009	0,037	0,083	0,050	21	144
2	5	3	0	2	0,005	0,009	0,021	0,105	0,056	54	576
2	5	3	1	0	0,007	0,020	0,027	0,093	0,048	21	144
2	5	3	1	3	0,017	0,056	0,069	0,604	0,299	30	144
2	5	3	2	0	0,013	0,012	0,152	0,132	0,139	6	16
2	5	3	2	3	0,007	0,020	0,015	0,202	0,120	12	144
2	5	3	3	0	0,006	0,012	0,031	0,319	0,085	12	144
2	5	3	3	1	0,009	0,015	0,020	0,308	0,078	9	81
2	5	3	3	2	0,007	0,019	0,029	0,414	0,112	12	144
2	5	3	3	3	0,003	0,010	0,013	0,184	0,093	69	4761
2	5	4	0	3	0,005	0,019	0,016	0,141	0,137	12	144
2	5	4	0	4	0,009	0,025	0,034	0,376	0,147	36	484
2	5	4	1	0	0,005	0,010	0,021	0,104	0,057	14	144
2	5	4	1	1	0,009	0,028	0,019	0,300	0,105	17	144
2	5	4	2	0	0,004	0,011	0,017	0,164	0,090	54	1296
2	5	4	2	1	0,005	0,026	0,020	0,186	0,177	37	729
2	5	4	2	3	0,007	0,019	0,030	0,432	0,136	142	3600
2	5	4	2	4	0,006	0,013	0,016	0,718	0,151	85	2500
2	5	4	3	0	0,006	0,021	0,059	0,156	0,270	9	81
2	5	4	3	1	0,009	0,008	0,025	0,231	0,135	21	225
2	5	4	4	0	0,031	0,023	0,084	0,375	0,107	47	225
2	5	4	4	1	0,005	0,022	0,023	0,568	0,140	36	1296
2	5	4	4	3	0,006	0,026	0,015	0,373	0,172	60	2116
2	5	4	4	4	0,009	0,031	0,037	0,491	0,199	33	676
3	3	1	0	0	0,013	0,017	0,066	0,100	0,076	11	16
3	3	1	0	1	0,004	0,011	0,028	0,071	0,072	81	1296
3	3	1	1	0	0,003	0,010	0,017	0,079	0,075	69	2304
3	3	1	1	1	0,003	0,013	0,016	0,088	0,090	36	1296
3	3	2	0	0	0,003	0,009	0,023	0,091	0,071	141	3249
3	3	2	0	1	0,007	0,013	0,042	0,113	0,080	302	2304
3	3	2	0	2	0,018	0,045	0,067	0,524	0,214	16	49
3	3	2	1	0	0,006	0,016	0,021	0,116	0,101	109	1296
3	3	2	1	1	0,004	0,018	0,017	0,274	0,114	33	676
3	3	2	1	2	0,004	0,014	0,022	0,267	0,103	234	5184
3	3	2	2	0	0,004	0,014	0,024	0,293	0,117	175	8281
3	3	2	2	1	0,006	0,021	0,027	0,351	0,125	93	1296
3	3	2	2	2	0,003	0,015	0,017	0,266	0,118	138	6561

Table 13: Dataset: Airport(1), Configuration: States/States.

Parameters					*Django*	*GC*	AllTheta	*DC*	*ST*	Results	
ac	s	t	p	b	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	total
3	4	1	0	0	0,004	0,013	0,039	0,073	0,076	129	729
3	4	1	0	1	0,003	0,009	0,019	0,072	0,068	111	2304
3	4	1	1	1	0,004	0,016	0,021	0,086	0,105	57	729
3	4	2	0	0	0,008	0,014	0,045	0,108	0,067	139	529
3	4	2	0	2	0,003	0,018	0,018	0,095	0,116	36	1296
3	4	2	1	0	0,003	0,014	0,020	0,223	0,101	173	3969
3	4	2	1	2	0,005	0,030	0,022	0,263	0,186	35	400
3	4	2	2	0	0,003	0,013	0,022	0,118	0,109	327	10000
3	4	2	2	1	0,006	0,015	0,030	0,190	0,127	99	1296
3	4	2	2	2	0,004	0,015	0,020	0,219	0,107	74	3600
3	4	3	0	0	0,007	0,016	0,036	0,257	0,109	471	3600
3	4	3	0	1	0,004	0,019	0,023	0,297	0,131	129	4356
3	4	3	0	2	0,004	0,022	0,018	0,225	0,137	51	1296
3	4	3	0	3	0,004	0,022	0,029	0,190	0,204	132	3969
3	4	3	1	1	0,010	0,018	0,040	0,162	0,093	452	3600
3	4	3	1	2	0,005	0,016	0,019	0,203	0,126	159	3600
3	4	3	1	3	0,003	0,019	0,018	0,434	0,221	115	10000
3	4	3	2	0	0,005	0,025	0,022	0,429	0,172	95	2304
3	4	3	2	1	0,018	0,061	0,050	0,639	0,320	31	225
3	4	3	2	2	0,005	0,017	0,018	0,337	0,143	67	2500
3	4	3	2	3	0,004	0,019	0,021	0,466	0,229	63	2025
3	4	3	3	0	0,011	0,018	0,044	0,183	0,177	195	2304
3	4	3	3	1	0,004	0,014	0,020	0,278	0,223	156	5776
3	4	3	3	2	0,003	0,020	0,018	0,844	0,295	116	10000
3	4	3	3	3	0,004	0,023	0,015	0,186	0,159	48	2304
3	5	1	0	0	0,005	0,017	0,045	0,084	0,112	26	64
3	5	2	0	2	0,003	0,023	0,015	0,107	0,155	27	729
3	5	2	1	0	0,003	0,015	0,026	0,085	0,084	141	1296
3	5	2	1	1	0,003	0,020	0,027	0,121	0,126	57	729
3	5	2	1	2	0,005	0,032	0,022	0,186	0,214	30	400
3	5	2	2	0	0,003	0,014	0,020	0,097	0,104	144	5184
3	5	2	2	1	0,006	0,034	0,029	0,285	0,209	19	256
3	5	2	2	2	0,004	0,017	0,027	0,273	0,105	43	1156
3	5	3	0	0	0,020	0,027	0,052	0,193	0,180	22	144
3	5	3	0	1	0,005	0,016	0,027	0,316	0,111	197	3969
3	5	3	0	2	0,014	0,060	0,081	0,223	0,378	42	144
3	5	3	0	3	0,003	0,026	0,019	0,285	0,253	190	10000
3	5	3	1	0	0,023	0,174	0,093	0,245	0,406	6	16
3	5	3	1	1	0,004	0,031	0,032	0,235	0,204	70	1296
3	5	3	1	2	0,013	0,044	0,039	0,348	0,244	47	484
3	5	3	1	3	0,030	0,099	0,091	1,004	0,514	16	49
3	5	3	2	0	0,004	0,020	0,017	0,250	0,129	49	1225
3	5	3	2	1	0,003	0,018	0,016	0,211	0,190	177	10000
3	5	3	2	2	0,004	0,033	0,017	0,377	0,268	80	3136
3	5	3	2	3	0,003	0,022	0,015	0,674	0,359	138	10000
3	5	3	3	0	0,003	0,014	0,018	0,280	0,146	100	10000
3	5	3	3	1	0,003	0,023	0,017	0,183	0,177	72	5184
3	5	3	3	2	0,006	0,029	0,027	0,407	0,171	34	841
3	5	3	3	3	0,008	0,025	0,041	0,313	0,473	27	484
3	5	4	0	1	0,005	0,028	0,065	0,399	0,253	372	10000
3	5	4	0	3	0,004	0,030	0,026	0,449	0,322	196	10000
3	5	4	1	0	0,004	0,022	0,038	0,255	0,171	241	7396
3	5	4	1	1	0,004	0,019	0,025	0,383	0,144	99	4761
3	5	4	1	3	0,004	0,029	0,034	0,352	0,177	36	1296
3	5	4	1	4	0,004	0,025	0,028	4,921	0,762	150	10000
3	5	4	2	0	0,003	0,015	0,038	0,214	0,222	98	7921
3	5	4	2	1	0,003	0,018	0,017	0,260	0,239	111	10000
3	5	4	2	3	0,003	0,024	0,020	2,659	0,585	129	10000
3	5	4	2	4	0,008	0,025	0,047	0,490	0,189	154	3249
3	5	4	3	0	0,004	0,031	0,029	0,477	0,345	180	10000
3	5	4	3	1	0,021	0,076	0,111	1,080	0,438	22	121
3	5	4	3	3	0,004	0,033	0,028	4,210	0,738	191	10000
3	5	4	3	4	0,004	0,027	0,023	2,365	0,630	117	10000
3	5	4	4	0	0,005	0,021	0,025	1,903	0,351	213	10000
3	5	4	4	1	0,004	0,019	0,035	4,339	0,564	127	10000
3	5	4	4	3	0,004	0,029	0,018	4,206	0,624	109	10000
3	5	4	4	4	0,005	0,044	0,023	1,351	0,841	167	10000

Table 14: Dataset: Airport(2), Configuration: States/States.

7.2.5 Mutagenesis

The problem concerns identifying mutagenic compounds using only the atomic and bond structure of the compounds [31], [19]. Mutagenic compounds are often known to be carcinogenic and cause damage to the DNA. So it is of high interest to the pharmaceutical industry to find out what are the key features in a compound having mutagenic activity.

The dataset used usually for testing ILP techniques and algorithms consist of 230 compounds listed in [4]. The atom and bond structure of the drugs were obtained from the standard molecular modelling package QUANTA. For each compound, the atoms, bonds, bond types (e.g. aromatic, single, double), atom types e.g. aromatic carbon, aryl carbon), and the partial charges on atoms are given.

This data is put into clausal form as follows:

- $bond(compound, atom1, atom2, bondtype)$ stating that $compound$ has a bond of type $bondtype$ between the atoms $atom1$ and $atom2$.
- $atm(compound, atom, element, atomtype, charge)$ stating that in $compound$ $atom$ has element $element$ of type $atomtype$ and partial charge $charge$.

For example

$$atom(127, 127\_1, c, 22, 0.191)$$
$$bond(127, 127\_1, 127\_6, 7)$$

means that in compound 127, atom number 1 is a carbon atom of QUANTA type 22 with a partial charge of 0.101, and atoms 1 and 6 are connected by a bond of type 7 (aromatic). This representation can be used to encode arbitrary chemical structures.

The hypotheses are generated randomly using a fixed schemata, approaching the hypotheses generated in real ILP systems. The varying parameters used for the generation are the size n of the clauses and the number of variables v. The size of the clauses must be greater than the number of variables.

The general form of the hypotheses clauses is:

$$\begin{array}{l} active(Mol), \\ bond(Mol, X_0, X_1, Xint_0), \\ bond(Mol, X_1, X_2, Xint_1), \\ ..., \\ bond(Mol, X_{v-2}, X_{v-1}, Xint_{v-1}), \\ ... \end{array}$$

followed by randomly generated literals of the form $bond(Mol, X_i, X_j, Xint_k)$, where $i, j \in \{0..v-1\}$ and k is increased with each new literal added, starting at $k = v$.

Example A hypothesis clauses for $v = 3$ variables and $n = 4$ literals could be:

$$\begin{array}{l} active(Mol), \\ bond(Mol, X_0, X_1, Xint_0), \\ bond(Mol, X_1, X_2, Xint_1), \\ bond(Mol, X_1, X_0, Xint_2), \\ bond(Mol, X_2, X_0, Xint_3) \end{array}$$

The data used for the example clauses is the real mutagensis data in clausal form, as presented above.

Parameters		*GC*	AllTheta	*DC*	*ST*	*Django*	Res	
lit	*var*	μ (ms)	μ (ms)	μ (ms)	μ (ms)	μ (ms)	pos	total
2	2	0,169	7,257	0,361	0,294	0,334	23000	23000
3	2	0,677	8,212	10,803	15,594	0,614	12190	23000
5	2	1,506	2,280			0,789	2990	23000
10	2	2,702	0,557			0,762	0	23000
20	2	4,380	0,584			0,815	0	23000
3	3	0,969	18,610	17,134	24,849	1,271	14490	23000
5	3	3,554	15,805			2,902	2530	23000
10	3	6,173	8,917			2,253	0	23000
15	3	7,677	5,988			1,603	0	23000
20	3	8,657	0,634			1,397	0	23000
5	5	7,201				4,385	14170	23000
10	5					10,861	313	23000
15	5					4,421	0	23000
20	5					4,540	0	23000
30	5		0,818			2,171	0	23000
50	10					15,198	0	23000
70	10		9,909			4,915	0	23000
100	10		1,600			4,580	0	23000

Table 15: Dataset: MUTAGENESIS, Variing: number of literals *lit* and number of variables *var* in the hypothesis-clause.

Parameters		*GC*	AllTheta	*DC*	*ST*	*Django*	Res	
lit	*var*	σ (ms)	σ (ms)	σ (ms)	σ (ms)	σ (ms)	pos	total
2	2	0,005	61,309	1,930	1,774	0,097	23000	23000
3	2	0,631	24,622	14,157	21,572	0,237	12190	23000
5	2	1,226	8,806			0,481	2990	23000
10	2	1,555	3,620			0,387	0	23000
20	2	5,015	3,913			1,007	0	23000
3	3	1,882	38,590	29,790	40,679	2,528	14490	23000
5	3	3,244	23,830			3,621	2530	23000
10	3	8,040	45,607			8,607	0	23000
15	3	9,818	65,273			2,820	0	23000
20	3	5,221	4,142			1,713	0	23000
5	5	11,611				6,703	14170	23000
10	5					10,364	313	23000
15	5					7,458	0	23000
20	5					9,305	0	23000
30	5		5,240			1,890	0	23000
50	10					49,039	0	23000
70	10		30,646			6,251	0	23000
100	10		10,091			5,128	0	23000

Table 16: Standard deviations. Dataset: MUTAGENESIS, Variing: number of literals *lit* and number of variables *var* in the hypothesis-clause.

7.2.6 Prover9

In the domain of theorem proving, a real theorem prover *prover9* has been used to generate clauses. Various theorems have been proved with it, and the clauses and (first-order) terms that have been generated and tested for subsumption have been flattened and dumped to a separated file. The θ-subsumption algorithms have then been run over these clauses. That way, the algorithms are put in a real theorem proving context.

Table 18 presents the results of the performance of the θ-subsumption algorithms.

Problem instance	*Django* μ (ms)	*GC* μ (ms)	AllTheta μ (ms)	*DC* μ (ms)	*ST* μ (ms)	Results pos	total
a1	2, 100	0, 500	2, 860		4, 160	148	500
a2	2, 220	0, 520	3, 380		4, 360	145	500
AD	0, 000	0, 000	5, 000	0, 000	0, 000	4	4
BA2	0, 220	0, 080	0, 640	2, 800	0, 820	99	500
dep-2b	0, 180	0, 040	0, 520	0, 980	0, 380	63	500
dist-long-short	0, 240	0, 060	0, 660	4, 840	0, 540	37	500
dist-short-long	0, 200	0, 060	0, 560	2, 380	0, 460	45	500
dn1	1, 500	0, 380	2, 240	28, 240	3, 360	105	500
H27d	0, 000	0, 000	0, 000	0, 000	0, 000	2	2
H42	3, 480	0, 920	5, 420		11, 000	201	500
mckenzie	0, 140	0, 040	0, 340	1, 100	0, 400	76	500
MOL-A	0, 600	0, 120	1, 060		1, 700	66	500
mol-ss1	0, 431	0, 062	0, 554	14, 585	1, 354	41	325
na-ring-1	0, 440	0, 140	0, 740	34, 780	1, 640	352	500
oml-4basis	0, 280	0, 080	0, 660	11, 140	1, 640	180	500
omlsax2	0, 780	0, 240	1, 300	41, 860	2, 800	351	500
pair-def	0, 200	0, 060	0, 620	1, 520	0, 540	128	500
quot-comm	0, 060	0, 020	0, 220	0, 420	0, 280	318	500
quot-general	0, 060	0, 020	0, 240	0, 460	0, 360	312	500
quot-xy3b	0, 140	0, 020	0, 280	0, 900	0, 220	313	500
sh1	0, 900	0, 180	1, 360		2, 980	55	500
t4_12	3, 360	1, 760	4, 420		11, 060	408	500
uc	0, 000	0, 000	0, 000	0, 000	0, 000	2	2
x2	0, 000	0, 000	0, 156	0, 313	0, 156	31	64
x3-ring	0, 320	0, 080	0, 580	1, 460	0, 600	287	500
xcb-reflex	1, 600	0, 640	4, 100	11, 400	6, 660	386	500

Table 17: Average time for one subsumption. Datasets: *prover9*.

We can observe, that GC has the best performance in most of the cases. This result has again to be relativated due to the non-completeness of the implementation of GC.

Django is the second best algorithm for this domain, immediately followed by AllTheta.

The θ-subsumption problems generated by the theorem prover are relatively hard problems, compared to the Random domains and the Planning Problems, since the average execution time for a subsumption is about 100 times longer for the theorem proving domain.

7.3 Discussion

Django gave good performances on various domains. It inherits the efficiency of well known CSP algorithms and the computational overhead due to the transformation from a θ-subsumption problem to a CSP-problem is often negligible. Django may be qualified the best allaround subsumer for our tested cases.

Besides, for θ-subsumption problems that tends to give a negative answer (C does not θ-subsume D), AllTheta outperforms Django up to one order

Problem instance	*Django* σ (ms)	*GC* σ (ms)	AllTheta σ (ms)	*DC* σ (ms)	*ST* σ (ms)	Results pos	total
a1	0,235	0,594	0,721		5,591	148	500
a2	0,308	0,613	0,766		5,949	145	500
AD	0,009	0,006	0,008	0,063	0,008	4	4
BA2	0,074	0,286	0,105	7,252	2,163	99	500
dep-2b	0,024	0,042	0,062	1,306	0,342	63	500
dist-long-short	0,019	0,038	0,393	12,989	0,706	37	500
dist-short-long	0,014	0,016	0,055	3,729	0,570	45	500
dn1	0,240	0,665	4,219	76,646	4,817	105	500
H27d	0,040	0,056	0,065	0,647	0,151	2	2
H42	0,273	0,625	0,801		10,502	201	500
mckenzie	0,024	0,033	0,071	2,571	0,400	76	500
MOL-A	0,104	0,163	0,415		2,849	66	500
mol-ss1	0,063	0,085	0,254	101,483	2,418	41	325
na-ring-1	0,145	0,251	0,420	547,414	2,689	352	500
oml-4basis	0,025	0,039	0,103	15,157	2,809	180	500
omlsax2	0,306	0,169	0,208	194,226	4,791	351	500
pair-def	0,022	0,023	0,049	7,284	0,632	128	500
quot-comm	0,004	0,004	0,009	0,933	0,781	318	500
quot-general	0,007	0,009	0,307	0,975	0,906	312	500
quot-xy3b	0,022	0,091	0,079	5,624	0,378	313	500
sh1	1,075	0,352	0,969		5,590	55	500
t4_12	0,547	0,885	8,625		13,542	408	500
uc	0,014	0,005	0,001	0,006	0,004	2	2
x2	0,010	0,009	0,028	0,279	0,035	31	64
x3-ring	0,028	0,035	0,070	1,931	0,812	287	500
xcb-reflex	0,068	0,228	7,731	21,269	7,984	386	500

Table 18: Standard deviations. Datasets: *prover9*.

of magnitude faster. AllTheta invests more time in advance for constructing graph-contexts for literals which shows up to be very efficient in the negative cases.

The κ-parameter can give us a clue how probable a successful subsumption is. The lower the κ value, the less probable.

Although the authors of Django claimed that Django is “orders of magnitudes” faster than GC, this could only be observed in some rare cases. GC gives competitive results but is often solely the second choice. It has to be pointed out again, that the implementation of GC is not complete, and the correction might render GC slower.

The older algorithms ST and DC stay ofter far behind in computational cost than the others. ST relies heavily on the order of the literals. And the literals cannot be sorted optimally without doing a full subsumption-like check.

DC did rarely outperform ST. This may partly be due to our implementation of DC which could surely be further optimized. (By the way: we also implemented ST.) No other experimental result than ours is available to the best of our knowledge for DC. What is known is the better worse case complexity of DC. So even an optimized implementation of DC may be slower in the average case. One has to note that in our real domains, the θ-subsumption problems were not often decomposable into simpler independent subproblems—which is the key idea of DC.

For the problem of finding all solutions, AllTheta is best for detecting the negative cases. FASϑ is clearly better suited if many substitutions are expected as result. This is particularly the case for real problems from the planning domain for calculating successor and predecessor states.

8 Conclusion

We have given a survey of the most popular θ-subsumption algorithms based on a deep study of the literature.

These algorithms were developed for different purposes. A unified framework and logical formalism was formally defined in order to have a common basis for comparison. Some authors imposed special restrictions on the input clauses, so that adaptations had to be applied.

Most of the algorithms were not presented in a formal way. Mainly the new feature were described in depth but not completely formally. For that reason, the correctness proofs given by the authors were often only proof-sketches and correctness ideas. We provided formal proofs for the correctness of the main parts of the algorithms where such were not available.

Furthermore, we provided a full scale experimental analysis with data from various domains. Most of the data had either been transformed into clausal form or been generated by specially developed generators.

Particular care has been put in either having data that was extracted from real problems (e.g. we extracted exactly those clauses that were really check for subsumption in a real theorem proving engine); or having artificial data very similar to real cases.

We analysed the results and provided arguments when a particular subsumption implementation should best be used.

Besides, this experimental analysis showed up to be useful for uncovering some hidden bugs in the implementations.

8.1 Future work

In some cases however, a deeper analysis would be required to determine the best algorithm.

A method would be to take the output of our experimental result as *input* to a learning system. It is not said that an easy to evaluate heuristic (the computational cost of the heuristic should only be some negligible part of the overall cost for the whole subsumption) could be derived from such a learning.

Another way could be to *guess* a function (in the spirit of the κ-parameter) with the basic parameters that are common to all domains, namely the size of the clause, the number of variables, the number of constants, the arity. Such guesses should be preceded by probabilistic considerations in order to find plausible useful functions.

References

[1] Craig Boutilier, Thomas Dean, and Steve Hanks. Decision-theoretic planning: Structural assumptions and computational leverage. *Journal of Artificial Intelligence Research*, 11:1–94, 1999.

[2] Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *IJCAI*, pages 690–700, 2001.

[3] C.-L. Chang and R. C.-T. Lee. Artificial intelligence, symbolic logic, and theorem proving. In C.-L. Chang and R. C.-T. Lee, editors, *Symbolic Logic and Mechanical Theorem Proving*, pages 1–5. Academic Press, New York, 1971.

[4] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. *Structure-Activity Relationship of Mutagenic Aromatic and Heteroaromatic Nitro Compounds*, volume 34. 1991.

[5] Luc Dehaspe. Frequent pattern discovery in first-order logic. *AI Commun.*, 12(1-2):115–117, 1999.

[6] Nicola Di Mauro, Teresa Basile, Stefano Ferilli, Floriana Esposito, and Nicola Fanizzi. An exhaustive matching procedure for the improvement of learning efficiency. In T. Horváth and A. Yamamoto, editors, *Inductive Logic Programming: 13th International Conference (ILP03)*, volume 2835 of *LNCS*, pages 112–129. Springer Verlag, 2003.

[7] B. Dolsak and S. Muggleton. The application of Inductive Logic Programming to finite element mesh design. In S. Muggleton, editor, *Inductive Logic Programming*. Academic Press, London, 1992.

[8] Norbert Eisinger. Subsumption and connection graphs. In J. H. Siekmann, editor, *GWAI-81, German Workshop on Artificial Intelligence, Bad Honnef, January 1981*, pages 188–198. Springer, Berlin, Heidelberg, 1981.

[9] Stefano Ferilli, Nicola Fanizzi, Nicola Di Mauro, and Teresa Basile. Efficient theta-subsumption under object identity. *AI*IA Workshop*, 2002.

[10] Attilio Giordana and Lorenza Saitta. Phase transitions in relational learning. *Mach. Learn.*, 41(2):217–251, 2000.

[11] G. Gottlob and A. Leitsch. On the efficiency of subsumption algorithms. *J. ACM*, 32(2):280–295, 1985.

[12] Georg Gottlob. Subsumption and implication. *Inf. Process. Lett.*, 24(2):109–111, 1987.

[13] Malte Helmert. New complexity results for classical planning benchmarks. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 52–61, 2006.

[14] Steffen Hölldobler, Edgar Karabaev, and Olga Skvortsova. Flucap: A heuristic search planner for first-order markov decision processes. *Journal of Artificial Intelligence Research*, 27:419–439, 2006.

[15] Steffen Hölldobler and Josef Schneeberger. A new deductive approach to planning. *New Gen. Comput.*, 8(3):225–244, 1990.

[16] Deepak Kapur and Paliath Narendran. Np-completeness of the set-unification and matching problems. *Proceedings of the Eighth International Conference on Automated Deduction*, pages 289–495, 1986.

[17] E. Karabaev, Rammé G., and O. Skvortsova. Efficient symbolic reasoning for First-Order MDPs. In *Proceedings of the ECAI'2006 Workshop on Planning, Learning and Monitoring with Uncertainty and Dynamic Worlds*, Riva del Garda, Italy, August 2006.

[18] J-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pages 97–106. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[19] Ross D. King, Michael J. E. Sternberg, and Ashwin Srinivasan. Relating chemical activity to structure: An examination of ILP successes. *New Generation Computing*, 13(3 and 4):411–433, 1995.

[20] Jérôme Maloberti and Michèle Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55:137–174, 2004.

[21] William McCune. Otter 3.3 reference manual, 2003.

[22] Ruy Luiz Milidiu, Frederico dos Santos Liporace, and Carlos Jose P. de Lucena. In *In Workshop on the Competition*, Trento, Italy, June, 2003.

[23] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.

[24] Shan-Hwei Nienhuys-Cheng, R. De Wolf, and Ronald de Wolf. *Foundations of Inductive Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[25] G.D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[26] J. R. Quinlan. Learning logical definitions from relations. *Mach. Learn.*, 5(3):239–266, 1990.

[27] Céline Rouveirol. Flattening and saturation: Two representation changes for generalization. *Mach. Learn.*, 14(2):219–232, 1994.

[28] Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient algorithms for θ-subsumption (revised version). *Inductive Logic Programming, 6th International Workshop, Selected Papers, LNAI 1314*, pages 212–228, 1996.

[29] Olga Skvortsova. θ-subsumption based on object context. 2006.

[30] Edgar Sommer, Katharina Morik, Jean-Michel André, and Marc Uszynski. What online machine learning can do for knowledge acquisition—a case study. *Knowl. Acquis.*, 6(4):435–460, 1994.

[31] A. Srinivasan, S. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: ILP experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pages 217–232. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[32] Rona B. Stillman. The concept of weak substitution in theorem-proving. *J. ACM*, 20(4):648–667, 1973.

[33] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.