Fast Estimation of First-Order Clause Coverage through Randomization and Maximum Likelihood

Ondřej Kuželka and Filip Železný {KUZELO1,ZELEZNY}@FEL.CVUT.CZ
Czech Technical University in Prague, Technická 2, 166 27 Prague 6, Czech Republic

Abstract

In inductive logic programming, θ-subsumption is a widely used coverage test. Unfortunately, testing θ-subsumption is NP-complete, which represents a crucial efficiency bottleneck for many relational learners. In this paper, we present a probabilistic estimator of clause coverage, based on a randomized restarted search strategy. Under a distribution assumption, our algorithm can estimate clause coverage without having to decide subsumption for all examples. We implement this algorithm in program ReCovEr. On generated graph data and real-world datasets, we show that ReCovEr provides reasonably accurate estimates while achieving dramatic runtimes improvements compared to a state-of-the-art algorithm.

1. Introduction

In most inductive logic programming (ILP) algorithms, learned hypothesis are (sets of) first-order clauses. Usually, θ-subsumption is used to test whether a clause entails an example. Since ILP systems need to evaluate large numbers of clauses during hypothesis search, efficiency of the subsumption procedure is one of the crucial factors for performance of learning. Unfortunately, deciding θ-subsumption is an NP-complete problem.

One line of research has focused on developing algorithms for this problem using sophisticated heuristics from the field of constraint satisfaction problems (CSP). Maloberti et Sebag (2004) exploited the correspondence of θ-subsumption with CSP to develop the algorithm Django. Django is currently considered the fastest subsumption checker, outperforming traditional techniques (based on the Prolog unification mechanism) by orders of magnitude. Therefore we employ Django in comparative experiments later in this paper. Another stream of research dealt with incomplete heuristic algorithms for θ-subsumption. Sebag et al. (1997) presented a tractable approximation of θ-subsumption called stochastic matching. Arias et al. (2007) implemented a randomized table-based method.

Unlike the mentioned incomplete heuristic algorithms, our approach uses a complete, albeit randomized, subsumption procedure that correctly decides both subsumption and non-subsumption if given sufficient finite time. Our ultimate estimation of the clause coverage (i.e. the number of subsumed examples) is however an approximation, rapidly achieved by restarting the subsumption procedure each time with a bounded runtime. Subsequent restarts generate an integer sequence, from which the coverage is estimated by maximum likelihood.

Randomized restarted strategies, exploited in our work, have been extensively studied in the past decade (Gomes et al., 2000). They have been demonstrated to be extremely useful for solving many hard combinatorial problems such as satisfiability of boolean formulas or for solving constraint satisfaction problems. Reported reduction in runtimes are often in orders of magnitude. Randomized restarted strategies have been also used in inductive logic programming (Železný et al., 2006), however, not for subsumption checking. Rather, restarts were applied on the clause-search procedure.

This paper is organized as follows. In Section 2 we formalize subsumption and expose the basic algorithms employed as building blocks in our estimation approach. In Section 3 we conduct a preliminary motivating study of runtime distribution. The estimation algorithm is then developed in Section 4. In Section 5, we compare our algorithm with Django on synthetic

Appearing in *Proceedings of the 25th International Conference on Machine Learning*, Helsinki, Finland, 2008. Copyright 2008 by the author(s)/owner(s).

Algorithm 1 *SubsumptionCheck*(C, e): A simple subsumption check algorithm

```
Input: Clause C, example e;
if C ⊆ e then
  return YES
else
  Choose variable V from C using a heuristic function (see main text)
  for ∀S ∈ PossibleSubstitutions(V, C, e) (see main text) do
    C' ← Substitute V with S
    if ∀W ∈ Adjacency(V) : PossibleSubstitutions(W, C', e) ≠ ∅ then
      SearchedNodes ← SearchedNodes + 1
      UsedTerms ← UsedTerms ∪ {S}
      if SubsumptionCheck(C', e) = YES then
        return YES
      end if
      UsedTerms ← UsedTerms\{S}
    end if
  end for
  return NO
end if
```

and on real-life data. Section 6 concludes the paper.

2. Preliminaries

2.1. Language

In the rest of the paper we assume for simplicity that hypotheses C are clauses without function and constant symbols and examples e are ground clauses. When needed, clauses will be treated as atom sets, e.g. for two clauses C and D, $C \subseteq D$ will denote that C contains all literals contained by D. θ-subsumption is defined as follows

Definition We say that clause C θ-subsumes clause D (denote $C \prec_\theta D$) iff there exists a substitution θ such that $C\theta \subseteq D$.

2.2. Subsumption Algorithm

We consider a simple heuristic algorithm (Algorithm 1) for verifying whether a clause C subsumes an example e. Similarly to Django (Maloberti & Sebag, 2004) this algorithm is inspired by the CSP framework. It is a backtracking search algorithm with forward checking, a variable selection heuristic and randomization. The heuristic function aims at choosing variables whose substitution makes it likely that an inconsistency, if one exists, is detected soon. For a variable V, the function computes the sum of occurrences of variables in clause C that have already been grounded and that share at least one literal with V. This sum is then multiplied by $1 + \frac{1}{D}$, where D is an upper bound on the size of the domain of V computed in the initialisation phase of the algorithm's run. The variable which maximizes this function is selected; in case of a tie, a random choice is made with uniform probability among the highest scoring variables. Function $PossibleSubstitutions(V, C, e)$ returns all terms S (in a random order), which can be substituted for V satisfying that all literals $l \in C$ remain consistent with e. The function prunes away a subset of possible groundings for V whose inclusion in θ would imply $C\theta \not\subseteq e$. In general though, not all such groundings are detected by the function.

3. Subsumption Test Runtimes

We first aimed at obtaining a domain-independent runtime distribution of the subsumption algorithm and thus conducted preliminary experiments with randomly generated hypotheses and examples from the domain of oriented colored graphs. In the clausal representation, each graph acquires the form of a definite clause

$$h \leftarrow l_1 \wedge l_2 \wedge \ldots$$

where h is a fixed head and l_i are first-order atoms, each being one of $edge(t_1, t_2)$, $black(t_3)$, $red(t_4)$. In hypotheses, t_i are variables, in examples these are constants.

For generality, we devised two different graph generators. The first generator generates Erdos-Rényi random graphs where any two vertices are connected with a pre-set probability c (by an edge of a random orientation). The second produces scale-free ("small world") graphs. Here, the graph grows until some desired size is reached; at any step a vertex is added and connected to k vertices already present in the graph. An edge is attached to a vertex with probability increasing with the number of edges already connected to the vertex. In both algorithms, all vertices are colored as black with probability 0.5 and red otherwise. We will refer to the parameter c (k, respectively) of a random uniform (scale-free, respectively) graph as the *connectivity* of the graph.

We subjected Algorithm 1 to experiments with random sets of hypotheses and examples, under various settings of n and c (n and k, respectively), where n denotes number of vertices in the underlying graph and c and k are parameters of the random graphs explained above. Here, we review our principal findings about the respective runtime distributions, since they motivate the design of the estimation algorithm in the next section.

Our first objective was to verify the presence of heavy tails in the runtime distributions $F(t)$. For $t > 0$, the number $F(t)$ is the probability that the tested algorithm resolves a random subsumption instance in no more than t units of time, corresponding to the

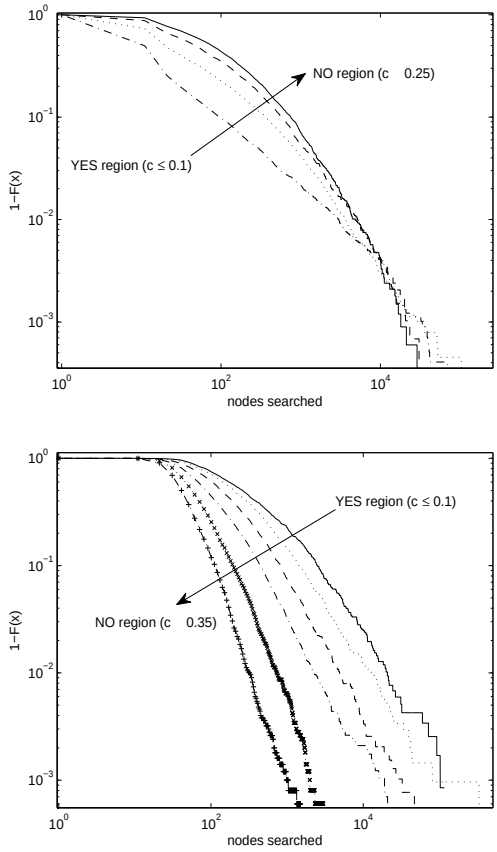

Figure 1. **Top:** The runtime distributions for satisfiable instances with hypotheses built using the Erdos-Rényi random graph generator with $n = 15$ vertices and connectivity consecutively $c \in \{0.1, 0.15, 0.2, 0.25\}$. The graphs corresponding to examples had $n = 50$ vertices and connectivity $p = 0.3$. **Bottom:** The subsumption test runtime distributions for unsatisfiable instances with hypotheses with connectivity consecutively $c \in \{0.15, 0.2, 0.25, 0.3, 0.35, 0.4\}$.

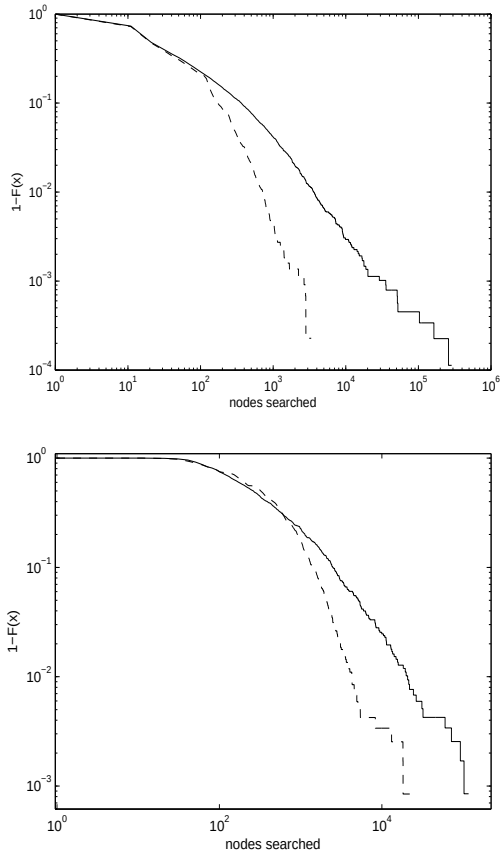

Figure 2. Effect of the restarted strategy for satisfiable (top) and unsatisfiable instances (bottom). The random graphs corresponding to hypotheses had $n = 15$ vertices and connectivity $c = 0.15$. In both cases, the random graphs corresponding to examples had $n = 50$ vertices and connectivity $c = 0.3$. Both hypotheses and examples were randomly generated by the Erdos-Rényi generator.

number of explored search nodes. Informally, a heavy-tailed distribution indicates the non-negligible probability of subsumption instances on which the checking algorithm gets stuck for an extremely long runtime. For example, a heavy tail is exhibited if $1-F(t)$ decays at a power-law rate, i.e. slower than standard distributions which decay exponentially. The presence of a heavy tail in an empirically obtained runtime distribution $F(t)$ is usually checked graphically, by plotting $1 - F(t)$ against t on a log-log scale. In the case of a power-law distribution, this plot then acquires a linear shape (Gomes et al., 2000).

A series of experiments in the phase transition framework (Giordana & Saitta, 2000), which we have performed, revealed a systematic progression from heavy-tailed regimes corresponding to configurations located in the YES region of the phase transition spectrum to non-heavy-tailed regimes corresponding to configurations located in the NO region. This observation agrees with the previous study (Gomes et al., 2005). This progression is shown in Fig. 1 for the Erdos-Rényi graph data. The same trends were observed for the small-world graph data. The plotted runtime distributions refer to subsumption checks between hypotheses with fixed numbers of vertices and connectivity changing among particular distributions, and examples with fixed numbers of vertices and with fixed connectivity. The runtime distributions plotted in the top panel of Fig. 1 refer to satisfiable problem instances, i.e. those where the hypotheses θ-subsume the examples. The distributions in the bottom panel of Fig. 1 refer to unsatisfiable problem instances.

Due to the observed presence of heavy tails in a range of parameters, we next assessed the impact of restarts. For this sake we designed a complete restarted randomized subsumption algorithm, which repeatedly executes Algorithm 1. At each execution $n = 1, 2, \ldots$, the number of search nodes in the Algorithm 1 is bounded by some pre-defined number $R(n)$. This loop is terminated once answer YES or NO is obtained from Algorithm 1. Completeness of this restarted approach is guaranteed by the assumption that $R(n) \to \infty$ as $n \to \infty$. Recall that randomization is facilitated by tie-breaking in the heuristic function used in Algorithm 1 and by randomization of the value ordering.

The basic trends we observed for all tested parameter values are represented by Fig. 2: (i) restarts significantly reduce runtime expectation for both satisfiable and unsatisfiable instances, (ii) unsatisfiable instances take much longer to prove in the restarted approach. Observation (i) alone motivates to use the restarted variant of Algorithm 1 as a fast complete method for subsumption testing. We explore this idea elsewhere (Kuželka & Železný, 2009), whereas this paper addresses observation (ii). This observation is easily explained: while satisfiability can in principle be shown in any single restart, unsatisfiability can only be shown after n restarts making $R(n)$ sufficiently high. We would like to avoid the runtime components corresponding to $R(n)$ series growing to excessive values.

4. ReCovEr: A Restarted Coverage Estimator

We first explain the intuition underlying ReCovEr. We are given a clause C, and example set E and we would like to estimate the coverage $cov(C, E) = |\{e \in E \mid C \prec_\theta e\}|$. Let us run Algorithm 1 on C and e, successively for all $e \in E$. For each e, we however stop the algorithm if no decision has been made in R steps. Let $\mathcal{E} \subseteq E$ be the subset of examples proven to be subsumed by C in this experiment. Denote $s_1 = |\mathcal{E}|$. We now remove all examples in $\mathcal{E}$ from E and repeat this experiment, obtaining analogical number s_2. Further such iterations generate numbers s_3, s_4, etc. Clearly, for the desired value $cov(C, E)$, we have that $cov(C, E) = \lim_{j\to\infty} S_j$ where $S_j = \sum_{i=1}^{j} s_i$. Under a certain assumption, the series S_j is geometrical rather than arbitrary. The main idea of ReCovEr is that the limit of S_j for $j \to \infty$ can thus be estimated by extrapolating the series from its first few elements $S_1, S_2, \ldots$. Thus we achieve a coverage estimate without excessive effort to refute subsumption for the examples not subsumed by C.

In order to precisely derive an estimation algorithm following the above idea, we first need to make the following assumption.

Algorithm 2 *ReCovEr*(C, E, R, M, Δ): Algorithm for coverage estimation

Input: Clause C and set of examples E, Integers R ('cutoff'), M, Δ;

```
tries ← 0
Unknown ← Examples
CoveredInIthTry ← []
repeat
  tries ← tries + 1
  CoveredInThisTry ← 0
  for ∀E ∈ Unknown do
    Answer ← Run SubsumptionCheck(C, E) with number of
    searched nodes limited to R
    if Answer = PositiveMatching then
      CoveredInThisTry ← CoveredInThisTry + 1
      Unknown ← Unknown\E
    end if
  end for
  CoveredInIthTry[tries] ← CoveredInThisTry
until TerminationCondition
return LikelihoodEstimate(tries)
```

Assumption 4.1 *Given a clause C and a set of examples E, the probability p that Algorithm 1 finds a solution (i.e. returns YES as its answer) before it explores more than R nodes of the search tree, is the same for all $e \in E$ such that C subsumes e.*

In other words, we assume that properties of particular examples such as their size are not dramatically different. The assumption will be empirically validated in the next section.

We assume a given clause C and we fix a constant cutoff value R. In the first step, for each $e \in E$ we run $SubsumptionCheck(P, e)$ (Algorithm 1), stopping it as soon as the number of searched nodes has reached R. Then, after $|E|$ restarts (each time with a different $e \in E$), we can derive the probability that the algorithm has produced exactly m_1 'YES' responses in this first step. In particular, this probability $P(m_1)$ is

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

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

$$P(m_2|m_1) = \binom{A-m_1}{m_2} p^{m_2}(1-p)^{A-m_1-m_2} \qquad (2)$$

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

$$P(m_i|m_{i-1},\ldots,m_1) = \binom{A-\sum_{j=1}^{i-1} m_j}{m_i} p^{m_i}(1-p)^{A-\sum_{j=1}^{i} m_j} \quad (3)$$

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

$$P(m_1,\ldots,m_k) = \prod_{i=1}^{k} P(m_i|m_{i-1},\ldots,m_1) \quad (4)$$

Substituting for $P(m_i|m_{i-1},\ldots,m_1)$ from Eq. 3 and taking the logarithm Eq. 4 results in

$$\ln\left(P(m_1,\ldots,m_k)\right) = \sum_{i=1}^{k}(\alpha + m_i \ln p + \beta) \quad (5)$$

where

$$\alpha = \ln\binom{A-\sum_{j=1}^{i-1} m_j}{m_i}$$

and

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

To find the parameters A and p for which $P(m_1,\ldots,m_k)$ is maximized, we take the partial derivative of Eq. 5 with respect to p and then find its roots, yielding

$$p = \frac{\sum_{i=1}^{k} m_i}{\sum_{i=1}^{k} m_i + \sum_{i=1}^{k}\left(A - \sum_{j=1}^{i} m_j\right)} \quad (6)$$

Finding the global maximum of $P(m_1,\ldots,m_k)$ from Eq. 4 on the set

$$D = \{(A,p)|A \in \{1,2,\ldots,|E|\} \wedge p \in [0;1]\} \quad (7)$$

is now straightforward, since using (6) we can find the maximum on every line

$$L_i = \{(i,p)|p \in [0;1]\} \quad (8)$$

The maximum on line L_i is located either at the value of p given by (6) or at one of the borders of L_i. It then suffices to evaluate (4) at these three points of L_i for every i ($1 \leq i \leq |E|$). The estimate of A then equals the index i of the L_i on which the maximum is located.

The described estimator is used in ReCovEr (Algorithm 2). The question how to choose k, i.e. how long a sequence $(m_1,\ldots,m_k)$ should be generated as the input to the estimator, is tackled iteratively: the sequence is being extended until a termination condition is met. We have considered several termination conditions, of which two turned out to be quite useful. The first termination condition stops generating the sequence when two subsequent estimates differ by less than some Δ_e, specified as a parameter. The second termination condition stops generating the sequence when estimate and number of examples already shown to be covered by the clause differ again by less than some Δ_c, which ensures that the estimator will never overestimate the actual coverage by more than Δ_c. A minimum length M of the sequence is however imposed in both previous cases, to avoid premature estimates coinciding by chance.

Another degree of freedom in Algorithm 2 is the cutoff R, which may significantly affect the performance of the restarted algorithm. A heuristic method suggests itself that first tries to find a suitable cutoff. Unlike Algorithm 2 it starts with a base cutoff value, and then doubles it after every single restart. If at any restart Algorithm 1 with cutoff set to R covers fewer examples than the same algorithm at previous restart with cutoff set to $\frac{R}{2}$, then we can accept cutoff $\frac{R}{2}$.

5. Experiments

In this section, we first investigate the sensitivity of ReCovEr to a violation of Assumption 4.1. Then we evaluate its performance and precision on graph data generated by the two random graph generators described in Section 2 and on real-world data from organic chemistry and from engineering. We compare performance of ReCovEr with that of the state-of-the-art θ-subsumption algorithm Django.

5.1. Sensitivity Analysis

We addressed the validity of Assumption 4.1 and the impact of its violation onto ReCovEr's precision. We first experimented with the settings from Section 3 with parameters of generated hypotheses $c = 0.35$, $n = 10$ and parameters of generated examples $c = 0.3$, $n = 50$, and fitted the actual distribution of p with Beta distributions. This resulted in Beta distributions with standard deviation extending up to about 0.25 (i.e. 25% of the p's range). These distributions are plotted in dashed lines in Fig. 3. To investigate ReCovEr's sensitivity to such deviations, we assumed to have $n = 100$ examples, of which 50 were covered by a clause C. Further, probabilities p_i that Algorithm

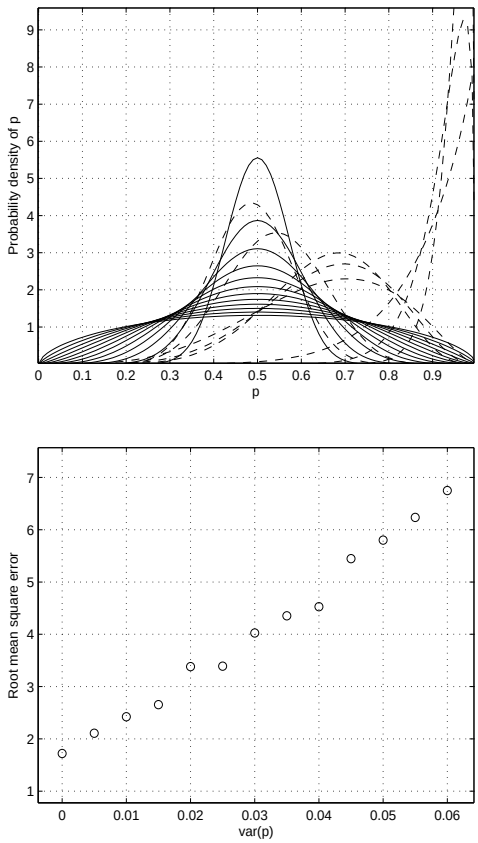

Figure 3. **Top:** Beta distributions with mean $\mu = 0.5$ and variance consecutively $0, 0.005, \ldots, 0.06$, which model the distribution of p (solid lines). Beta distributions fitted to actual probabilities are shown in dashed lines. **Bottom:** Dependence of root mean square error of ReCovEr's estimates on the variance of p.

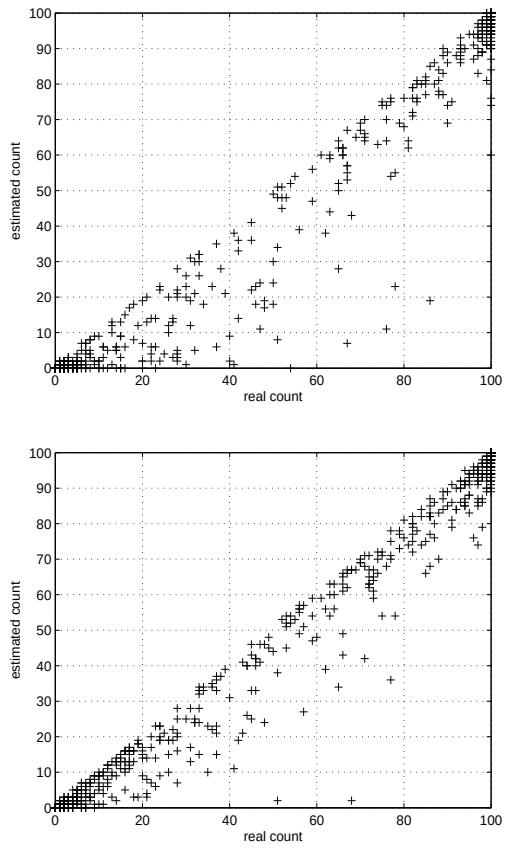

Figure 4. Precision of ReCovEr (Algorithm 2) presented as 1000 points with coordinates (estimated coverage, actual coverage). Hypotheses and examples were generated by the Erdos-Rényi random graph generator with $c = 0.3$, $n = 15$ for hypotheses and $c = 0.3$, $n = 100$ for examples. The 1000 estimates correspond to 1000 different hypotheses tested on a pre-fixed set of 100 examples. **Top:** Base value for cutoff is $R = 100$. **Bottom:** Base value for cutoff is $R = 200$.

1 finds a solution for a covered example e_i in time less than R were sampled from the Beta distribution with given mean $\mu = 0.5$ and variance consecutively $0, 0.005, \ldots, 0.06$ (i.e. growing up to the 25% standard deviation). Then, we simulated ReCovEr's estimation procedure on these data. The top panel of Fig. 3 displays the beta distributions (solid lines) from which probabilities p_i were sampled. We used the stopping condition based on difference of estimate and lower bound, the parameters were $R = 75$, $M = 3$, $\Delta = 1$.

The bottom panel of Fig. 3 displays the dependence of root mean square error on the variance of the beta distributions in the top panel. It is encouraging to see that the mean *squared error* grows roughly *linearly* with growing variance in p's distribution, indicating ReCovEr's robustness towards this variance. Of course, the ultimate judge of whether this dependence is acceptable is the extent to which a learning algorithm based on ReCovEr would be affected by the estimation imprecision caused by the estimation. This is studied further.

5.2. Experiments with Generated Graph Data

Figure 4 demonstrates the precision of ReCovEr on the graph data generated by the Erdos-Rényi generator by showing 1000 pairs (estimated coverage, actual coverage). Hypotheses and examples were generated with $c = 0.3$ for hypotheses and $c = 0.3$ for examples. The graphs corresponding to hypotheses had 15 vertices, and the graphs corresponding to examples had 100 vertices. The top panel refers to estimates obtained by Algorithm 2 enhanced by cutoff selection

with base cutoff $R = 100$, while the bottom panel refers to estimates obtained by the same algorithm with base cutoff $R = 200$. A bias towards coverage under-estimation can be observed, as well as a positive effect of the higher base cutoff on estimation precision.

Table 1 shows average runtimes of ReCovEr and Django. Table 2 shows average runtimes of Django and ReCovEr on artificial graph data with small-world topology generated by Algorithm 5. In this case, the graphs underlying the hypotheses had 15 vertices and their connectivity was $k = 4$. The graphs underlying the examples had 100 vertices and connectivity $k = 20$. A dramatic speedup from Django's runtime is exhibited in both cases.

Note that there is no immediate reason to avoid the under-estimation bias because coverage is usually tested on two example sets (positive and negative). The two results are usually subtracted thus (mostly) canceling the bias. Whether the observed estimation variance is tolerable for the task of clause *ranking* usual in inductive logic programming is the subject of the experiments in the next section.

Algorithm	Avg. Time [s]
ReCovEr, $R = 100$	6.7
ReCovEr, $R = 200$	12.5
Django	483.2

Table 1. Average coverage test runtimes for the configuration from Fig. 4.

Algorithm	Avg. Time [s]
ReCovEr, $R = 100$	8.9
ReCovEr, $R = 200$	16.3
Django	519.8

Table 2. Average coverage test runtimes for the configuration with small world graph data.

5.3. Experiments with Real-World Data

In order to assess performance in conditions of a real-life learning setting, we decided not to generate clauses entirely randomly. Our intention was to simulate general principles of clause production in an inductive logic programming system, while avoiding an overfit to a specific clause search strategy (which would e.g. be a result of adhering to a specific heuristic function for selecting literals). Thus we developed a simple relational learner, which we use for further experiments with ReCovEr. The learner (Algorithm 3) is a randomized variation of a specific-to-general beam search. It starts with the most specific clause $\bot$ and at each search step, it generates at least $n \cdot |Beam|$ new hypotheses by removing random subsets of literals from the hypotheses already present in $Beam$. The output of the algorithm is one best clause, which is why we assess its quality through precision and recall.

Algorithm 3 $Learner(\bot, p, BeamSize, Tries)$: A Clause Learner

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

$Beam \leftarrow \{\bot\}$
$BestClause \leftarrow \bot$
repeat
  $Candidates \leftarrow Beam$
  for $\forall h_i \in Beam$ **do**
    for $i = 1 \ldots BeamSize$ **do**
      $GenerateClause(h_i)$
      $C \leftarrow$ connected components of c
      Evaluate each $c_i \in C$
      $Candidates \leftarrow candidates \cup C$
    end for
  end for
  for $\forall h \in Candidates$ such that h is estimated to be better than $BestClause$ **do**
    if h is shown to be better than $BestClause$ by a deterministic subsumption algorithm **then**
      $BestClause \leftarrow h$
    end if
  end for
  Choose $BeamSize$ best hypotheses from $Candidates$ and add them to $Beam$
  $Explored \leftarrow Explored + 1$
until $Beam = \{\}$ or $Explored = Tries$

The first set of experiments, which we have conducted with Algorithm 3, deals with the Mutagenesis dataset (Srinivasan et al., 1996). This dataset consists of descriptions of 188 organic molecules, which are marked according to their mutagenicity. In our experiments, we used only the information about atom-bond relationships and about types of atoms. We did not consider numerical parameters such as *lumo* or *logp*. Our relational-logic representation of these molecules consisted of ternary literals for atomic bonds $bond(at1, at2, bondType)$, unary literals representing types of particular bonds and unary literals for atom types. We have considered three variants of relational logic description of the molecules, with growing complexity (size of examples). The first version **Muta-v1** uses a naive representation. Here, each molecular bond is represented by a single literal $bond(at1, at2, bondType)$, thus imposing a bond orientation (atom order) chosen at random. The second source of imprecision of this representation is that two variables in a clause may represent the same (chemical) atom, which does not make intuitive sense. The second version **Muta-v2** deals with the first source of imprecision, as it represents every atomic bond with a pair of literals $bond(at1, at2, bondType)$ and $bond(at2, at1, bondType)$. The third version **Muta-**

v3 solves the second source of imprecision by adding literals *different*(a, b) for all pairs of atom-representing constants a, b.

The second set of experiments pertains to class-labeled CAD data (product structures) described in (Žáková et al., 2007), consisting of 96 CAD examples each containing several hundreds of first-order literals.

The main observation provided by the experiments is that ReCovEr becomes quickly superior to Django as the example size grows, whereas the two algorithms do not significantly differ in terms of the training-set[1] accuracy of the discovered clauses. It is interesting to note that Django's poor runtime performance on the learning tasks with large examples (**CAD** data and **Muta-v2**) was often due to occasional subsumption cases. Clearly, this is a manifestation of heavy tails present in Django's runtime distribution. Unlike Django, ReCovEr was exhibiting steady performance.

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

Table 3. Average runtimes of the learner (Algorithm 3, $p = 0.75$, $Tries = 10$) for real-world datasets.

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

Table 4. Quality of learned hypotheses for ReCovEr

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

Table 5. Quality of learned hypotheses for Django

6. Conclusions

In this paper, we have introduced ReCovEr, an algorithm exploiting restarts for a maximum-likelihood based estimation of clause coverage. ReCovEr avoids heavy tails as well as laborious proving of certain unsatisfiable subsumption instances. We have shown that ReCovEr provides favorable runtimes while achieving reasonable precision, which is illustrated by experiments on synthetic graph data and on real-life data from organic chemistry and engineering. In future work we mainly want to develop theoretical bounds for ReCovEr's estimation precision.

[1] As this paper is not concerned with improving generalization performance, we did not measure accuracies on hold-out test sets.

References

Giordana, A., & Saitta, L. (2000). Phase transitions in relational learning. *Machine Learning*, *41*, 217–251.

Gomes, C. P., Fernández, C., Selman, B., & Bessière, C. (2005). Statistical regimes across constrainedness regions. *Constraints*, *10*, 317–337.

Gomes, C. P., Selman, B., Crato, N., & Kautz, H. A. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, *24*, 67–100.

Kuželka, O., & Železný, F. (2009). A restarted strategy for efficient subsumption testing. *Fundamenta Informaticae, spec. issue on multi-relational data mining. (Accepted).*

Maloberti, J., & Sebag, M. (2004). Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, *55*, 137–174.

Marta Arias, Roni Khardon, J. M. (2007). Learning horn expressions with logan-h. *Journal of Machine Learning Research*, *8*, 549–587.

Sebag, M., & Rouveirol, C. (1997). Tractable induction and classification in first-order logic via stochastic matching. *IJCAI97* (pp. 888–893). MK.

Srinivasan, A., Muggleton, S., Sternberg, M. J. E., & King, R. D. (1996). Theories for mutagenicity: A study in first-order and feature-based induction. *Artificial Intelligence*, *85*, 277–299.

Železný, F., Srinivasan, A., & Page, D. (2006). Randomised restarted search in ILP. *Machine Learning*, *64*, 183–208.

Žáková, M., Železný, F., Garcia-Sedano, J., Tissot, C. M., Lavrač, N., Křemen, P., & Molina, J. (2007). Relational data mining applied to virtual engineering of product designs. *Procs of the 16th Int. Conference on Inductive Logic Programming* (pp. 439–453). Springer.