# A Restart Strategy for Fast Subsumption Check and Coverage Estimation

Ondřej Kuželka and Filip Železný

Intelligent Data Analysis Research Group
Dept. of Cybernetics, Czech Technical University in Prague
`http://ida.felk.cvut.cz`
{`kuzelo1,zelezny`}`@fel.cvut.cz`

**Abstract.** We study the runtime distributions of a simple subsumption check algorithm and show that in some conditions they exhibit heavy tails, indicating a possible runtime advantage achievable by randomizing and restarting the algorithm. Therefore we design ReSumEr, a restarted subsumption tester, incorporating randomization while preserving completeness. On generated graph data, ReSumEr outperforms the state-of-the-art subsumption algorithm Django (i) significantly in the YES region of the phase transition domain and (ii) in the entire phase transition domain given a sufficient size difference between the tested subsumer and subsumee. Importantly, we further show how, under a distributional assumption, a restarted strategy can be used to quickly obtain a maximum likelihood estimate of the coverage of a pattern (proportion of examples subsumed thereby) without requiring to verify subsumption for all examples. We implement this technique in the program ReCovEr and show that it provides accurate coverage estimates in favorable runtimes.

## 1 Introduction

Recent statistical performance studies of search algorithms in difficult combinatorial problems [1, 2] have demonstrated the benefits of randomizing and restarting the search procedure. Specifically, it has been found that if the search cost distribution of the non-restarted randomized search exhibits a slower-than-exponential decay (that is, a "heavy tail"), restarts can reduce the search cost expectation. In [6] we have demonstrated the benefits of randomized restarted strategies in the lattice search conducted by an inductive logic programming system. While the size of pattern spaces represents one source of the complexity of relational data mining, another such source follows from the problem of verifying the subsumption relation between a relational pattern and an example.

This paper first focuses on this latter problem by investigating the possible benefits of a randomized restarted strategy in subsumption testing. Previous research has demonstrated that vast gains in efficiency can be achieved by using unorthodox subsumption algorithms as opposed to standard procedures provided e.g. by a Prolog engine. The pioneering work [4] introduced a tractable approximation to the subsumption test called *stochastic matching*. This randomized algorithm is incomplete in that its failure to prove subsumption in a finite number of steps does not refute the subsumption. On the contrary, we aim at preserving completeness in our randomized restarted procedure called ReSumEr. A complete deterministic approach, called Django, was presented in [3]. Django converts subsumption into a constraint satisfaction problem (CSP) then solved by state-of-the-art heuristic techniques. Django was shown to outperform by orders of magnitude the subsumption testing mechanism used in ILP. Therefore we use Django as the baseline algorithm for comparative experiments with ReSumEr.

Secondly, this paper focuses on the development of an algorithm for fast estimation of pattern coverage, i.e. the proportion of a given set $E$ of examples subsumed thereby. The paper [5] is relevant to this part of our work, in that it estimates total coverage by checking subsumption with respect to a small sample of $E$. We take a different approach in our algorithm, called ReCovEr, enabled by the fact that ReSumEr's runtimes neccessarily follow an exponential distribution. ReCovEr computes pattern coverage through a maximum-likelihood estimation of parameters of this exponential distribution, on the basis of information collected during a (possibly small) sequence of restarts of ReSumEr.

Informally, the main intended contribution of this work is to support efficient mining in large relational structures by enabling fast pattern evaluation. In real-life applications, e.g. in bioinformatics, such structures can typically be represented by oriented graphs. For this reason, we will assume the oriented graph structure of examples and hypotheses, and the subsumption test will coincide with subgraph isomorphism checking.

The rest of the paper is organized as follows. Section 2 defines the syntax of patterns and examples considered and explains how examples are generated for sakes of empirical measurements throughout the paper. In Section 3 we devise a simple subsumption test algorithm, investigate the runtime distributions of both its non-restarted and restarted version (ReSumEr), and empirically evaluate ReSumEr in comparison to Django. Section 4 explains the maximum-likelihood technique for estimating hypothesis coverage implemented in the algorithm ReCovEr, and tests its efficiency and estimation accuracy. Section 5 concludes the paper.

## 2 Preliminaries

In the rest of the paper we assume that patterns and examples are oriented graphs where each vertex may be assigned one of two possible colors. In the dual, relational-logic representation, examples $e$ and patterns $P$ are viewed as conjunctions of positive atoms, each being one of $edge(t_1, t_2)$, $black(t)$, $red(t)$ where $t, t_1, t_2$ are placeholders for terms. All terms in an example $e$ are assumed to be constants and all terms in a pattern $P$ are assumed to be variables. The correspondence between the graph and logic representation is such that vertices correspond to terms and the orientation of and edge is given by the order of term appearance in the corresponding atom. We will refer to the described dual notions interchangeably. When needed, conjunctions will be treated as atom sets, e.g. for two conjunctions $a$ and $b$, $a \subseteq b$ will denote that $b$ contains all atoms contained by $a$.

**Algorithm 1** $SubsumptionCheck(P, e)$: A simple subsumption test algorithm

```
Input: Pattern $P$, example $e$;

if $P \subseteq e$ then
   return YES
else
   Choose variable $V$ from $P$ using a heuristic function (see main text)
   for $\forall S \in PossibleSubstitutions(V, P, e)$ (see main text) do
      $SearchedNodes \leftarrow SearchedNodes + 1$
      Substitute $V$ with $S$
      if $\forall W \in Adjacency(V) : PossibleSubstitutions(W, P, e) \neq \emptyset$ then
         if $SubsumptionCheck(P, e) =$ YES then
            return YES
         end if
      end if
   end for
   return NO
end if
```

**Algorithm 2** $SubstitutionPossible(V, C, P, e)$: Returns NO if $P$ cannot subsume $e$ when $V$ is substituted by $C$.

```
Input: Variable $V$, constant $C$, Pattern $P$, example $e$;

for $\forall A \in P$ such that atom $A$ contains variable $V$ do
   $A' \leftarrow$ replace all occurrences of variable $V$ in atom $A$ by $C$.
   if $A'\theta \nsubseteq E$ (easy to check for a single atom A) then
      return NO
   end if
end for
return YES
```

We consider a simple heuristic algorithm (Algorithm 1) for verifying whether a pattern $P$ subsumes an example $e$. Similarly to Django [3] this algorithm is inspired by the CSP framework and in its terminology it can be described as 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 exists, is detected soon. For a variable $V$, the function returns the sum of occurrences of variables in pattern $P$ that have already been grounded and that share at least one literal with $V$. 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. The function $PossibleSubstitutions(V, P, e)$ returns all constants $C$ for which $SubstitutionPossible(V, C, P, e)$ (Algorithm 2) returns YES.

**Algorithm 3** $RandomGraph(n, p)$: A generator of uniform random graphs

```
Input: Integer n, Real p;

Let V be a set of $n$ vertices and $G$ an empty edge set.
for $\forall \{v_i \in V, v_j \in V | v_i \neq v_j\}$ do
    With probability p, $G \leftarrow G \cup \{v_i, v_j\}$
end for
For all edges in $G$ choose a random orientation, and for all vertices in $V$ choose a random color
with uniform probability from $\{red, black\}$.
return graph with vertex set $V$ and edge set $G$
```

**Algorithm 4** $ScaleFreeGraph(n, k)$: A generator of scale-free random graphs

```
Input: Integers n, k;

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

To obtain a domain-independent runtime distribution of the algorithm, we test it on randomly generated patterns and examples. For generality, we devised two different graph generators for this purpose. The first (Algorithm 3) generates graphs where any two vertices are connected with a pre-set probability $p$ (by an edge of a random orientation). The second (Algorithm 4) produces scale-free ("small world") graphs; here, an edge is attached to a vertex with probability increasing with the number of edges already connected to the vertex. In both algorithms, all vertices are colored as black with probability 0.5 and red otherwise. We will refer to the parameter $p$ ($k$, respectively) of a random uniform (scale-free, respectively) graph as the *connectivity* of the graph.

## 3 ReSumEr: a restarted subsumption tester

We subjected Algorithm 1 to experiments with random sets of patterns and examples generated by Algorithm 3 (Algorithm 4, respectively), under various settings of $n$ and $p$ ($n$ and $k$, respectively). Our objective was to verify the

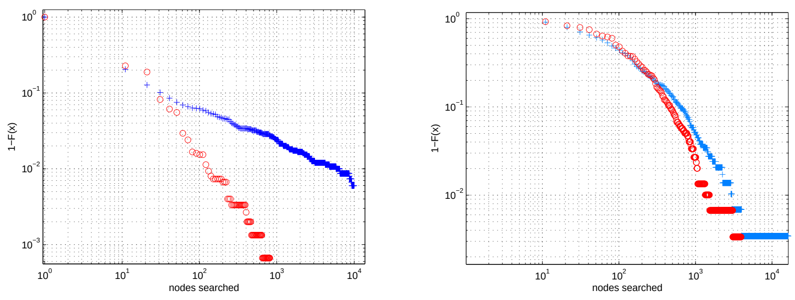

**Fig. 1.** The subsumption test runtime distribution for patterns with $n = 12$ (left) and $n = 14$ (right) vertices and connectivity $p = 0.2$. In both cases, examples had $n = 50$ vertices and connectivity $p = 0.5$. Both patterns and examples were randomly generated by Algorithm 3. A heavy tail is observed in the left panel for the non-restarted version (blue) and significant speed-up is achieved by the restarted version (red). Although no heavy tail is observed in the right panel for the non-restarted version (blue), small speed-up is still observed for the restarted version (red).

presence of *heavy tails* in the runtime distributions $F(t)$. For a $t > 0$, $F(t)$ is the probability that the tested algorithm resolves a random subsumption instance in no more than $t$ units of time, corresponding to the number of explored search nodes. A heavy tail is exhibited if $1 - F(t)$ decays at a power-law rate, i.e. slower than exponentially. Informally, a heavy-tailed distribution indicates the non-negligible probability of subsumption instances on which the checking algorithm gets stuck for an extremly long runtime. The presence of a heavy tail in an empirical runtime distribution $F(t)$ can be checked graphically, by plotting $1 - F(t)$ against $t$ on a log-log scale. For a growing $t$, a heavy-tailed distribution here acquires a linear shape [2].

Our findings were not conclusive in that for various configurations mentioned above, some runtime distributions were heavy-tailed while others were not. Two examples are shown in Fig. 1 in blue; while differing very slightly in a single parameter ($n$) value, the respective distributions posses largely different shapes of the tails. We have not yet been able to establish a principled correspondence between the respective parameter values and the occurrence of heavy tails.

While the presence of heavy tails for some classes of subsumption instances indicates possible large runtime benefits achievable by a restarting strategy [2], its effect on the non-heavy-tailed classes may not be necessarily detrimental. We thus decided to assess the overall impact of restarting empirically. For this sake we designed a complete restarted randomized subsumption algorithm ReSumEr (Algorithm 5). Its completeness is guaranteed by the assumption that for the cutoff sequence $R(n)$, $R(n) \to \infty$ as $n \to \infty$. Note that the randomization is facilitated by tie-breaking in the heuristic function used in the embedded Algorithm 1.

**Algorithm 5** *ReSumEr*$(P, e, R)$: A restarted subsumption algorithm

```
Input: Pattern $P$, example $e$, cutoff sequence $R$;
$n \leftarrow 1$
repeat
    Answer $\leftarrow$ Run $SubsumptionCheck(P, e)$ with number of searched nodes limited to $R(n)$
    $n \leftarrow n + 1$
until Answer YES or NO is returned
return Answer
```

The runtime distributions for ReSumEr, with an ad-hoc chosen restart sequence $R(n) = 10n^2 + 30$ are plotted in red in Fig. 1 for the earlier exemplified cases of both heavy-tailed and non-heavy-tailed behavior. In both cases, restarts generally reduce runtime, although the difference is much more significant in the heavy-tailed case. The set of random subsumption instances naturally comprise of both satisfiable (where $P$ subsumes $e$) and non-satisfiable instances. Of relevance, the times taken by ReSumEr on the non-satisfiable instances were in this experiment on average about $10^3$ times higher than on the satisfiable ones. This is natural due to the 'iterative' character of ReSumEr; while satisfiability can in principle be shown in any single restart, non-satisfiability can only be shown after $n$ restarts making $R(n)$ sufficiently high.

We next aimed to compare ReSumEr to a baseline algorithm used for subsumption in relational data mining. As explained earlier, the graph structures we here deal with are easily embedded into conjunctions of first-order positive atoms. Thus an obvious baseline algorithm candidate would have been the unification mechanism in Prolog. However the sizes of patterns and examples (tens of vertices in patterns, hundreds in examples) we focus on, consistently result in unmeasurably large runtimes of this procedure. A much faster alternative, which we adopt for comparisons, is represented by the state-of-the-art subsumption algorithm Django [3].

All experiments were conducted on the same computer. Django is implemented in C and we used its version 11. ReSumEr is implemented in JAVA. Figures 2 and 3 display the results for patterns and examples generated as uniform random graphs (Fig. 2) and scale-free graphs (Fig. 3). The comparative runtimes (top panels) are accompanied by the corresponding phase transition diagrams (bottom panels). The left (right, respectively) panels pertain to a smaller (larger, respectively) size difference between the patterns and the examples. Size is understood as the number of contained vertices.

We now note on the principled trends apparent from the results. First, ReSumEr consistently and significantly outperformed Django in the YES region of the phase transition spectrum.[1] Second, in the experiments with a larger size-difference between the patterns and the examples, ReSumEr was faster across the entire phase transition domain. Third, heavy-tailed behavior of Django was

[1] which corresponds to the left parts of all diagrams in Figures 2 and 3. Although the observed absolute difference is larger in the NO (right-hand side) region, in relative terms it is much smaller than the difference in the YES region.

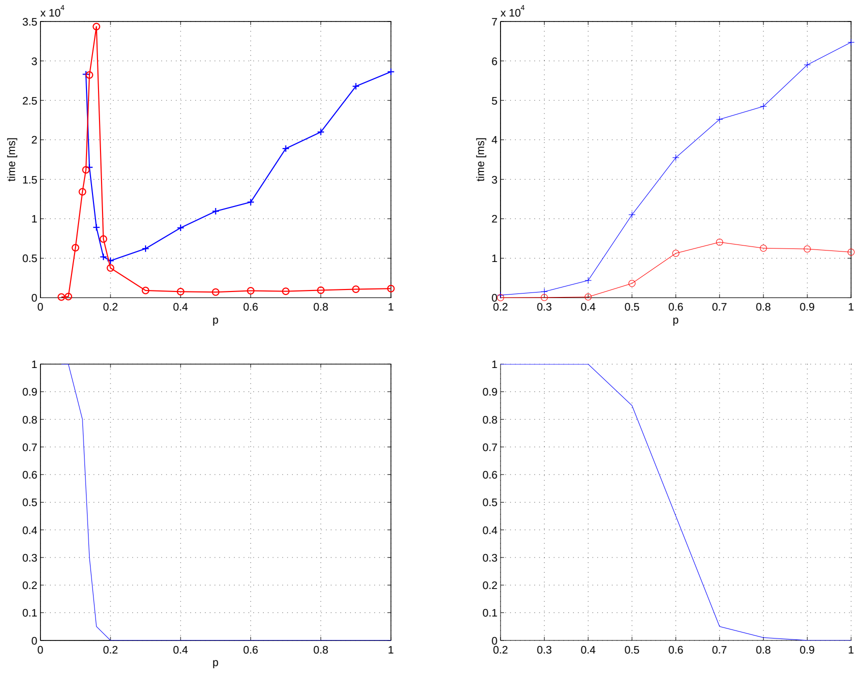

**Fig. 2. Top:** Comparisons of Django (blue) and ReSumEr (red) runtimes of subsumption checks between patterns and examples generated by Algorithm 3 with connectivity $p = 0.3$ for examples and varying $p$ (horizontal axis) for patterns. In the left panel, patterns have 30 vertices and examples have 100 vertices. In the right panel, patterns have 10 vertices and examples have 200 vertices. All shown points are averages of 50 measurements. **Bottom:** The phase transition landscapes for the respective settings above: the probability that a random pattern with connectivity $p$ (horizontal axis) subsumes a random example with connectivity $p = 0.3$.

observed: in spite of its typical measured runtimes in the order of milliseconds to seconds, occasional runs in satisfiable instances took up to tens of minutes and had to be curtailed. This resulted in Django's excessive runtimes in the top-left panel of Fig. 2 (Fig. 3, respectively) for $p \leq 0.1$ ($k = 3$, respectively). Heavy-tailed behavior is prevented by ReSumEr resulting in its vast superiority in the $p \leq 0.1$ region of Fig. 2, top-left panel. In Fig. 3, however, Resumer's averaged runtimes were also excessive for $k = 3$ and $k = 4$. Unlike for Django, here the reason was not in occasional excessive runs, but rather in the systematic increase of runtime required to complete the unsatisfiable subsumption instances. Fourth, the generally high runtimes of Django in the NO region are surprising. In particular, for large size-differences between the patterns and the examples (right panels in Fig. 2 and 3), Django's runtimes in the NO region were even

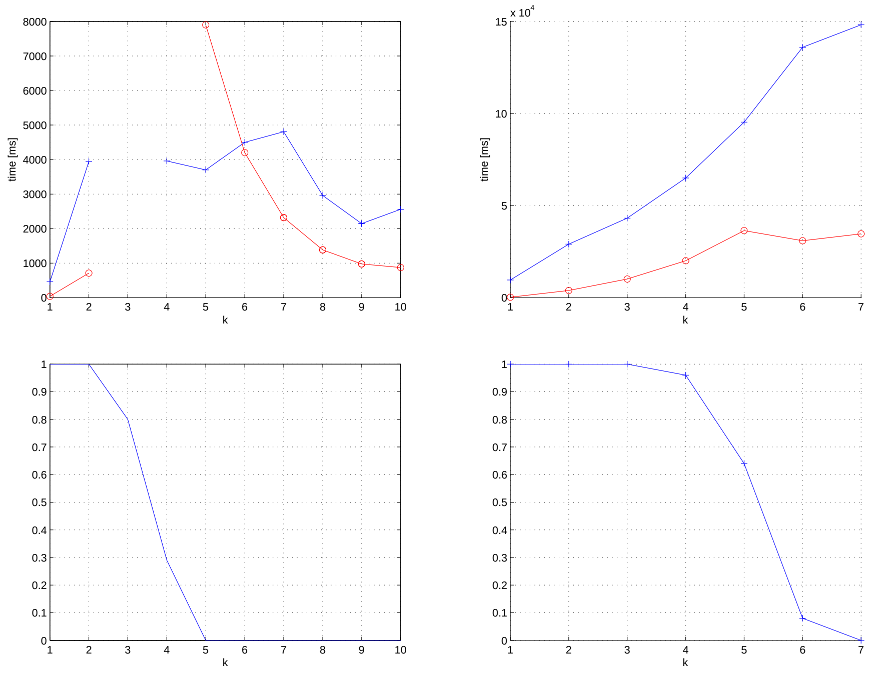

**Fig. 3. Top:** Comparisons of Django (blue) and ReSumEr (red) runtimes of subsumption checks between patterns and examples generated by Algorithm 4 with connectivity $k = 20$ for examples and varying $k$ (horizontal axis) for patterns. In the left panel, patterns have 30 vertices and examples have 100 vertices; for some $k$, runtimes were not measurable (see main text). In the right panel, patterns have 20 vertices and examples have 500 vertices. All shown points are averages of 50 measurements. **Bottom:** The phase transition landscapes for the respective settings above: the probability that a random pattern with connectivity $k$ (horizontal axis) subsumes a random example with $k = 20$.

consistently higher than those in the YES/NO (transition) region.[2] Although this phenomenon was also reported in [3] (Table 4 therein) for Django version 1, in general the runtimes reported by [3] for the NO region are much smaller than those in the transition region. Further investigation is thus needed to clarify this discrepancy in light of the differences between our experimental setting and that in [3].

[2] Thus eliminating the usual runtime spikes in the transition area. For ReSumEr, such spikes are also small in the right panels of Fig. 2 and 3, however, here the reason clearly lies in ReSumEr's laborious 'iterative' approach for proving unsatisfiable subsumption instances, as commented earlier.

## 4 ReCovEr: a restart-based coverage estimator

We will now explain how to exploit a restarted strategy to obtain a maximum likelihood estimate of the proportion of examples subsumed by a pattern, without the need to complete a subsumption with a definitive (YES/NO) answer for any particular example.

We first need to make the assumption that given a pattern $P$ and a set of examples $E$, the probability that Algorithm 1 finds a solution (i.e. returns YES as its answer) before it explores more than *cutoff* nodes of the search tree, is same for all $e \in E$ such that $P$ subsumes $e$. Denote this probability by $p$. We will not directly assess the empirical accuracy of this assumption, but we will later verify it indirectly by testing the accuracy of the algorithm built upon it.

We assume a given pattern $P$ and we fix a constant cutoff value $R$. In the first step, for each $e \in E$ we run $\mathit{SubsumptionCheck}(P, e)$ (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

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

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

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

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

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

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

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

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

$$\ln(P(m_1, \ldots, m_k)) =$$

$$= \sum_{i=1}^{k} \ln \binom{A - \sum_{j=1}^{i-1} m_j}{m_i} + \sum_{i=1}^{k} m_i \ln p + \sum_{i=1}^{k} \left(A - \sum_{j=1}^{i} m_j\right) \ln(1-p) \tag{5}$$

**Algorithm 6** *ReCovEr*$(P, E, R, M, \Delta)$: Algorithm for coverage estimation

```
Input: Pattern $P$ and set of examples $E$, Integers $R$ (‘cutoff’), $M$, $\Delta$;

$tries \leftarrow 0$
$Unknown \leftarrow Examples$
$CoveredInIthTry \leftarrow []$
repeat
    $tries \leftarrow tries + 1$
    $CoveredInThisTry \leftarrow 0$
    for $\forall E \in Unknown$ do
        $Answer \leftarrow$ Run $SubsumptionTest(P, E)$ with number of searched nodes limited to $R$
        if $Answer = PositiveMatching$ then
            $CoveredInThisTry \leftarrow CoveredInThisTry + 1$
            $Unknown \leftarrow Unknown \setminus E$
        end if
    end for
    $CoveredInIthTry[tries] \leftarrow CoveredInThisTry$
until $tries \geq M \wedge \|LikelihoodEstimate(tries - 1) - LikelihoodEstimate(tries)\| \leq \Delta$
return $LikelihoodEstimate(tries)$
```

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

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

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

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

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

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

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 6). The question of 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 two subsequent estimates differ by less than some $\Delta$, specified as a parameter. A minimum length $M$ of the sequence is however imposed, to avoid premature estimates coinciding by chance.

For purposes of initial empirical assessment of ReCovEr, we generated patterns and examples by Algorithm 3 with $p = 0.4$ for patterns and $p = 0.3$ for examples. Figure 4 demonstrates that ReCovEr produces estimates of acceptable accuracy. Certain imprecision indeed can be accepted: the coverage estimate should be accurate to the extent allowing to establish a reliable ranking of candidate hypotheses. Apart from that, the actual coverage value is seldom of interest.

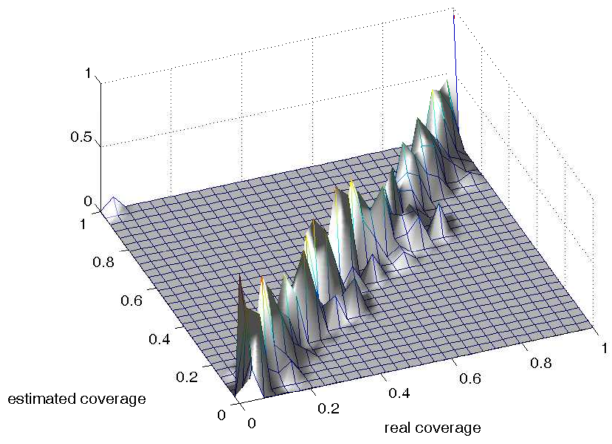

**Fig. 4.** Precision of ReCovEr (Algorithm 6) estimates reflected as the joint distribution of 1000 pairs [estimated, real], for $R = 100$, $M = 6$ and $\Delta = 1$. Patterns and examples were generated by Algorithm 3 with $p = 0.4$ for patterns and $p = 0.3$ for examples. Patterns have 10 vertices, examples have 100 vertices. The 1000 estimates correspond to 1000 different hypotheses tested on a pre-fixed set of 100 examples.

| **Algorithm** | **Avg. Time [s]** |
|---|---|
| ReCovEr | 20.2 |
| ReSumEr | 41.7 |
| Django | 45.9 |

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

Table 1 shows the average runtime of ReCovEr testing one pattern on 100 examples in the same experimental configuration as in Fig. 4. For comparison, the table also shows the analogous runtimes needed to compute the coverage by testing subsumption for each $e \in E$ by ReSumEr or Django. While the runtime benefit provided by ReCovEr does not appear particularly significant from this table, it must be noted that the experimental parameters (detailed in the caption of Fig. 4) chosen for this first assessment correspond to areas where both ReSumEr and Django perform well. It is our expectation that ReCovEr will outperform much more significantly both Django in the YES domain and ReSumEr in the NO domain. In the former case, the expectation is based on the fact that in the YES region, Django exhibits heavy tails which are, by construction, prevented by ReCovEr. In the latter case, ReSumEr conducts an expensive iterative approach for showing unsatisfiability of subsumption, which becomes a crucial factor in the NO region. On the contrary, ReCovEr prevents this burden because it does not need to complete the subsumption check for any unsatisfiable subsumption instance.

As a last remark, we did not compare ReCovEr to the sampling method from [5]. Although [5] alleviates coverage computation by taking only a small sample of $E$, subsumption for any single $e$ from that sample is tested in the standard Prolog framework. Such an approach exceeds measurable runtimes for the sizes of $e$ considered in this paper.

## 5 Conclusions and Future Work

We have demonstrated the benefits of using a complete restarted randomized algorithm ReSumEr for subsumption testing, a procedure at heart of most relational data mining systems. We have further introduced ReCovEr, an algorithm exploiting restarts for a maximum-likelihood based estimation of pattern coverage, eliminating the needed to prove subsumption for any particular example; the set of examples for which subsumption is actually proved during the application of ReCovEr is a result of a random process. ReCovEr prevents heavy tails as well as laborious proving of unsatisfiable subsumption instances. All of our experimental evaluations were constrained to generated data in the form of oriented colored graphs (uniform and scale-free, respectively). However, the principles behind ReSumEr and ReCovEr do not rely on this representation bias. Our next work will therefore concentrate on tests with a more general structure representation language and with real-life relational data mining benchmarks.

**Acknowledgements**
The first author is supported by the Czech Academy of Sciences through the project 1ET101210513 Relational Machine Learning for Biomedical Data Analysis. The second author is supported by the European Commission through the project FP6-027473 SEVENPRO.

## References

1. H. Chen, C. Gomes, and B. Selman. Formal models of heavy-tailed behavior in combinatorial search. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 408–421. Springer-Verlag, 2001.
2. C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.
3. J. Maloberti and M. Sebag. Fast theta-subsumption with constraint satisfaction algorithms. *Machine Learning*, 55(2):137–174, 2004.
4. M. Sebag and C. Rouveirol. Tractable induction and classification in first-order logic via stochastic matching. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 888–893. Morgan Kaufmann, 1997.
5. A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.
6. F. Zelezny, A. Srinivasan, and D. Page. Randomised restarted search in ILP. *Machine Learning*, 64(1–2):183–208, 2006.