Fundamenta Informaticae XXI (2001) 1001–1015 1001
IOS Press

A Restarted Strategy for Efficient Subsumption Testing

Ondřej Kuželka *
kuzelo1@fel.cvut.cz

Filip Železný †
zelezny@fel.cvut.cz
Intelligent Data Analysis Research Group
Department of Cybernetics
Czech Technical University in Prague
Prague, Czech Republic

Abstract. We study runtime distributions of subsumption testing. On graph data randomly sampled from two different generative models we observe a gradual growth of the tails of the distributions as a function of the problem instance location in the phase transition space. To avoid the heavy tails, we design a randomized restarted subsumption testing algorithm RESUMER2. The algorithm is complete in that it correctly decides both subsumption and non-subsumption in finite time. A basic restarted strategy is augmented by allowing certain communication between odd and even restarts without losing the exponential runtime distribution decay guarantee resulting from mutual independence of restart pairs. We empirically test RESUMER2 against the state-of-the-art subsumption algorithm Django on generated graph data as well as on the predictive toxicology challenge (PTC) data set. RESUMER2 performs comparably with Django for relatively small examples (tens to hundreds of literals), while for further growing example sizes, RESUMER2 becomes vastly superior.

Keywords: Relational learning, Graph Mining, Subsumption, Homomorphism, Randomized Complete Algorithm

Address for correspondence: Czech Technical University in Prague, Technická 2, 166 27 Prague 6, Czech Republic
*Ondřej Kuželka is supported by the Grant Agency of the Czech Republic through the project 201/08/0486 Merging Machine Learning with Constraint Satisfaction
†Filip Železný is supported by EU FP6 project 027473-SEVENPRO and project 1ET101210513 (Relational Machine Learning for Analysis of Biomedical Data) of the Czech Academy of Sciences

1. Introduction

Recent statistical performance studies of search algorithms in difficult combinatorial problems [3, 6] 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 [14] 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 [10] 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 [9]. 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.

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 homomorphism checking.

The rest of the paper is organized as follows. Section 2 constrains the syntax of patterns and examples considered in this study and introduces a simple baseline complete subsumption test algorithm. Section 3 subjects the baseline algorithm to empirical runtime distribution measurements on sets of patterns and examples randomly generated from two graph families: uniformly random graphs and scale-free graphs. This section indicates possible benefits achievable through restarts. Subsequently, Section 4 presents a randomized restarted modification, called RESUMER[1] of the baseline algorithm which is complete and avoids heavy tails of the baseline algorithm's runtime distributions. The algorithm is further augmented into RESUMER2 by allowing transfer of information among individual restarts. In Section 5 we empirically evaluate RESUMER2 in comparison to the state-of-the-art subsumption algorithm Django. Here we first continue to use the generated graph data and then validate the results on a real-life organic chemistry data set (predictive toxicology challenge). Section 6 concludes the paper.

2. Preliminaries

Until Section 5.2 we shall assume that patterns and examples are oriented graphs where each vertex may be assigned one of two possible colors unless stated otherwise. The colors are to be understood

[1] Restarted Subsumption Tester

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

```
1: Input: Pattern P, example e;
2: if P ⊆ e then
3:    return YES
4: else
5:    Choose variable V from P using a heuristic function /* see main text */
6:    for ∀S ∈ PossibleSubstitutions(V, P, e) /* see main text */ do
7:       SearchedNodes ← SearchedNodes + 1
8:       /* SearchedNodes is a global variable initiated to 0 prior to executing this algorithm. */
9:       P' ← Substitute V with S
10:      if ∀W ∈ Adjacency(V) : PossibleSubstitutions(W, P', e) ≠ ∅ then
11:         if SubsumptionCheck(P', e) = YES then
12:            return YES
13:         end if
14:      end if
15:   end for
16:   return NO
17: end if
```

as an abstraction of arbitrary properties assigned to vertices. 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 an 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. Following this notation we can define θ-subsumption as follows.

Definition 2.1. We say that pattern P θ-subsumes example e if and only if there is a substitution θ such that $P\theta \subseteq e$.

Example 2.1. Let us have the following pattern P and example e

$$P = red(X), edge(X, Y), edge(Y, Z)$$

$$e = black(a), red(b), black(c), edge(a, b), edge(b, c), edge(c, a)$$

Then P θ-subsumes e, because for $\theta = \{X/b, Y/c, Z/a\}$ it is true that $P\theta \subseteq e$.

We consider a simple heuristic algorithm (Algorithm 1) for verifying whether a pattern P θ-subsumes an example e. Similarly to Django [9] 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

Algorithm 2 $SubstitutionPossible(V, C, P, e)$: Returns NO if P cannot subsume e when V is grounded to C. (The reverse implication may not hold, see main text.)

```
1: Input: Variable V, constant C, Pattern P, example e;
2: for ∀A ∈ P such that atom A contains variable V do
3:    A' ← replace all occurrences of variable V in atom A by C.
4:    if ∀θ.A'θ ⊈ e /* easy to check for a single atom A */ then
5:       return NO
6:    end if
7: end for
8: return YES
```

one exists, is detected soon. For a variable V, the function computes the sum of occurrences of variables in pattern P 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 $PossibleSubstitutions(V, P, e)$ returns all constants C (in random order) for which $SubstitutionPossible(V, C, P, e)$ (Algorithm 2) returns YES. The function prunes away a subset of possible groundings for V whose inclusion in θ would imply $P\theta \not\subseteq e$. In general though, not all such groundings are detected by the function.

3. Subsumption Test Runtime Distributions

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 Erdos-Rényi random 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 parameter p (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 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 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. 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 that standard distributions which decay exponentially. The presence of a heavy tail in an empirical 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 [6].

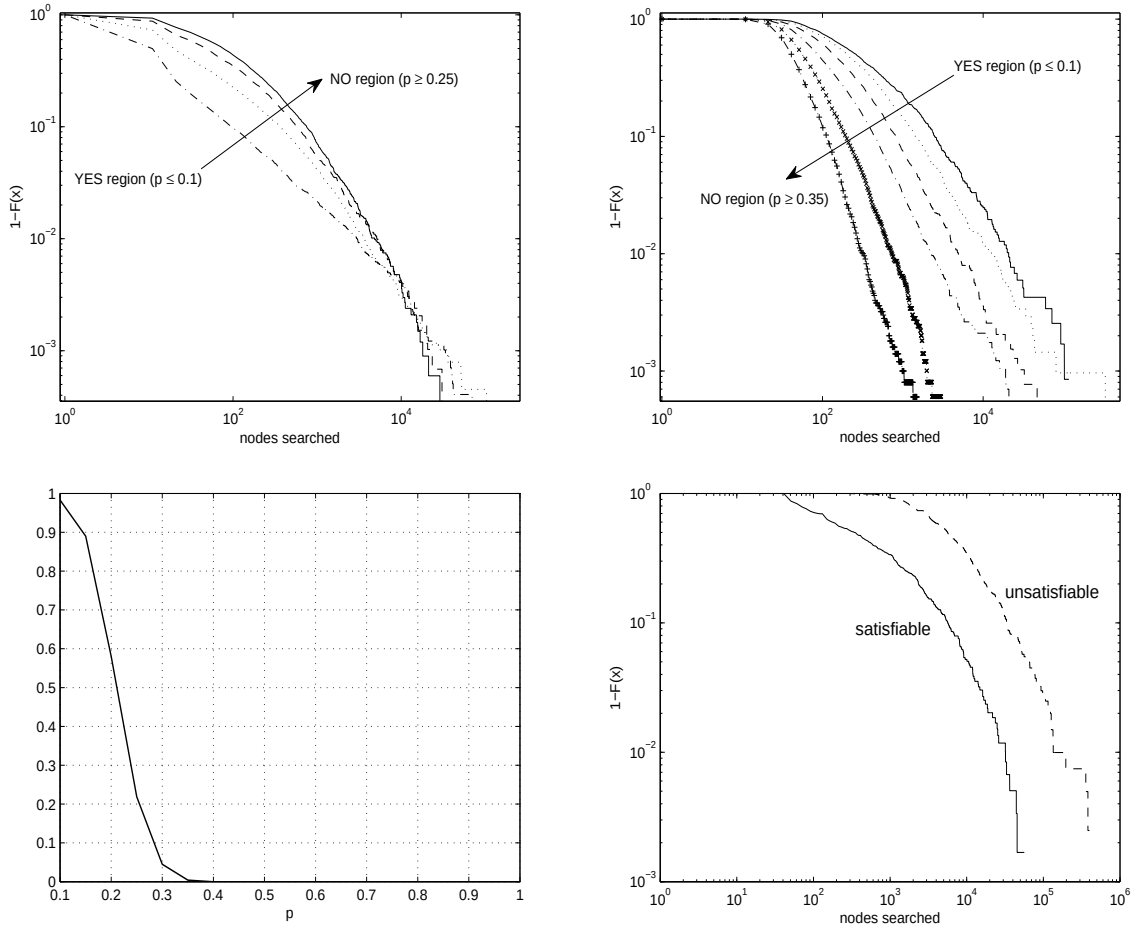

Figure 1. **Top left:** The subsumption test runtime distributions for satisfiable instances with patterns with $n = 15$ vertices and connectivity consecutively $p = 0.1$, $p = 0.15$, $p = 0.2$, $p = 0.25$. The examples had $n = 50$ vertices and connectivity $p = 0.3$. **Top right:** The subsumption test runtime distributions for unsatisfiable instances with patterns with $n = 15$ vertices and connectivity consecutively $p = 0.15$, $p = 0.2$, $p = 0.25$, $p = 0.3$, $p = 0.35$. The examples had $n = 50$ vertices and connectivity $p = 0.3$. **Bottom left:** The phase transition spectrum corresponding to the runtime distributions displayed in the top panels. **Bottom right:** Runtime distributions for basic restarted algorithm on satisfiable and unsatisfiable instances.

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

Our findings were initially not conclusive in that for various configurations, some runtime distributions were clearly governed by a power-law while others were not. We have therefore performed a series of experiments in the phase transition framework, which was imported to relational learning by Giordana et. Saitta [4]. Phase transition framework studies the correspondence between parameters of patterns and probability p_{subs} that a randomly selected pattern P with these parameters θ-subsumes an example e. A phase transition spectrum is divided into three disjoint regions: the YES region, the NO region and the PT (phase transition) region. For patterns correponding to the YES region the probability p_{subs} is close to 1, as patterns are mostly underconstrained in this region, while it is nearly 0 for patterns corresponding to the NO region, as these are mostly overconstrained. The PT region then corresponds to the region where p_{subs} usually quickly drops from being close to 1 to being almost 0 and subsumption checking is usually computationally most expensive for patterns in this region. The computational difficulty in the transition region is due to patterns being neither overconstrained nor underconstrained. In this case, subsumption typically cannot be quickly refuted by discovering an inconsistency early, or proved. For an extensive general introduction to phase transitions in computational complexity, see the seminal paper [8].

Our experiments in phase transition framework revealed a systematic progression from heavy-tailed regimes corresponding to configurations located in the YES region (low connectivity in patterns) of the phase transition spectrum to non-heavy-tailed regimes corresponding to configurations located in the

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

Algorithm 6 $ReSumEr2(P, e, R)$: A modified restarted subsumption algorithm

Input: Pattern P, example e, cutoff sequence R;
$n \leftarrow 1$
repeat
  if n is odd **then**
    $Answer \leftarrow$ Run $SubsumptionCheck(P, e)$ with number of searched nodes limited to $R(n)$ and record $LastVariable \leftarrow$ the last variable that caused backtracking there in.
  else
    $Answer \leftarrow$ Run $SubsumptionCheck(P, e)$ with number of searched nodes limited to $R(n)$ and the first checked variable set to $LastVariable$
  end if
  $n \leftarrow n + 1$
until $Answer$ YES or NO is returned
return Answer

NO region (high connectivity in patterns). This observation agrees with the previous study [3]. This progression is shown in Fig. 1 for the Erdos-Renyi graph data (Algorithm 3). The same trends were observed for the small-world graph data. The runtime distributions plotted refer to subsumption checks between examples with fixed numbers of vertices and with fixed connectivity and patterns with fixed numbers of vertices and connectivity changing among particular distributions. The runtime distributions plotted in the top left panel of Fig. 1 refer to satisfiable problem instances, i.e. those where the patterns θ-subsume the examples. The distributions in the top right panel of Fig. 1 refer to unsatisfiable problem instances.

The runtime distributions for both satisfiable and unsatisfiable instances have heaviest tails when the problem instances are located in the YES region and they decay faster as the problem instances get closer to the NO region. A difference between runtime distributions for satisfiable and unsatisfiable problem instances is observed in the probability of very short runs, which is high for the satisfiable instances located in the YES region, but which is negligible for the unsatisfiable instances in that region. As a consequence, individual distributions cross each other in the top left panel of Fig. 1 whereas this effect is not observed in the top right panel of Fig. 1.

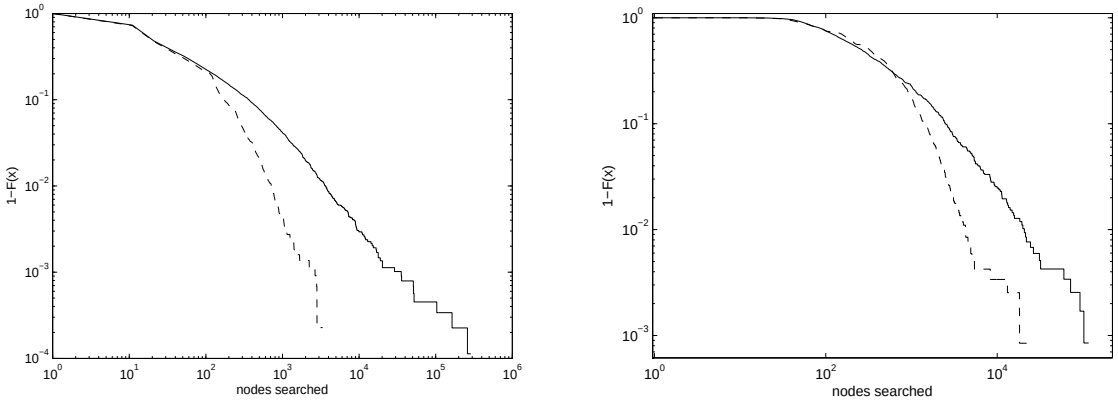

Figure 2. Effect of the basic restarted strategy for satisfiable (left panel) and unsatisfiable instances (right panel). The patterns have n = 15 vertices and connectivity p = 0.15. In both cases, examples had n = 50 vertices and connectivity p = 0.3. Both patterns and examples were randomly generated by Algorithm 3. A speed-up is observed for the basic restarted strategy. It is more significant for the satisfiable instances (left panel).

4. Designing a Restarted Subsumption Test Algorithm

While the presence of heavy tails for some classes of subsumption instances indicates possible large runtime benefits achievable by a restarting strategy [6], 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 $R(n)$, $R(n) \rightarrow \infty$ as $n \rightarrow \infty$, because then there is always such n_0, for which the cutoff $R(n_0)$ enables Algorithm 5 to explore the whole search tree. Note that randomization is facilitated by tie-breaking in the heuristic function used to select variables, which should be grounded, in line 5 of Algorithm 1 and by randomization of the ordering of possible groundings.

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

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

are plotted in Fig. 2 for two of the cases exemplified in Fig. 1. In both of these cases, restarts reduce runtime, although the difference is much more significant in the satisfiable case (left panel). The set of random subsumption instances naturally comprise of both satisfiable (where P subsumes e) and unsatisfiable instances. Of relevance, the times taken by RESUMER on the unsatisfiable instances were in some of our experiments about 10^2 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, unsatisfiability can only be shown after n restarts making $R(n)$ sufficiently high.

It is further possible to increase the performance of RESUMER by a heuristic modification of the basic restarted strategy (Algorithm 5), which repeatedly calls Algorithm 1. The basic idea is to allow a certain transfer of knowledge between individual restarts by guiding the initial selection of a variable variable in Algorithm 1 (line 4). In particular, when this algorithm is called from RESUMER, we choose

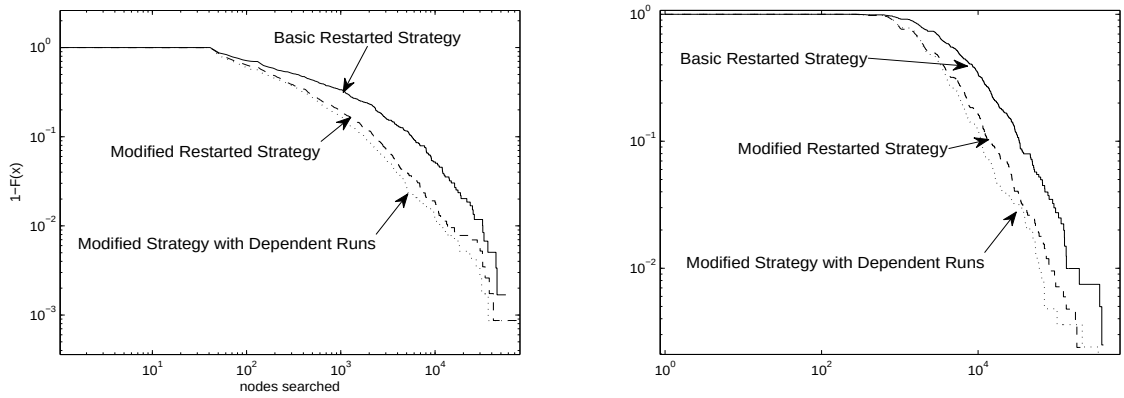

Figure 3. Effect of the basic restarted strategy (RESUMER), modified restarted strategy (RESUMER2) and modified restarted strategy with dependent runs for satisfiable (left panel) and unsatisfiable instances (right panel). The patterns have n = 20 vertices and connectivity p = 0.2. In both cases, examples had n = 100 vertices and connectivity p = 0.3. Both patterns and examples were randomly generated by Algorithm 3.

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

This straightforward modification would have the consequence that pairs of restarts would no longer be statistically independent trials. In general, this might represent a problem. A restarted strategy exhibits the desirable property of an exponentially decaying runtime only if individual restarts are independent. For this reason we only allow the above described transfer of knowledge from odd restarts into the subsequent even restarts, resulting in a series of restart pairs, which are mutually independent. Thus we maintain the exponential decay guarantee. We denote this version of the algorithm as RESUMER2.

The effect of using this strategy is shown in Fig. 3. For completeness, the runtime distributions for the modification of RESUMER, where every single run of the algorithm starts with the last variable that caused backtracking, is also plotted in Fig. 3. Clearly, the additional gain of the latter strategy over RESUMER2 is marginal, and not worth sacrificing the exponential decay guarantee.

5. Experimental Evaluation

Finally we subjected RESUMER2 to a comparative empirical evaluation with 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 [9].

All experiments were conducted on the same computer. Both Django and RESUMER2 are implemented in C. We used Django version 11.

5.1. Generated Data

Figures 4 and 5 display the results for patterns and examples generated as uniform random graphs (Fig. 4) and scale-free graphs (Fig. 5). 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, RESUMER2 consistently and significantly outperformed Django in the YES region of the phase transition spectrum. This region corresponds to the left parts of all diagrams in Figures 4 and 5. 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.

Second, in the experiments with a larger size-difference between the patterns and the examples (examples much larger), RESUMER2 was faster across the entire phase transition domain.

Third, heavy-tailed behavior of Django was observed: in spite of its typical measured runtimes in the order of milliseconds to seconds, occasional runs in satisfiable instances took up to tens of minutes and had to be curtailed. This resulted in Django's excessive runtimes in the top-left panel of Fig. 4 (Fig. 5, respectively) for $p \leq 0.1$ ($k = 3$, respectively). Heavy-tailed behavior is prevented by RESUMER2 resulting in its vast superiority in the $p \leq 0.1$ region of Fig. 4, top-left panel. In Fig. 5, however, RESUMER2's averaged runtimes were also excessive for $k = 3$. 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. 4 and 5), Django's runtimes in the NO region were even consistently higher than those in the YES/NO (transition) region.[2] Although this phenomenon was also reported in [9] (Table 4 therein) for Django version 1, in general the runtimes reported by [9] 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 [9].

5.2. Predictive Toxicology Challenge Data

Next, we studied performance of RESUMER2 on a real-life dataset from the Predictive Toxicology Challenge (PTC) [7]. The PTC dataset consists of 344 organic molecules marked according to their carcinogenicity on male and female mice and rats. Our relational-logic representation of these molecules consisted of ternary literals for atomic bonds $bond(at1, at2, bondName)$, unary literals representing types of particular bonds $singleBond(bondName)$, $doubleBond(bondName)$, $tripleBond(bondName)$ and $resonantBond(bondName)$ and unary literals for atom types $c(atom)$, $h(atom)$, $n(atom)$ etc.

[2] Thus eliminating the usual runtime spikes in the transition area. For RESUMER2, such spikes are also small in the right panels of Fig. 4 and 5, however, here the reason clearly lies in RESUMER2's laborious 'iterative' approach for proving unsatisfiable subsumption instances, as commented earlier.

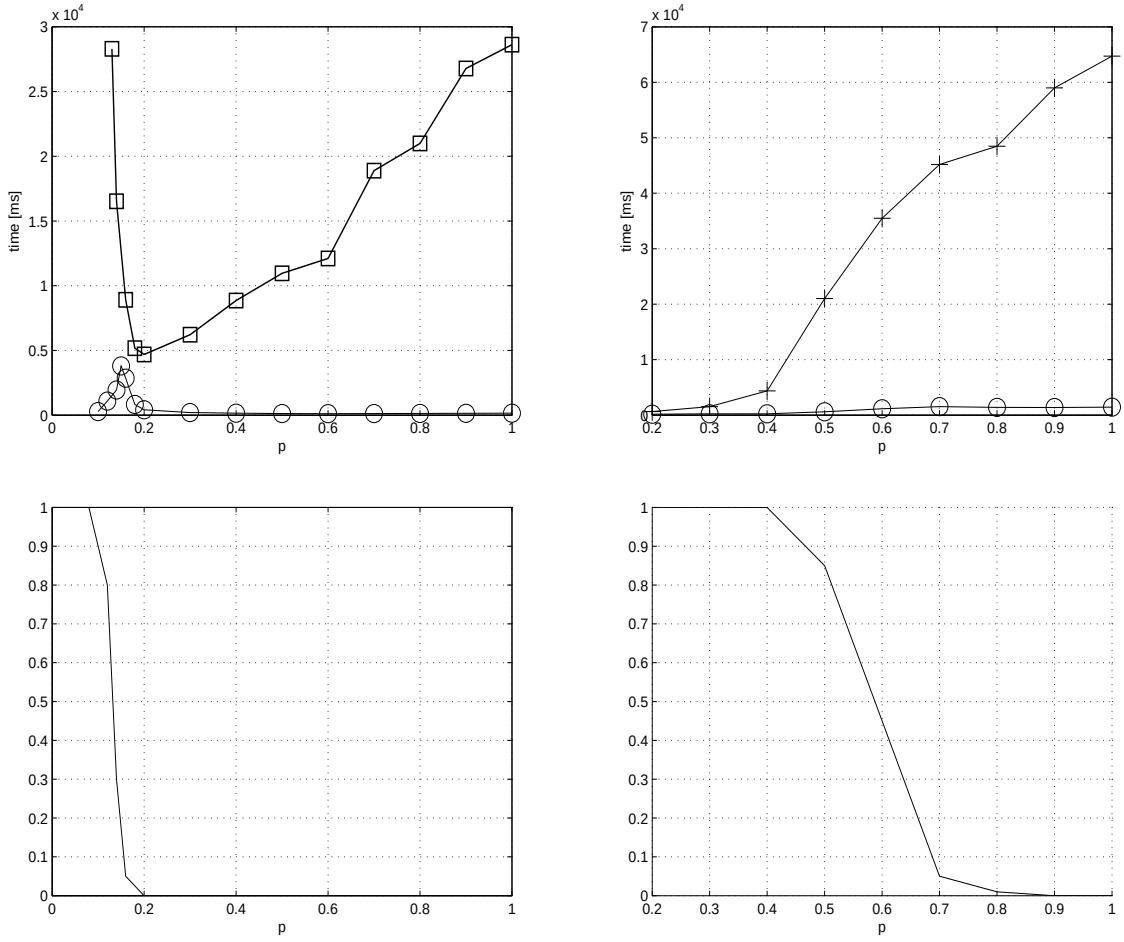

Figure 4. **Top:** Comparisons of Django (squares) and RESUMER (circles) runtimes of subsumption checks between patterns and examples generated by Algorithm 3 with connectivity $p = 0.3$ for examples and varying p (horizontal axis) for patterns. In the left panel, patterns have 30 vertices and examples 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 $p = 0.3$.

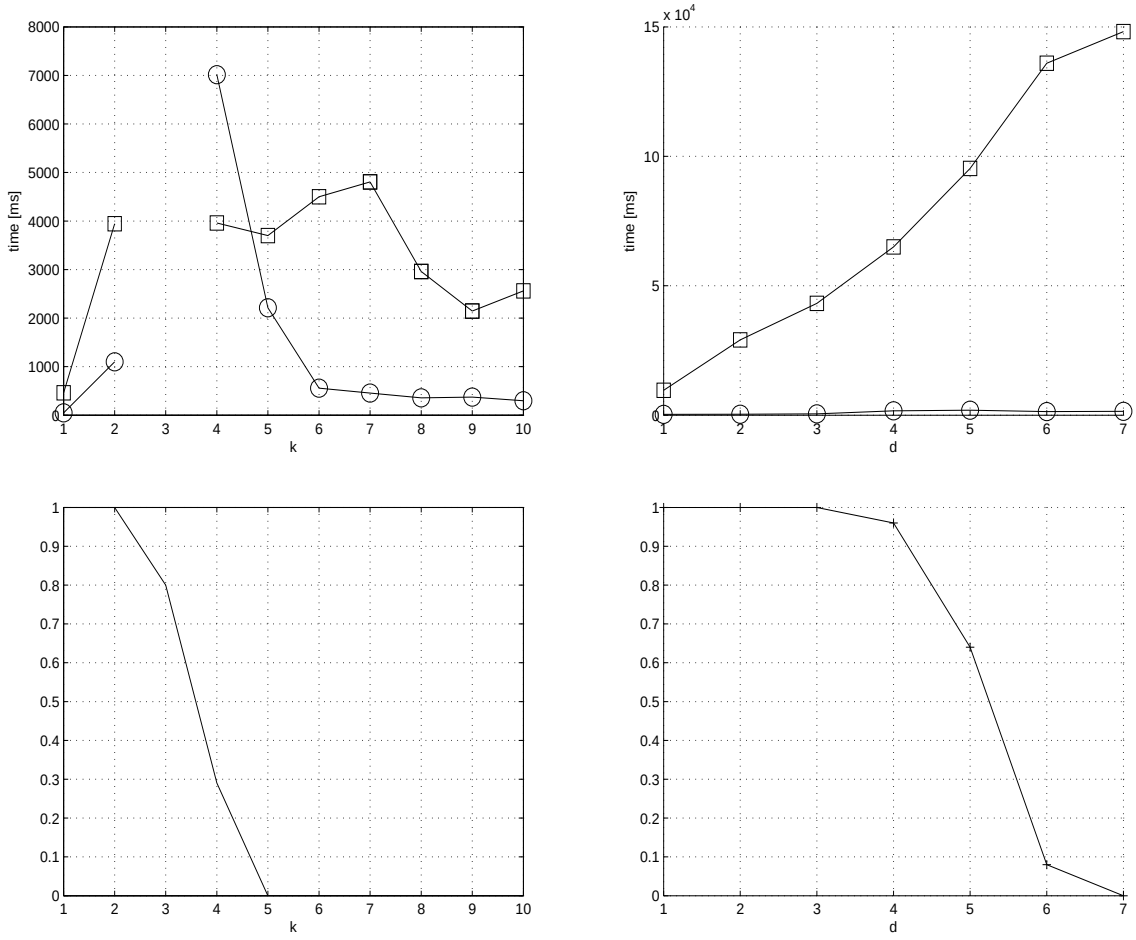

Figure 5. **Top:** Comparisons of Django (squares) and RESUMER (circles) runtimes of subsumption checks between patterns and examples generated by Algorithm 4 with connectivity $k = 20$ for examples and varying k (horizontal axis) for patterns. In the left panel, patterns have 30 vertices and examples 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 $k = 20$.

PTC-v1	YES	PT	NO
RESUMER2	0.03	0.09	0.13
Django	0.11	0.38	0.24

PTC-v2	YES	PT	NO
RESUMER2	2	22	70
Django	570	2511	1408

PTC-v3	YES	PT	NO
RESUMER2	0.1	1.0	0.7
Django	6.4	38.1	6.8

Table 1. Average runtimes in milliseconds for subsumption checks between patterns and examples from the PTC dataset. YES region corresponds to patterns that covered more than 80% examples, phase transition region corresponds to patterns that covered more than 20% and less than 80% examples and NO region corresponds to patterns that covered less than 20% examples.

In this real data domain, we decided not to generate patterns entirely randomly. Our intention was to simulate general principles of pattern production in a typical general-to-specific inductive logic programming system, while avoiding an overfit to a specific pattern search strategy (which would e.g. be a result of adhering to a specific heuristic function for selecting literals).

So motivated, we decided to generate patterns by random walks through the subsumption lattice. The random walks start with the most general pattern ('$true$') and traverse the subsumption lattice in a top-down manner by consecutively adding randomly selected literals, which preserved connectedness of the generated patterns. As usual in general-to-specific learners, subsumption checks are not carried out between a pattern P and an examples e, if a pattern more general than P has been shown not to subsume e.

The results of the PTC dataset experiments, summarized in Table 1, differ according to the chosen relational logic representation of the molecules.

The first version **PTC-v1** uses a rather naive representation. Here, each molecular bond is represented by a single literal $bond(at1, at2, bondName)$, thus imposing a bond orientation (atom order) chosen at random. The second source of imprecision of this representation is that two variables in a pattern may represent the same atom, which does not make intuitive sense. For this setting, the difference in performance of RESUMER2 and Django was not significant.

The **PTC-v2** version deals with the deficiencies of the **PTC-v1** data representation. Here, every bond is represented by two literals $bond(at1, at2, bondName)$ and $bond(at2, at1, bondName)$. Furthermore, we added literals $different(a, b)$ for all pairs of atom-representing constants a and b in examples. In this setting, the representation size of examples grew to thousands of atoms. In this case, RESUMER2 was significantly faster than Django over the entire phase transition landscape.

In the **PTC-v3** experiment we reduce the size of the representation of examples by inserting the

$different(a, b)$ literal only for atoms a, b that both have a bond with a common atom. In this last case, RESUMER2 was again significantly faster than Django.

While results achieved in Section 5.1 on generated graph data indicated that RESUMER2's performance superiority over Django grows with increasing size of examples, the PTC domain experiments clearly confirm this observation.

6. Conclusions and Future Work

We have studied runtime distributional properties of subsumption testing, a procedure at heart of most relational data mining systems based on inductive logic programming. On graph data randomly sampled from two different generative models (Erdos-Renyi and scale-free) we have observed the gradual growth of the tails of the distributions as a function of the problem instance location in the phase transition space. To avoid the heavy tails, we have designed a randomized restarted subsumption testing algorithm RESUMER2. The algorithm is complete in that it correctly decides both subsumption and non-subsumption in finite time. A basic restarted strategy is augmented in RESUMER2 by allowing certain communication between odd and even restarts without losing the exponential runtime distribution decay guarantee resulting from mutual independence of restart *pairs*. We empirically tested RESUMER2 against the state-of-the-art subsumption algorithm Django on the generated graph data as well as on three versions of the predictive toxicology challenge (PTC) data set. RESUMER2 performs comparably with Django for relatively small examples (tens to hundreds of literals). For growing example sizes, RESUMER2 becomes vastly superior. In a PTC formulation with examples containing several thousands of literals, RESUMER2 solves subsumption instances by orders of magnitude faster than Django.

Due to the promising results obtained by RESUMER2, our next step will be towards its inclusion in an actual relational pattern discovery system. Some issues pertaining to the design of the RESUMER2 algorithm however call for further exploration. For example, Figure 3 seems to indicate that allowing more communication between each pair of subsequent restarts might bring further efficiency benefits, even at the price of completely sacrificing statistical independence between restart and thus losing the exponential runtime distribution decay guarantee for the restarted strategy. In particular, an interesting suggestion, made by a reviewer of this paper, is to use the information collected in the previous restart with a pre-set probability p. This parameter would then reflect the overall degree of dependence between successive restarts and would enable to study the effect of this dependence systematically in the range of $p \in [0; 1]$.

References

[1] Barabasi, A. L., Albert, R.: Emergence of scaling in random networks, *Science*, **286**(5439), 1999, 509–512.

[2] Bollobas, B.: *Modern Graph Theory*, Springer, 1998, ISBN 0387984887.

[3] Chen, H., Gomes, C., Selman, B.: Formal Models of Heavy-Tailed Behavior in Combinatorial Search, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, Springer-Verlag, 2001.

[4] Giordana, A., Saitta, L.: Phase Transitions in Relational Learning, *Machine Learning*, **41**(2), 2000, 217–251.

[5] Gomes, C. P., Fernández, C., Selman, B., Bessière, C.: Statistical Regimes Across Constrainedness Regions, *Constraints*, **10**(4), 2005, 317–337, ISSN 1383-7133.

[6] Gomes, C. P., Selman, B., Crato, N., Kautz, H. A.: Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems, *Journal of Automated Reasoning*, **24**(1/2), 2000, 67–100.

[7] Helma, C., King, R. D., Kramer, S., Srinivasan, A.: The predictive toxicology challenge 2000-2001, *Bioinformatics*, **17**(1), 2001, 107–108.

[8] Kirkpatrick, S., Selman, B.: Critical behavior in the satisfiability of random Boolean expressions, *Science*, **264**(5163), 27 May 1994, 1297–1301.

[9] Maloberti, J., Sebag, M.: Fast Theta-Subsumption with Constraint Satisfaction Algorithms, *Machine Learning*, **55**(2), 2004, 137–174, ISSN 0885-6125.

[10] Sebag, M., Rouveirol, C.: Tractable Induction and Classification in First-Order Logic via Stochastic Matching, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, 1997.

[11] Srinivasan, A.: A study of two sampling methods for analysing large datasets with ILP, *Data Mining and Knowledge Discovery*, **3**(1), 1999, 95–123.

[12] Walsh, T.: Search in a Small World, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann, San Francisco, CA, USA, 1999, ISBN 1-55860-613-0.

[13] Wu, H., van Beek, P.: Restart Strategies: Analysis and Simulation, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, Springer, 2003, ISBN 3-540-20202-1.

[14] Zelezny, F., Srinivasan, A., Page, D.: Randomised Restarted Search in ILP, *Machine Learning*, **64**(1–2), 2006, 183–208.