Graph sampling with applications to estimating the number of pattern embeddings and the parameters of a statistical relational model

Irma Ravkic, Martiň Znidaršič, Jan Ramon, Jesse Davis

Noname manuscript No.
(will be inserted by the editor)

Graph sampling with applications to estimating the number of pattern embeddings and the parameters of a statistical relational model

Irma Ravkic · Martin Žnidaršič · Jan Ramon · Jesse Davis

Received: date / Accepted: date

Abstract Counting the number of times a pattern occurs in a database is a fundamental data mining problem. It is a subroutine in a diverse set of tasks ranging from pattern mining to supervised learning and probabilistic model learning. While a pattern and a database can take many forms, this paper focuses on the case where both the pattern and the database are graphs (networks). Unfortunately, in general, the problem of counting graph occurrences is #P-complete. In contrast to earlier work, which focused on exact counting for simple (i.e., very short) patterns, we present a sampling approach for estimating the statistics of larger graph pattern occurrences. We perform an empirical evaluation on synthetic and real-world data that validates the proposed algorithm, illustrates its practical behavior and provides insight into the trade-off between its accuracy of estimation and computational efficiency.

Keywords graph sampling · graph pattern matching · parameter estimation · statistical relational learning

The first two authors contributed equally to this work and are ordered alphabetically.

Irma Ravkic
Department of Computer Science, KU Leuven, 3001 Heverlee, Leuven, Belgium
E-mail: irma.ravkic@cs.kuleuven.be

Martin Žnidaršič
Jožef Stefan Institute, Jamova cesta 39, SI-1000 Ljubljana, Slovenia
E-mail: martin.znidarsic@ijs.si

Jan Ramon
Department of Computer Science, KU Leuven, 3001 Heverlee, Leuven, Belgium
E-mail: jan.ramon@cs.kuleuven.be

Jesse Davis
Department of Computer Science, KU Leuven, 3001 Heverlee, Leuven, Belgium
E-mail: jesse.davis@cs.kuleuven.be

1 Introduction

Counting the number of times a pattern occurs in a database is an essential step in many applications. In frequent pattern mining, computing the support of an itemset, a sequence or a graph requires counting the number of times it occurs in the data. In supervised learning, many algorithms measure the correlations between features and target variables, which requires iterating over the examples that satisfy a given pattern. In probabilistic modeling, estimating the parameters of a model (e.g., constructing a conditional probability table for a Bayesian network) requires counting the number of times a set of variables takes on a specific combination of values in the data. Furthermore, mining and learning applications typically require addressing this counting problem for a large number of patterns. For example, pattern mining approaches compute the support for progressively more complex patterns and probabilistic model structure learning algorithms require estimating the parameters for a large number of candidate structures.

Unfortunately, in general, counting the number of occurrences of a graph pattern in a network is #P-complete. Most earlier work addressed problems such as graph characterization (Shervashidze et al 2009; Bordino et al 2008) or frequent pattern discovery (Inokuchi et al 2003; Wernicke 2005), which consider all possible subgraphs. However, these approaches tend to focus only on small patterns.

In this paper, we propose a new algorithm for approximately counting the number of embeddings of a pattern in a graph based on a theoretical sampling idea suggested by Fürer and Kasiviswanathan (2014) for larger unlabeled undirected graphs. For practical use in a data mining context, the theoretical algorithm of Fürer and Kasiviswanathan (2014) has several weaknesses and open questions. First, it assumes that all graphs are unlabeled and undirected. Second, it assumes that the input pattern has an ordered-bipartite decomposition (OBD) whereas not all interesting patterns will have an OBD. Third, it assumes that the pattern's OBD is provided to the algorithm. From a practical perspective, it is necessary to automatically construct an OBD (if one exists) for an input pattern. Finally, only the count of the number of embeddings is estimated.

This paper attempts to address the aforementioned concerns and makes the following contributions. First, we generalize the approach of Fürer and Kasiviswanathan (2014) to labeled and directed graphs. Second, we propose a heuristic approach to automatically search for an OBD for a given pattern, whereas it was previously assumed that the OBD was given. Third, while Fürer and Kasiviswanathan (2014) provide only an error bound, in this paper we also estimate the accuracy of computed statistics from the sample. Fourth, we implement and empirically evaluate the approach on two tasks whereas the work of Fürer and Kasiviswanathan (2014) was purely theoretical. The first task investigates the accuracy of the estimated counts. The second task evaluates the suitability of using the approach for parameter learning within statistical relational learning (SRL). Fifth, we investigate how the algorithm performs when some of the conditions required for the theoretical guarantees are violated. Namely, we investigate the algorithm's performance when it is given (1) a decomposition that is not a OBD, and (2) a data graph that is not an Erdős-Rényi random graph.[1] Finally, our implementation is publicly available.[2]

[1] Graphs where each edge is included, independent of all other edges, with probability p.

[2] For the code see: https://dtai.cs.kuleuven.be/software/gs-srl

2 Related work

Data in domains, such as the Semantic Web, social networks, citation networks, biology, relational learning and geography among others, are often naturally represented as a graph. Therefore, analyzing large graphs or networks is a very active research field. Furthermore, problems related to the analysis (searching, counting, etc.) of subgraphs in large network graphs attract significant attention. Next, we present some of the more prominent lines of research concerned with subgraph analysis and their relation to this paper.

Many of the following approaches employ subgraph identification, either by *subgraph isomorphism* or *subgraph homomorphism* as a subtask. Typically, they employ existing state-of-the-art algorithms for this task (e.g., Cordella et al (2004); Ullmann (1976)).

One line of work looks at defining features of graphs. Graph particles are a promising approach for *graph characterization and comparison* (Pržulj 2007; Shervashidze et al 2009; Bordino et al 2008). Graph particles are usually small subgraphs that contain up to five nodes, and are sometimes called graphlets. The distribution of graph particles can be used as a "fingerprint", a characteristic invariant, of a graph. Then, the similarity between pairs of graphs can be measured by comparing their distributions of graph particles. *Graph querying* (Di Natale et al 2010; Giugno and Shasha 2002) focuses on searching for a graph in a database of graphs. Here, most research focuses on identifying graph features (e.g., paths, walks, subtrees, small subgraphs, etc.) for filtering and indexing in order to enable faster searching and querying of graphs.

Frequent pattern mining in graph databases is another active topic of research (Yan and Han 2002; Inokuchi et al 2003). This usually entails finding frequent subgraphs, which are also called *motifs* in biological networks (Kashtan et al 2004; Wernicke 2005). Motifs are subgraphs that appear more often than expected according to a null model. Although subgraph isomorphism is an integral subtask in this problem, the main research focus is determining the significance of subgraphs, which is measured by their support in the data. Usually, these approaches only consider small subgraphs since the goal is to find all subgraphs that meet the support threshold. Some approaches (e.g., Baskerville et al (2007)) use approximate methods to scale to larger subgraphs.

Finally, general graph sampling approaches explore how to obtain a representative sample of a graph for a specific purpose (Leskovec and Faloutsos 2006). Some approaches, like the work of Zou and Holder (2010), rely on sampling for performing subgraph analysis. Zou and Holder (2010) first sample from a single large graph and then mine the sample for frequent subgraphs. They empirically assess the viability of various graph sampling approaches for frequent subgraph mining.

The two key differentiating factors of our work are that we focus on approximating the frequency of a subgraph in larger graph (i.e., the inverse task of Zou and Holder (2010)), and we consider relatively large subgraphs (up to 15 nodes). Perhaps the most closely related work is that of Bordino et al (2008) who approximately count the subgraphs for graph characterization. But again, they only consider small subgraphs.

Within SRL, there has been work on using approximations to scale up both parameter estimation and structure learning. The early work of Kok and Domingos

(2005) recognized the potential of sampling to improve the scalability of structure learning for Markov logic networks. Their approach offered the ability to both subsample the data, as well as the number of true groundings of a clause. More recently, Venugopal et al (2015) proposed an approach that improves the scalability of parameter learning in Markov logic networks by approximately counting the number of satisfied groundings of a clause. This problem is equivalent to counting paths in a graph. Venugopal et al (2015)'s approach is specific to Markov logic and is focused on the parameter learning task. The fast approximate counting (FACT) approach proposed by Das et al (2016) uses a graph database to scale learning and lifted inference. In contrast to this paper's sampling-based scheme, their approach is based on message passing and looks at the in and out degrees of a node relative to the maximum possible degree. Das et al (2016)'s approach does not offer any theoretical guarantees whereas this paper inherits the original guarantees from the work of Fürer and Kasiviswanathan (2014) (under certain conditions). Furthermore, Das et al (2016) focus solely on estimating a pattern's count, whereas we discuss computing additional statistics (see Section 8).

3 Background

Throughout this paper, we use the term *domain graph* to refer to the graph that represents the full dataset. We use the term *pattern graph* to refer to a small subgraph that we are evaluating (e.g., to estimate its frequency in the domain graph). Next, we introduce the necessary background information and terminology used throughout this paper.

3.1 Graph representation

A labeled graph is a tuple $G = (V, E, \Sigma, \lambda)$, where V is a set of vertices (also called nodes), $E \subseteq \big\{\{u, v\} \mid u, v \in V\big\}$ is a set of edges, Σ is a set of labels and $\lambda : (V \cup E) \to \Sigma$ is a function that assigns labels to nodes and edges. For a graph G, we refer to its set of vertices with $V(G)$, to its set of edges with $E(G)$, to its alphabet with Σ_G and to its labeling function with λ_G. For a graph G and a vertex $v \in V(G)$ the set of *neighbors* of v in G is $N_G(v) = \{u \in V(G) \mid \{u, v\} \in E(G)\}$. A graph H is a *subgraph* of a graph G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and $\forall x \in V(H) \cup E(H) : \lambda_H(x) = \lambda_G(x)$. The label set Σ can contain attribute-value pairs as well. More precisely, for a node v with label $attribute = value$ we denote the $value$ of v with $value(v)$. We also allow for wildcards # to be used as $value$. This means that the node can take on values, but we do not care in this particular case which value it is. We will often omit # in $attribute = \#$, and denote the label of the node with $attribute$.

Consider the labeled graph in Figure 1. The nodes in the graph denote people (men or women) together with their friendship or marriage relations to other people, and their satisfaction with their salary. Note that labels for people, salary and satisfaction are attribute-value pairs (e.g., $satisfaction = high$ for $mike$).

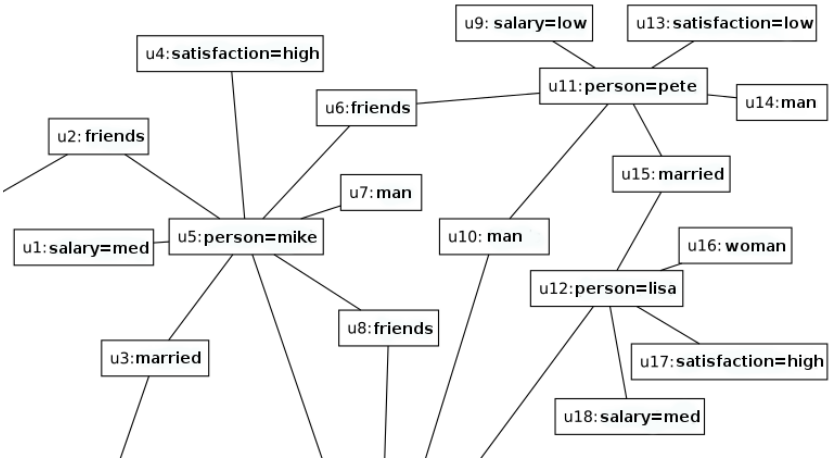

Fig. 1: An example of a labeled domain graph depicting a number of *people (men or women)* together with their *friendship* or *marriage* relations to other people and their *satisfaction* with their *salary*.

3.2 Pattern matching

There are multiple ways of performing the matching between two graphs. A *homomorphism* from a graph H to a graph D is a mapping $\varphi : V(H) \to V(D)$ such that (i) $\forall v \in V(H), \lambda_H(v) = \lambda_D(\varphi(v))$, (ii) $\forall \{u, v\} \in E(H)$, $\{\varphi(u), \varphi(v)\} \in E(D) \wedge \lambda_H\big(\{u, v\}\big) = \lambda_D\big(\{\varphi(u), \varphi(v)\}\big)$. With $\mathrm{emb}_{hom}(H, D)$ we denote the set of all homomorphisms between H and D. A *subgraph isomorphism* from H to D is a homomorphism φ from H to D such that $\forall u, v \in V(H), \varphi(u) \neq \varphi(v)$. We denote with $\mathrm{emb}_{iso}(H, D)$ the set of all subgraph isomorphisms between H and D. Whether homomorphism or subgraph isomorphism is more appropriate to use as a matching operator depends on the application. Our results are generic and apply to both cases. We use $\mathrm{emb}(H, D)$ to refer to $\mathrm{emb}_x(H, D)$ for any $x \in \{hom, iso\}$ and refer to its elements as *embeddings*.

A pattern over Σ is a graph P with $\Sigma_P = 2^\Sigma$ (i.e., the labels on vertices and edges of P are subsets of Σ). We use the term values to refer to the labels of the images of pattern vertices. In particular, for an embedding φ of P in D and a set $S \subseteq V(P)$, we define $val_{S:D}(\varphi) = \lambda_D \circ \varphi|_S = \{(u, \lambda_D(v)) \mid u \in S \wedge (u, v) \in \varphi\}$. We define the range of values for S of P in D by $\mathcal{R}ange(S : P, D) = \{val_{S:D}(\varphi) \mid \varphi \in \mathrm{emb}(P, D)\}$. We also define the multiset of values for S of P in D by $\mathcal{V}al(S : P, D) = \{val_{S:D}(\varphi)\}_{\varphi \in \mathrm{emb}(P,D)}$. So the essential difference between $\mathcal{R}ange(S : P, D)$ and $\mathcal{V}al(S : P, D)$ is that the first is a normal set (containing each value once) while the second is a multiset possibly containing the same value several times. For the special case $S = V(P)$, we define $val_D(\varphi) = val_{V(P):D}(\varphi)$, $\mathcal{R}ange(P, D) = \mathcal{R}ange(V(P) : P, D)$, and $\mathcal{V}al(P, D) = \mathcal{V}al(V(P) : P, D)$.

4 Decomposition of graphs

Decomposing graphs permits applying a divide-and-conquer strategy to complex problems and it represents a preprocessing step for many graph theory techniques. In this paper we use this strategy for obtaining the count of a pattern's embeddings in a graph. The general idea is to decompose the vertices involved in a complex pattern into multiple partitions. The matching task then becomes simpler as only the vertices in a single partition need to be matched simultaneously, with the current partition's matching being conditioned on the assignment given to the vertices in previous partitions. We describe two decompositions: an ordered bipartite decomposition (Fürer and Kasiviswanathan 2014) and an arbitrary decomposition.

4.1 Ordered bipartite decompositions

Fürer and Kasiviswanathan (2014) introduced the concept of an ordered bipartite decomposition (OBD), which is a crucial component of our approach. We now define the relevant terminology and describe this decomposition.

An *independent set* is a set of vertices in a graph, no two of which are adjacent. A *graph partition* is a division of a graph's vertices and edges into smaller components with specific properties. A *bipartite* graph $G = (U, V, E, \Sigma, \lambda)$ is a graph whose vertices can be divided into two disjoint independent sets U and V such that every edge in $e \in E$ connects a vertex in U to one in V. A bipartite graph can be seen as coloring the nodes in U one color, and the nodes in V another color, where each edge connects two vertices with different colors. For example, representing the relationship of football players to their clubs can be modeled as a bipartite graph: at a specific point in a time, each player is associated with one club.

Informally, an OBD decomposes a graph pattern into an implicit sequence of bipartite graphs by numerically labeling all the vertices in a graph pattern. The labeling must ensure that (1) each edge connects two vertices with different labels, and (2) all of a vertex's neighbors with a higher label have the same label. Now we formally define an OBD.

Definition 1 (Ordered bipartite decomposition (Fürer and Kasiviswanathan 2014)) An ordered bipartite decomposition of a graph $P = (V_P, E_P)$ is a sequence $V_1, ..., V_l$ of subsets of V_P such that:

1. $V_1, ..., V_l$ form a partition of V_P
2. Each of the V_i (for $i \in [l] = 1, ..., l$) is an independent set in P
3. $\forall v \exists j$ such that $v \in V_i$ implies $N_P(v) \subseteq (\bigcup_{k<i} V_k) \cup V_j$

Fürer and Kasiviswanathan (2014) defined an OBD's *size* as the number of partition classes in it and its *width* as the number of vertices in the largest partition class.

Fürer and Kasiviswanathan (2014) proved that a number of graph classes (e.g., trees, grids, etc.) will always have an OBD. However, not all graphs will have an OBD. Consider the two graphs in Figure 2. Graph (a) is a triangle graph. There exists a trivial decomposition $\{v1\}, \{v2\}, \{v3\}$ satisfying Properties 1 and 2 of Definition 1. However, it is not possible to satisfy Property 3 for this graph,

hence, there is no OBD for it. The OBD for graph (b) is $\{v1\}, \{v2\}, \{v3, v4\}$ where each partition is labeled with, for example, I, II, and III. The reader can check that all three properties hold. For example, for the vertex $v2$ labeled as II, all the neighbors ($v3$ and $v4$) are labeled identically with a higher label III.

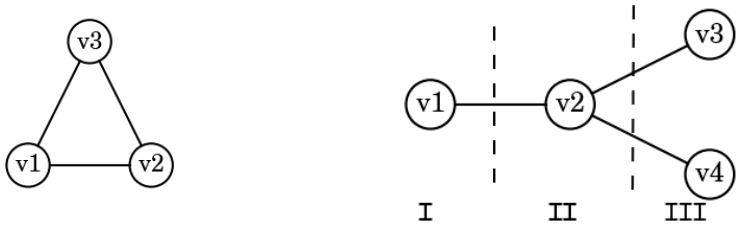

Fig. 2: An example of a) a non-bipartite triangle graph for which there does not exist an OBD, b) a graph that has the following OBD: $\{v1\}\{v2\}\{v3, v4\}$ (where $\{v1\}$ is the first partition, $\{v2\}$ is second partition, and $\{v3, v4\}$ is the third partition).

4.2 Arbitrary decompositions

As many potentially interesting patterns may not have an OBD, we also want to investigate how the Fürer and Kasiviswanathan (2014) approach works when the algorithm is provided with a decomposition that is not an OBD. Here, we will still need a decomposition of the graph pattern and in this case, we will work with an arbitrary decomposition.

Definition 2 (Arbitrary decomposition) An arbitrary decomposition of a graph $P = (V_P, E_P)$ is a sequence $V_1, ..., V_l$ of subsets of V_P such that:

1. $V_1, ..., V_l$ form a partition of V_P
2. Each of the V_i (for $i \in [l] = 1, ..., l$) is an independent set in P

5 Counting the number of embeddings in graphs

The problem of collecting statistics about the occurrences of a graph pattern in a network can be formalized as:

- **Given:** a domain graph D, a pattern P, and a statistic of interest f
- **Find:** $f(P, D)$

The basic procedure that is used in all the approaches in this paper is to find all extensions to a partial embedding φ. For that purpose we use the procedure in Algorithm 1. The function receives a domain graph D, a pattern graph P, a partial embedding φ and a set of vertices $S \subseteq P$ for which we want to find the images. The function loops through all the vertices $v \in S$ and tries to find images of v which are then stored in the C_v variable. It then constructs the set

of all possible embeddings such that each embedding is the concatenation of the partial embedding φ and one of the combination of images found for vertices in S. The test $\varphi' \in emb(P[{\varphi'}^{-1}(D)], D)$ checks if the edges and labels are preserved by φ'. If subgraph isomorphism is used as the matching operator (rather than homomorphism), one must additionally check that no two vertices receive the same image.

Algorithm 1 Extend partial embeddings.

1: **function** PARTIALEXT(D, P, φ, S)
2: **for all** $v \in S$ **do** ▷ Loop through vertices in the partition
3: **if** $\exists (x, y) \in \varphi : x \in N_P(v)$ **then**
4: $C_v \leftarrow \{u \in N_D(y) \mid \lambda(u) \in \lambda(v)\}$
5: **else**
6: $C_v \leftarrow \{u \in V(D) \mid \lambda(u) \in \lambda(v)\}$
7: **return** $\{\varphi' \mid \varphi' = \varphi \cup \{(v, u_v)\}_{v \in S} \wedge \forall v \in S : u_v \in C_v \wedge \varphi' \in emb(P[{\varphi'}^{-1}(D)], D)\}$

Next, we introduce an exhaustive approach and a sampling approach for finding the number of embeddings of a pattern in a domain graph.

5.1 Exhaustive approach

Algorithm 2 shows an exhaustive algorithm for computing all embeddings of a pattern. The algorithm receives a domain graph D, a pattern graph P, and a fixed ordering $\mathcal{O}$ on the pattern vertices as input. We use $\mathcal{O}_i$ to denote the i-th vertex in the ordering $\mathcal{O}$, that is, the i-th vertex that is assigned an image.

Starting from an empty embedding, the algorithm works as follows. The PARTIALEXT function finds all extensions to a partial embedding φ by assigning an image to the $\mathcal{O}_{|\varphi|+1}$th vertex. Then, in line 7 these images are appended to the embedding. The test $\varphi' \in emb(P[{\varphi'}^{-1}(D)], D)$ checks if the edges and labels are preserved by φ'. If subgraph isomorphism is used as the matching operator (rather than homomorphism), one must additionally check that no two vertices receive the same image.

When choosing an ordering $\mathcal{O}$, a typical heuristic is to first select images for vertices that have a small number of possible images. We require that every vertex, except the first one, is adjacent to at least one other vertex which precedes it in the ordering. A large literature exists on heuristics and various forms of lookahead (Ullmann 1976), which is outside the scope of our paper.

Algorithm 2 Exhaustive approach.

1: **function** ALLEMB(graph D, pattern P, ordering $\mathcal{O}$)
2: **r**eturn ALLEMB(D, P, $\mathcal{O}$, {}) ▷ Return a set of embeddings
3: **function** ALLEMB(D, P, $\mathcal{O}$, φ)
4: **if** $|\varphi| = |V(P)|$ **then** ▷ If the pattern is fully embedded
5: **return** $\{\varphi\}$
6: **else** ▷ Repeat the procedure for each found extension
7: **return** $\cup_{\varphi' \in \text{PARTIALEXT}(D,\ P,\ \varphi,\ \{\mathcal{O}_{|\varphi|+1}\})}$ALLEMB($D,\ P,\ \mathcal{O},\ \varphi'$)

5.2 Random vertex sampling

Algorithm 3 presents an approximate algorithm for computing all embeddings of a pattern based on sampling. It receives the same input as the exhaustive approach (i.e., function AllEmb in Algorithm 2). At a high level, this algorithm samples the first part of the embedding randomly and then exhaustively computes all of its completions. This algorithm differs from the exhaustive algorithm in that it first finds an image for the first vertex $\mathcal{O}_1$ by randomly sampling, with replacement, from the set of vertices with a matching label instead of exhaustively iterating over all of them.

Algorithm 3 Random vertex sampling approach.

```
1: function RndV1Emb(graph D, pattern P, ordering O)
2:     M ← {}
3:     R ← {u ∈ V(D) | λ(u) = λ(O_1)}          ▷ Find images for the first vertex of P
4:     while not timeout do
5:         Select a random u ∈ R
6:         M ← RndV1Emb(D, P, O, {(O_1, u)})
7:         M ← M ∪ {M}
8:     return M

9: function RndV1Emb(D, P, O, φ)               ▷ Find extensions for the partial embedding
10:    if timeout then
11:        return {}
12:    else if |φ| = |V(P)| then
13:        return {φ}
14:    else
15:        M ← {}
16:        for all φ' ∈ PartialExt(D, P, φ, {O_{|φ|+1}}) do
17:            M' ← RndV1Emb(D, P, O, φ')
18:            M = M ∪ M'
19:        return M
```

One can observe that every embedding has the same probability of being encountered by this algorithm. However, several embeddings may be included in the sample at the same iteration, and hence these may not be independent. Therefore, in line 7 we form a sample by a set of sets of embeddings (indicating these dependencies explicitly) rather than just merging all sets M into one large unstructured sample $\mathcal{M}$. This algorithm is simple but has the potential disadvantage that, for large patterns, it may spend all its time finding embeddings for the first sampled vertex.

Similar strategies have previously been proposed, such as in the work on triangle counting (c.g., Jowhari and Ghodsi (2005)).

6 The Fürer-Kasiviswanathan algorithm

The algorithms we introduced in the previous section represent the baseline algorithms for finding the pattern embeddings. However, they can be very costly for larger patterns and domain graphs. Next, we propose an extension of a theoreti-

cal sampling idea suggested by Fürer and Kasiviswanathan (2014) for unlabeled undirected graphs.

Figure 3 depicts one iteration of the the Fürer and Kasiviswanathan algorithm. Each iteration of the algorithm returns a single embedding and an estimate for the pattern's total number of embeddings. The key intuition behind the algorithm is that overall problem of estimating the number of embeddings of a pattern can be broken down into the problem of estimating the number of embeddings for each partition $\mathcal{V}_i$ of the OBD. A single iteration sequentially goes through all of the OBD's partitions and builds up both an embedding of the pattern and an estimate of the pattern's total number of embeddings. For each partition, it finds all possible extensions of the current partial embedding by matching the vertices in the partition. It then randomly samples one of these extensions and multiplies its estimate of the pattern's number of embeddings by the number of possible extensions found for the current partition.

When searching for an OBD, the crucial step is to ensure that each partition is small. Since all vertices in a partition must be simultaneously matched, it is easier to a find a legal extension when fewer vertices must be matched. If the decomposition provided as input is not an ordered bipartite decomposition, the sampling algorithm will still converge to the correct value, but the convergence speed guaranteed by Fürer and Kasiviswanathan (2014) no longer holds.

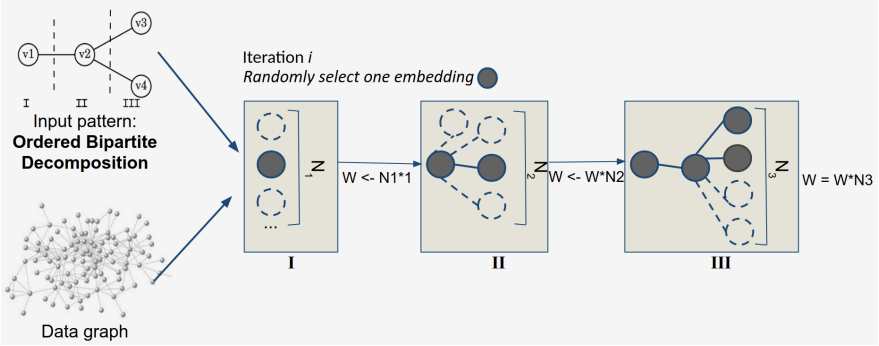

Fig. 3: An illustration of one iteration of the Fürer and Kasiviswanathan approach. The panel labeled I shows how one node is selected from N_1 total possibilities to match the vertex in the first partition. The panel labeled II shows how one node is selected from N_2 total possibilities to match the vertex in the second partition, and the new count is $N_1 \times N_2$. Finally, the panel labeled III shows how two nodes are selected from N_3 total possibilities to match the two vertices in the third partition, and the final count is $N_1 \times N_2 \times N_3$.

Algorithm 4 shows our extension of the Fürer and Kasiviswanathan approach to labeled graphs. The algorithm consists of two parts: the control loop (lines 2-7), and the single attempt procedure (lines 9-18). It receives the same parameters as Algorithm 3, except that it receives a decomposition $\mathcal{V}$ instead of an ordering $\mathcal{O}$. While the time limit is not exceeded, the control loop randomly samples, with replacement, a path in the search space that possibly leads to an embedding. The

Algorithm 4 Fürer and Kasiviswanathan approach.

```
1: function FK-V(domain graph D, pattern subgraph P, decomposition V)
2:     M ← {}
3:     while not timeout do
4:         M ← FK-V(1, D, P, V, 1, {})
5:         if M ≠ ∅ then
6:             M ← M ∪ {M}
7:     return M
8:
9: function FK-V(W, D, P, V, i, φ)
10:    if i > |V| then
11:        return ({(φ, W)})                          ▷ If all partitions embedded
12:    else if timeout then
13:        return {}
14:    C ← PartialExt(D, P, φ, V_i)                   ▷ Find all extensions for the partition
15:    if C = {} then
16:        return {}
17:    Select φ' randomly from C                      ▷ Randomly select one extensions
18:    return FK-V(W × |C|, D, P, V, i + 1, φ')
```

function FK-$\mathcal{V}$ returns a set of pairs (φ, W) where φ is the embedding found and W is a positive number equal to the inverse probability that an attempt will find that particular embedding. The expected value of this W (assuming implicitly 0 for the unsuccessful attempts) is the total number of embeddings.

A call to FK-$\mathcal{V}$ identifies C, the set of all possible extensions to the current partial embedding. It randomly samples a set of images for a partition $\mathcal{V}_i$ and extends the partial embedding accordingly. While it only samples one element from C, it uses the size of C as an estimate for the number of subpattern represented by $\mathcal{V}_i$. It then recurses, with both the extended partial embedding φ' and an updated count estimate of $W \times |C|$ (which becomes 0 when the attempt fails).

For example, if $\mathcal{V}_i = \{w1, w2, w3\}$, each vertex $w \in \mathcal{V}_i$ has a set of candidate matches C_w in D. Suppose these are $C_{w1} = \{a1, a2\}$, $C_{w2} = \{a3\}$ and $C_{w3} = \{a1, a4, a5\}$. Then each element of the Cartesian product $C_{w1} \times C_{w2} \times C_{w3}$ is a valid match for the vertices in $\mathcal{V}_i$ if the matching operator is homomorphism. In this case, $|C|$ would equal 6. In the case of subgraph isomorphism, embeddings mapping several vertices to the same image (here $\{(w1, a1), (w2, a3), (w3, a1)\}$) must be eliminated, so $|C|$ would equal 5.

7 Computing ordered bipartite decompositions

When estimating the number of embeddings for pattern P, Algorithm 4 requires an ordered bipartite decomposition (OBD) for P. For efficiency, the size of each partition class in an OBD is important. Having more vertices in a single partition requires matching multiple vertices simultaneously, which is more computationally expensive. In particular, the size of set C in line 14 of Algorithm 4 may be exponential in $\mathcal{V}_i$. This implies that we want to minimize the number of vertices in each partition class. This also suggests that we prefer having OBDs that are as large as possible (i.e., have as many partition classes as possible). Fürer and Kasiviswanathan (2014) assume that a pattern's OBD is given. Thus, we propose two different search strategies for automatically finding an OBD.

7.1 Exact search

Algorithm 5 presents a level-wise search for finding the largest possible OBD (i.e., the one with the most partition classes). First, it checks if P contains a triangle, because then no OBD exists. The maximal OBD size for pattern P is $|V(P)|$, which is the number of vertices in P. Starting with the largest possible size $s = |V(P)|$, the algorithm performs a search through the space of possible OBDs. In each iteration, it checks whether an OBD of size s exists by generating all possible candidate OBDs of size s. This entails enumerating all ways of dividing $V(P)$ into s partition classes, and then building one candidate for each possible ordering of the partition classes. It then applies the VALID_OBD function to check if each candidate is a valid OBD (i.e., it satisfies the requirements of Definition 1). As soon as it finds a valid OBD, it is returned and the search terminates. If no OBD of size s is found, then s is decremented, and the search proceeds. If an OBD is found, its size is maximized because all larger decompositions were evaluated and none were valid OBDs.

Algorithm 5 Level-wise exact OBD search

```
1: function FindLargestOBD(graph P)
   return Largest ODB for P or ∅ if no OBD exists
2:     if contains_triangle(P) then
3:         return ∅
4:     s ← |V(P)|
5:     while s > 0 do
6:         for all ordered partitions 𝒱 of V(P) of size s do
7:             if Valid_OBD(𝒱) then
8:                 return 𝒱
9:         s ← s − 1
10:    return ∅
```

7.2 Greedy search

Algorithm 6 presents a depth-first search for finding an ODB for a pattern P. It greedily searches the most promising path and returns the first OBD found. Because it performs backtracking, it is guaranteed to find an OBD if one exists. However, its greedy nature means that it is not guaranteed to find the largest valid OBD.

The main algorithm loops through each node n in P, and builds a partial OBD (n) consisting of a single partition class containing n. (If there exists an OBD, then there exists one with a singleton as first partition class.) Next, it calls GreedyOBD which recursively extends the partial OBD by adding one partition class at a time.

If GreedyOBD receives a valid and complete OBD (i.e., it contains all the nodes in P), it terminates and returns the current OBD. Otherwise, it generates a set of candidate extensions for $\mathcal{V}$. To do so, it first builds a graph $G' = (R, L)$, which has one node for each vertex in P that has not been assigned a partition class in the partial OBD $\mathcal{V}$. It has one edge $\{x, y\} \in L$ for each $x, y \in R$ such that

x and y share a neighbor in P that already appears in $\mathcal{V}$. This implies that x and y must be in the same partition class. Because a node $x \in R$ may neighbor multiple nodes in $\mathcal{V}$, the set of candidate extensions $\mathcal{C}$ consists of each connected component in G'. Next, line 14 checks if any candidate partition requires assigning the same label to neighboring vertices in P, which would violate the definition of an OBD, meaning no complete OBD can be constructed by extending this partial ODB. In this case, the algorithm backtracks. Otherwise, it creates a set of candidate extensions by concatenating each element (i.e., set of vertices) in $\mathcal{C}$ as a partition class to the end of $\mathcal{V}$. It sorts the set of candidates from smallest to largest on the basis of the size of the newest partition class. This heuristic is an attempt to favor OBDs with small partition classes. It then recursively calls GreedyOBD on each candidate in this order.

Algorithm 6 Depth-first greedy OBD search.

```
 1: function GreedyFindOBD(graph P)
    return OBD for P or ∅ if no OBD exists
 2:     for all vertices v ∈ V(P) do
 3:         result = greedyOBD(({v}), P)
 4:         if result ≠ ∅ then
 5:             return result
 6:     return ∅

 7: function greedyOBD(partial OBD 𝒱, graph P)
                                          ▷ 𝒱 contains all nodes of P and is a valid OBD
 8:     if Complete(𝒱) && Valid_OBD(𝒱) then
 9:         return 𝒱
10:     else
                                                        ▷ Vertices not yet in partition class
11:         R ← V(P) \ ∪_{S∈𝒱} S
12:         L ← {{x, y} ∈ R | ∃V ∈ 𝒱 : N_P(x) ∩ V ≠ ∅ ∧ N_P(y) ∩ V ≠ ∅}
13:         𝒞 ← ConnectedCompents((R, L)))
14:         if ∃C ∈ 𝒞, ∃x, y ∈ V(C) : {x, y} ∈ E(P) then
                                                  ▷ x and y must be apart ⇒ no OBD possible
15:             return ∅
16:         for all C ∈ 𝒫 do
17:             result ← greedyOBD(𝒱 · V(C), P)
18:             if result ≠ ∅ then
19:                 return result
20:     return ∅
```

7.3 Empirical evaluation of OBD search

We empirically compare the run time performance of the two OBD search algorithms. We generate all simple connected graphs with between five and eight vertices and use both Algorithm 5 and Algorithm 6 to find an OBD for each generated pattern. Table 1 reports the mean, median and maximum run time in seconds for both algorithms on each pattern size. While the depth-first approach does not guarantee finding an optimal solution, on average it is several orders of magnitude faster than the level-wise approach. As it scales better than the exact approach,

our empirical evaluation in Section 9 employs the depth-first search approach to find an OBD for each evaluated pattern.

	Exact			Greedy		
|V|	Mean	Median	Max	Mean	Median	Max
5	0.0014	<0.0001	0.0096	0.0002	0.0003	0.0004
6	0.0106	<0.0001	0.1611	0.0004	0.0004	0.0010
7	0.0618	<0.0001	2.8392	0.0005	0.0005	0.0025
8	0.3894	<0.0001	54.9054	0.0008	0.0007	0.0084

Table 1: This table reports the mean, median and maximum run times in seconds for the exact and depth-first search algorithms for finding an OBD on all simple connected graphs with between five and eight vertices.

8 Computing statistics

The algorithms described in the previous sections all generate a sample of embeddings. In this section we will discuss how we can compute a number of statistics from such samples, and we will study their accuracy.

8.1 Common statistics

Let P be a pattern and D a domain graph. We first define the statistics we will consider.

Count. The *count* statistic just returns $|\mathcal{V}al(P, D)|$, the number of embeddings of P in D.

Mean and variance Let $t : \mathcal{R}ange(P, D) \to \mathbb{R}$ be a function mapping values of embeddings on real numbers. Then, $\mu_t(X) = \sum_{x \in X} t(x)/|X|$ and $var_t(X) = \sum_{x \in X}(t(x) - \mu_t(X))^2/|X|$.

Confusion matrix. Some vertices of the pattern could be labeled by finite sets. In that case, it may be interesting to compare the relative frequencies with which these vertices map on vertices with certain labels.

More formally, let $S \subseteq V(P)$ be a set of vertices of P. For $x \in \mathcal{R}ange(S : P, D)$ a possible value of the vertices in S and $y \in \mathcal{V}al(P, D)$ the value of a particular embedding, we define $I_{S \to x}(y) = 1$ if $y|_S = x$ and $I_{S \to x}(y) = 0$ if $y|_S \neq x$. Next, for a set of values X, $freq_{S \to x}(X) = \mu_{I_{S \to x}}$ and hence $freq_{S \to x}(P, D)$ is the fraction of embeddings of P in D for which the vertices in S get the value x. We define $freq_S(P, D) = \{(x, freq_{S \to x}(P, D) \mid x \in \mathcal{R}ange(S : P, D)\}$. $freq_S(P, D)$ is a confusion matrix (or tensor), as for each possible combination of values for the vertices in S it gives the observed (relative) frequency.

8.2 Estimators

We now discuss how to compute statistics from samples. In the above, we defined the elements of the confusion matrix statistic using a mean statistic, so we can

limit our discussion to *count*, μ and *var* here. In the empirical evaluation section, we will focus more on the confusion matrix statistic.

Let us first consider the random sampling. It returns a set $\mathcal{M} = \{M_i\}_{i=1}^{n}$ of groups of embeddings with n the number of main loop iterations of the algorithm. Each group $M_i = \{\varphi_{i,j}\}_{j=1}^{n_i}$ of embeddings is generated together during one iteration of random and shares the same image for the first pattern vertex. We can estimate $count(P, D)$ by $count^{rnd}(P, D) = |V(P)| \sum_{i=1}^{n} n_i/n$. We can estimate $\mu_t(P, D)$ by $\hat{\mu}_t^{rnd} = \sum_{i=1}^{n} \sum_{j=1}^{n_i} t(val_D(\varphi_{i,j}))/\sum_{i=1}^{n} n_i$. This is a consistent estimator (it converges to $\mu_t(P, D)$ when n gets large) but even though $count^{rnd}$ and $\sum_{i=1}^{n} \sum_{j=1}^{n_i} t(val_D(\varphi_{i,j}))$ are unbiased estimators of the count and the sum, $\hat{\mu}_t^{rnd}$ is not an unbiased estimator itself. For instance, suppose that the first vertex has two possible images u_1 and u_2, u_1 is in three embeddings with t-values 1, 2 and 3 and u_2 is in one embedding with t-value 6. Then, a single iteration of the sampling algorithm will either produce values 1, 2 and 3 leading to an estimation $(1 + 2 + 3)/3 = 2$ or a single value 6, leading to an estimate of 6. On average, we expect an estimation of 4, while the average of all values is 3.

Next, we consider the FK-$\mathcal{V}$ algorithm. It returns a set $M = \{(\varphi_i, w_i)\}_{i=1}^{n}$ of independently sampled weighted embeddings. Fürer and Kasiviswanathan (2014) showed that $count^{fk}(P, D) = \sum_{i=1}^{n} w_i/n$ is an unbiased estimator. For μ_t we can derive an unbiased estimator too (but omit details due to lack of space). Let $W = \sum_{i=1}^{n} w_i$ and $W_2 = \sum_{i=1}^{n} w_i^2$. Then, $\hat{\mu}_t^{fk}(P, D) = \sum_{i=1}^{n} w_i t(val_D(\varphi_i))/W$ is an unbiased estimator of $\mu_t(P, D)$. Moreover, we can estimate the square error from the sample with $\mathbb{E}[\hat{\mu}_t^{fk}(P, D) - \mu_t(P, D) = (W_2/W.(W^2 - W_2))$ $\sum_{i=1}^{n} w_i(t(val_D(\varphi_i)) - \hat{\mu}_t^{fk})^2$. It is straightforward to show that these estimators remain unbiased even if Fürer-Kasiviswanathan is run with a decomposition which is not an OBD. The main difference is that in that case the error converges more slowly to zero as the sample size increases.

9 Experiments and results

The goal of the empirical evaluation is to address the following questions:

- **Q1:** How well do the sampling algorithms approximate the true embedding statistics obtained using the exhaustive approach?
- **Q2:** How does having an OBD versus an arbitrary decomposition (AD) affect the performance of the Fürer-Kasiviswanathan algorithm?
- **Q3:** How does the method of determining an arbitrary decomposition affect the performance of the Fürer-Kasiviswanathan algorithm?
- **Q4:** How does the Fürer-Kasiviswanathan algorithm perform on random and non-random graphs?
- **Q5:** How does using the Fürer-Kasiviswanathan algorithm to compute approximate counts affect weight learning and inference in statistical relational learning?
- **Q6:** How well does the Fürer-Kasiviswanathan algorithm perform on longer patterns?

The first two questions attempt to assess the real-world viability of the algorithm described in this paper, while the last three questions attempt to provide more insight into the Fürer and Kasiviswanathan approach.

The majority of the empirical evaluation focuses on comparing the performance of the following five approaches:

Exhaustive: This is the approach of Algorithm 2.
Random: This is the approach of Algorithm 3 that randomly samples an initial vertex in the domain graph from which to start the search for the embeddings of a pattern.
FACT: This is the approach proposed by Das et al (2016). This method approximately counts the satisfied instances by using the summary statistics of in- and out-degrees of the graph representing the database.
FK-OBD: This is the extended Fürer and Kasiviswanathan approach outlined in Algorithm 4 that uses depth-first search to find an OBD for a pattern. It only applies to generated patterns that have an OBD.
FK-AD: This the extended Fürer and Kasiviswanathan approach outlined in Algorithm 4 that works with an arbitrary decomposition of a pattern. To construct an arbitrary decomposition, we simply flatten the OBD, which means that each partition contains exactly one node.

Next, we describe our datasets, the experimental methodology, and then we present and discuss the results.

9.1 Datasets

We evaluate the algorithms using 12 synthetic datasets and seven real-world datasets. Our real-world datasets are initially stored in a relational format. We use a standard transformation (e.g., Richards and Mooney (1992)) to convert these datasets into a graph form. The general idea is that graphs consist of constants connected through their relations.

9.1.1 Synthetic datasets

We created a series of synthetic datasets that contain the following node types: *person, salary, satisfaction, man, woman, married* and *friends*. We create two types of graphs by varying the way friend nodes are connected within the graph. The first denoted with S adds the nodes according to the power law distribution of the Barabási-Albert algorithm (Barabasi and Albert 1999). The second, denoted with R, employs the Erdős-Rény random graph model, and adds them randomly. Finally, each person receives a salary and a satisfaction, 40% of the people are male, the rest are female, and 70% of the people are married. Table 2 shows the synthetic graphs and their characteristics.

9.1.2 Real-world datasets

We consider a standard bibliography dataset, three well-known relational learning datasets, and three datasets from the Stanford Network Analysis Platform (SNAP) dataset collection.[3] For the real-world datasets, we transformed directed or labeled edges into 2-paths with an intermediate labeled vertex such that our graphs only

[3] https://snap.stanford.edu/data/

			Power Law		Random		
#Nodes	#Edges	#*Friends*	Name	MaxDeg	Name	MaxDeg	Density
143,445	236,890	99,945	S1	551	R1	32	1.56×10^{-5}
24,140	43,280	19,790	S2	383	R2	29	3.13×10^{-5}
14,295	23,590	9,945	S3	261	R3	26	6.26×10^{-5}
93,485	136,970	49,985	S4	158	R4	26	1.61×10^{-4}
243,290	436,580	199,790	S5	174	R5	36	2.31×10^{-4}
9,335	13,670	4,985	S6	131	R6	24	3.12×10^{-4}

Table 2: Characteristics of the synthetic graphs, which are sorted in descending order by density. The maximum degree (MaxDeg) is the maximum number of outgoing edges of the nodes, and the density is calculated as $Density = \frac{2m}{n(n-1)}$, where m is the number of edges and n is the number of nodes in the graph.

have vertex labels. Next, we describe the datasets in more detail. Table 3 lists the properties of the real-world datasets we used.

DBLP dataset[4] is a bibliography database from the ArnetMiner repository (Tang et al 2008). We only consider data from 2001 and 2002, to ensure that the Exhaustive approach is computationally feasible. The DBLP domain graph consists of *paper, coauthored, reference* and *citations* nodes.

Dataset	#Nodes	#Edges	Average Degree	MaxDeg	Density
DBLP	393,230	447,650	2.28	1,036	$5.7e-06$
Yeast	16,233	18,355	2.26	124	0.00014
IMDB	548	782	2.85	29	0.005
WEBKB	1,477	61,766	83.63	286	0.057
Facebook (SNAP)	28,057	112,252	8.00	1,051	0.0003
Enron e-mails (SNAP)	36,692	183,831	10.02	1,383	0.0003
Amazon (SNAP)	334,863	925,872	5.53	549	1.65e-05

Table 3: Characteristics of the real-world graphs. The average degree represents the average number of outgoing edges of the nodes, the maximum degree is the maximum number of outgoing edges of the nodes, and the density is calculated as $Density = \frac{2m}{n(n-1)}$, where m is the number of edges and n is the number of nodes in the graph. Note that the number of nodes in our SNAP datasets is larger than specified on their website because in our representation both objects (i.e., users, e-mails) and their attributes appear as nodes.

The following three datasets are commonly used in relational learning

Yeast is about protein-protein interactions in yeast (Mewes et al 2000; Davis et al 2005). This paper uses a modified version (Van Haaren et al 2015) from the Alchemy repository, such that all self-referential links are removed.[5] The graph has the following node types: *protein, location, function, phenotype, class, enzymes,* and *interaction.*

[4] http://www.informatik.uni-trier.de/ ley/db/index.html

[5] http://alchemy.cs.washington.edu

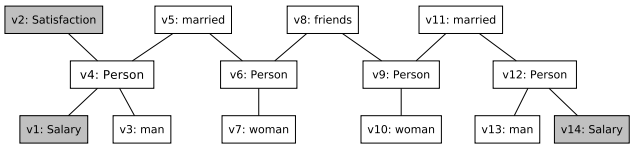

Fig. 4: Pattern graph $P_{sat-sal}$ for our example. Shaded nodes are the target nodes of the pattern.

IMDB consists of *movies* which are described by a *genre*, the movie's *directors*, and the *actors* who appeared in the movie. Actors and directors have a specific *gender*.

WEBKB consists of labeled *pages* and their hyperlinks from computer science departments of four universities.

The following three datasets come from SNAP:

Facebook consists of *users* from Facebook connected through *friendship* relations and where each user has a number of anonymized features such as *birthday*, *gender*, *education type*, *education degree*, *hometown*, and *language*.

Enron is a communication network consisting of email addresses such that if an address i sent at least one email to address j, the graph contains an undirected edge from i to j.

Amazon is a network consisting of products from Amazon such that if a product i is frequently co-purchased with product j, the graph contains an undirected edge from i to j.

9.2 Experimental methodology

Next, we introduce our experimental methodology which involves generating and evaluating patterns.

9.2.1 Generating pattern graphs

Synthetic pattern graph For the synthetic dataset we only consider one pattern graph, which is depicted in Figure 4. This pattern is a slight modification of a phenomena described by Ariely (2008).

Generating pattern graphs for real-world data We want to evaluate our approach on a variety of patterns. However, even with a restriction on the pattern size, it is computationally too demanding to evaluate all possible candidate patterns. Hence, we generate patterns of increasing size in a level-wise manner, and evaluate a subset of patterns of each size.

Each pattern is an undirected graph. We vary the pattern size from 4 to 10 nodes. Given a set C_L of patterns of size L, we generate a candidate set C_{L+1} of patterns of size $L+1$ by enumerating all valid extensions of each pattern in C_L. Each pattern is extended by first adding a new node to it. We consider adding all node types. Then, new candidates are created for all ways of adding up to

$m = n \cdot (n-1)/2$ edges between the new node and n existing nodes in the pattern. We remove duplicates from the candidate set by checking for isomorphism between pairs of candidate patterns.

We focus our evaluation on a sample of patterns where each selected pattern is neither too frequent nor too infrequent as these tend to be more interesting patterns for analysis. Thus, we randomly sample patterns from C_{L+1} and run FK-OBD or FK-AD for one hour on the larger datasets, and for five minutes for smaller datasets. FK-OBD is used if a pattern has an OBD, and FK-AD if it does not. We continue sampling until we find 100 patterns per each pattern size that meet the following criterion:

$$\sqrt{M} - 3 \cdot N_{std} \leq N_{avg} \leq M + 3 \cdot N_{std},$$

where M denotes the number of nodes in the data graph, N_{avg} denotes the average and N_{std} the standard deviation of the estimated number of embeddings after the one-hour run.

9.2.2 Pattern evaluation

We evaluate all algorithms on the selected patterns for each size. We run each algorithm for a maximum of T time units and record intermediate results after each K time units. For DBLP, Enron, and Amazon, we set T to ten hours and K to five minutes. For Yeast, IMDB, WEBKB, Facebook, and the synthetic datasets, T is ten minutes and K is five seconds. This means that we record 120 intermediate results for each pattern, regardless of the dataset or method.

9.2.3 Evaluation tasks and metrics

We consider two different evaluation tasks.

Frequency estimation This corresponds to the standard pattern mining task, where the support of a pattern in the data is calculated.

Parameter estimation Parameter learning for statistical relational (SRL) (Getoor and Taskar 2007) formalisms such as Markov logic networks (MLNs) (Richardson and Domingos 2006), Bayesian logic programs (BLPs) (Kersting et al 2000), and logical Bayesian networks (LBNs) (Fierens et al 2004), is a widely studied problem (e.g., Venugopal et al (2015); Das et al (2016); Ravkic et al (2015)). Addressing this task requires estimating the frequency of (many) patterns in the data. In this paper, we explore the suitability of using the proposed approaches for parameter estimation in LBNs.

For frequency estimation of a pattern, we report the relative error of a pattern's estimated frequency (using a sampling algorithm) up until time t, denoted as $\#\hat{emb}_i(t)$, versus its true frequency as computed by the exact approach. The relative error is given by the following formula:

$$RelErr_i(t) = \frac{|\#emb_i - \#\hat{emb}_i(t)|}{\#emb_i} \quad (1)$$

Parameter estimation in LBNs requires estimating many conditional probability distributions (CPD) of the form $P(\texttt{Y} \mid \texttt{X} \leftarrow \texttt{Context})$. In these CPDs, `Y` is

a single random variable that can take on multiple values, `X` is a set of random variables and `Context` defines a set of conditions that specifies when the CPD is valid (e.g., there only exists a random variable for a student's grade in a course `C` if she has taken course `C`.).[6] The CPD captures the conditional probability of `Y` for each joint assignment of values to `X`, when `Context` holds in the data. Thus, in a constructed pattern we must specify which nodes represent each of `X`, `Y`, and `Context`. For example, in Figure 4 the shaded nodes represent `X`, `Y` (target) nodes, while other nodes represent `Context`. For example, when creating the patterns for the DBLP dataset, `X`, `Y` are nodes labeled with *citations*. Nodes labeled as $references$, dir, $coauthored$, $citations = low$, $citations = med$, $citations = high$ represent `Context`. For the patterns in the Yeast dataset, `X`, `Y` can be nodes labeled with $function$, $location$, $class$, $enzyme$ or $phenotype$. Each of these has a range of values. Nodes labeled with $interaction$ and $predicate = value$ where $predicate \in \{function,\ location,\ class, enzyme,\ phenotype\}$ and $value \in range(predicate)$ are `Context`. This procedure is similar for the other datasets.

To make the results less susceptible to fluctuations arising from infrequently occurring combinations, we employ the standard Laplace smoothing of our estimates. Furthermore, we aggregate each value combination that has a relative frequency of $< 1\%$ in the Exhaustive approach into a single "default" row. In probabilistic models, this is known as using a default table representation for the local probability distributions (Friedman and Goldzsmidt 1996).

In order to compare the quality of the estimated and the true distribution for a CPD we calculate the Kullback-Leibler divergence (KLD).

$$KLD(P||Q) = \sum_i p(i) \times \log_2 \left(\frac{p(i)}{q(i)} \right) \tag{2}$$

The estimated distribution (P) is obtained by one of the sampling algorithms, and the true distribution (Q) is obtained by the Exhaustive approach.

9.3 Experiments and results for Q1 and Q2

Ultimately, one of the primary goals of sampling is to achieve similar performance at a fraction of the run time required by an Exhaustive approach. We consider two possible ways to vary the run time of the sampling approaches: by comparing them to the Exhaustive approach's and FACT's run times. In both scenarios, we focus on estimating the counts. Hence, we only consider patterns where we know the true count (i.e., those patterns where the Exhaustive approach finished within the time limit). Table 4 provides for each pattern size the percentage of patterns where the Exhaustive approach does not finish. This analysis does not consider the Amazon and Enron datasets as the Exhaustive approach never completes within the time limit. Table 5 shows the average run times for the Exhaustive approach and FACT for each pattern size.

First, we allow each sampling approach to run for a percentage of the Exhaustive approach's total run time. For example, if the Exhaustive approach finished in

[6] Note that this is a slight simplification as LBN uses first-order logic to perform parameter tying across multiple random variables.

Dataset	Pattern Size 4	5	6	7	8	9	10	Average
Yeast (10 minutes)	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
WEBKB	0.0	9.1	30.0	36.7	58.1	77.0	80.0	41.5
IMDB	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Facebook	10.7	47.0	69.0	94.0	96.0	96.0	98.1	73.1
DBLP (10 hours)	0.0	0.0	0.0	0.0	3.0	1.0	10.0	2.0
Amazon	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
Enron	100.0	100.0	100.0	100.0	100.0	99.0	100.0	99.9

Table 4: For each pattern size, the percentage of patterns where the Exhaustive approach failed to finish within the time limit on every dataset.

5 hours for a pattern, then 10% represents estimated results for FK-OBD and FK-AD after 30 minutes. Moreover, we focus on more challenging patterns by omitting any pattern where FK-AD had less than 10% relative error on the estimated number of embeddings in the first interval (that is, the first 5 seconds for YEAST, IMDB, FACEBOOK and WEBKB, and the first 5 minutes for DBLP). Figure 5 shows how the average relative error (averaged over all pattern sizes) varies as a function of run time for the FK-OBD, FK-AD and Random approaches. The plots show that the FK-OBD and FK-AD perform similarly and have a better convergence than Random for all the datasets. Figure 6 shows similar plots but for three specific pattern sizes: 4, 7 and 10.[7] Regardless of the pattern size, both FK-ODB and FK-AD converge very quickly, typically in around 10% of the run time needed by the Exhaustive approach, to a stable estimate. There is a slight decrease in accuracy as the pattern size increases. Finally, regardless of the pattern size, both FK approaches are better than Random, and the gap between them and Random grows as the pattern size increases.

Second, we measure FACT's run time for each pattern and then allow each sampling algorithm (FK-OBD, FK-AD and Random) to run for the same amount of time. Table 6 shows the average relative errors of the sampling approaches and FACT on four real-world datasets. FACT exceeded the amount of available memory[8] on DBLP, Amazon, and Enron. These results also show that FK-AD and FK-OBD have similar performance. On three datasets, they perform better than Random, and on one dataset their performance is equivalent. FK-OBD, FK-AD, and Random are consistently much more accurate than FACT. Table 7 shows the average relative error for each of these four approaches on a per pattern size basis on each dataset for this setting. On the Yeast, WEBKB, and IMDB datasets, FK-ODB and FK-AD result in substantially better performance than FACT, with reductions in the average relative error between 57.5% to 98% on patterns sizes up to length seven. On larger pattern sizes, there are reductions of between 25% and 94%. On Facebook, particularly for pattern sizes nine and ten, the results are closer, but FK-ODB and FK-AD are always better on average. Hence, both FK approaches seems to offer superior performance than FACT for estimating counts.

[7] We omit FACT on these plots to declutter them and because its results would simply be a point.

[8] The experiments were run on a machine with 10Gb of RAM.

Dataset	Method	Pattern Size 4	5	6	7	8	9	10	Average
Yeast	Exhaustive	3.50	5.24	10.10	5.12	10.27	21.54	24.95	11.53
	FACT	16.50	17.77	14.69	14.14	17.45	14.31	17.78	16.09
WEBKB	Exhaustive	1.08	96.32	321.44	338.03	505.39	561.20	565.10	341.22
	FACT	7.00	7.23	7.22	7.37	7.48	7.42	7.83	7.36
IMDB	Exhaustive	0.00	0.03	1.59	15.18	25.60	25.37	16.86	12.09
	FACT	4.34	4.01	3.99	3.99	4.10	4.16	4.01	4.09
Facebook	Exhaustive	69.90	447.36	657.18	614.36	625.59	609.16	608.44	518.86
	FACT	34.00	34.02	35.88	31.30	41.69	27.39	24.91	32.74

Table 5: The average run time in seconds for each pattern size for FACT and the Exhaustive approach.

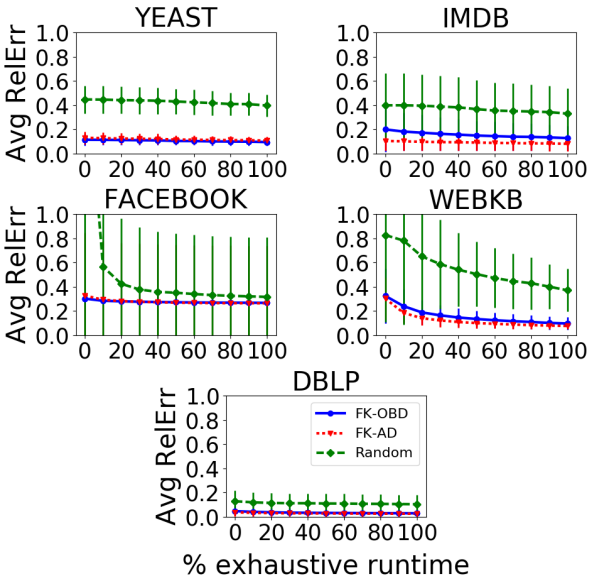

Fig. 5: The average relative error as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD and Random approaches on the Yeast, IMDB, Facebook, WEBKB and DBLP datasets. The results are averaged over all pattern sizes.

Next, we focus on the parameter estimation task. We only consider the Yeast, IMDB, DBLP, Facebook, and WEBKB datasets because Amazon and Enron only contain links (no attributes), so the conditional probability tables would be less interesting. Again, we omit those patterns where the Exhaustive approach failed to finish within the time limit as we lack the ground truth answers for them.

Table 8 presents the KLD averaged over all pattern sizes and all patterns. Here, we see that FK-ODB and FK-AD obtain highly similar results. On the Yeast, Facebook, WEBKB, and DBLP datasets, FK-ODB and FK-AD do substantially

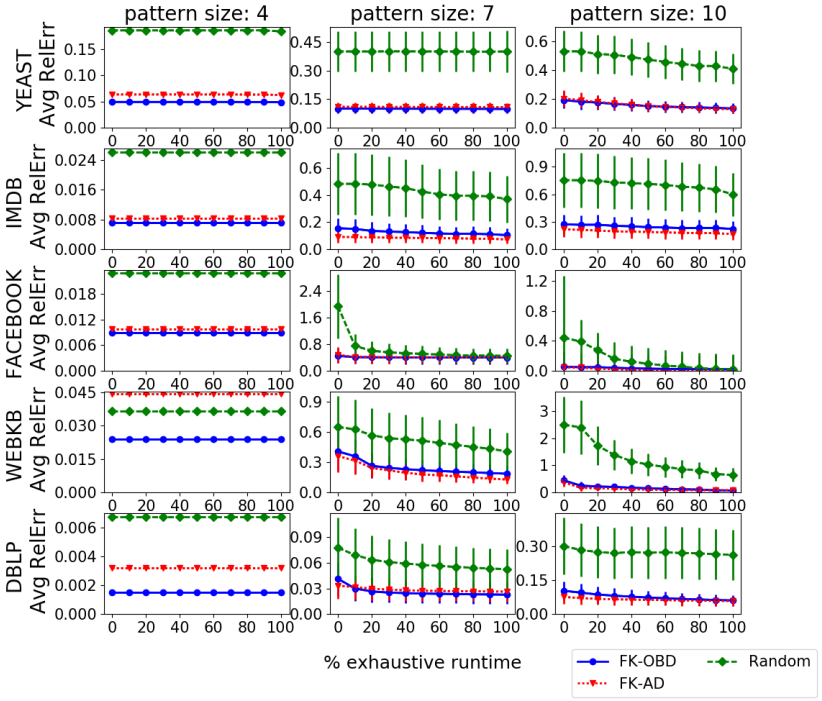

Fig. 6: The average relative error for a specific pattern size in relation to the percentage of the Exhaustive approach's run time for FK-OBD, FK-AD and Random on the Yeast, IMDB, Facebook, WEBKB and DBLP datasets. The plots shows results for pattern sizes 4 (left column), 7 (middle column), and 10 (right column).

Dataset	FK-OBD	FK-AD	FACT	Random
Yeast	**0.048± 0.061**	0.054 ± 0.072	0.917 ± 0.191	0.184 ± 0.190
WEBKB	0.456 ± 0.376	**0.432± 0.382**	0.872 ± 0.245	0.592 ± 0.332
IMDB	0.124 ± 0.011	**0.077± 0.017**	0.861 ± 0.268	0.298 ± 0.061
Facebook	**0.605± 0.375**	0.607 ± 0.376	0.999 ± 0.002	0.606 ± 0.452

Table 6: The average relative error and its standard deviation for the FK-OBD, FK-AD, FACT and Random approaches. The average is over all pattern sizes on the Yeast, WEBKB, IMDB, and Facebook datasets. The best results are shown in bold.

better (i.e., at least a 45% reduction in the KLD) than the Random approach. On the IMDB dataset Random does the best, but all three approaches have similar results. Table 9 shows the average KLD for each pattern size. The KLD is relatively stable for all approaches with increases becoming apparent around pattern size seven or eight. At this size, there is spike that is most pronounced for Yeast, WEBKB, and Facebook. For the latter two datasets, this may be due to the fact that we exclude a large number of longer patterns as we lack ground truth for

Dataset	Method	Pattern Size 4	5	6	7	8	9	10	Average
Yeast	FK-OBD	0.023	0.029	0.046	0.029	0.058	0.067	0.083	0.048
	FK-AD	0.026	0.032	0.047	0.037	0.072	0.081	0.082	0.054
	FACT	0.887	0.942	0.949	0.956	0.956	0.858	0.873	0.917
	Random	0.069	0.153	0.232	0.171	0.194	0.210	0.263	0.184
WEBKB	FK-OBD	0.012	0.128	0.257	0.378	0.512	0.621	0.637	0.456
	FK-AD	0.030	0.145	0.290	0.376	0.430	0.569	0.637	0.432
	FACT	0.953	0.849	0.824	0.890	0.873	0.873	0.873	0.872
	Random	0.036	0.281	0.503	0.576	0.658	0.704	0.630	0.592
IMDB	FK-OBD	0.006	0.011	0.034	0.089	0.218	0.206	0.188	0.124
	FK-AD	0.007	0.014	0.023	0.063	0.098	0.128	0.135	0.077
	FACT	0.825	0.814	0.840	0.831	0.927	0.913	0.838	0.861
	Random	0.018	0.056	0.118	0.275	0.414	0.486	0.458	0.298
Facebook	FK-OBD	0.007	0.209	0.405	0.658	0.792	0.887	0.933	0.606
	FK-AD	0.008	0.208	0.406	0.662	0.794	0.890	0.933	0.607
	FACT	0.996	0.999	0.999	0.999	0.999	0.999	0.999	0.999
	Random	0.014	0.247	0.438	0.673	0.789	0.848	0.901	0.605

Table 7: The average relative error for each pattern size for FK-OBD, FK-AD, FACT and Random on the Yeast, WEBKB, IMDB, and Facebook datasets. FK-OBD, FK-AD, and Random's were permitted the same amount of run time as FACT took for each pattern.

them (i.e., because the Exhaustive approach did not finish in its allotted time). Based on this results, it seems that FK-OBD and FK-AD can be used to learn accurate parameters for a LBN.

Dataset	FK-OBD	FK-AD	Random
Yeast	0.053 ± 0.170	$\mathbf{0.049 \pm 0.165}$	0.283 ± 0.277
Facebook	$\mathbf{0.237 \pm 0.256}$	0.243 ± 0.252	0.331 ± 0.322
WEBKB	$\mathbf{0.088 \pm 0.176}$	0.090 ± 0.189	0.163 ± 0.241
IMDB	0.006 ± 0.027	0.004 ± 0.012	$\mathbf{0.002 \pm 0.006}$
DBLP	0.003 ± 0.023	$\mathbf{0.001 \pm 0.011}$	0.006 ± 0.031

Table 8: The average KLD over all pattern sizes and its standard deviation for the FK-OBD, FK-AD and Random approaches on the Yeast, Facebook, WEBKB, IMDB and DBLP dataset. The best results are shown in bold.

Finally, the Exhaustive approach exceeded the given time limit for all patterns on Amazon and Enron, for a significant number of patterns on WEBKB and Facebook, and some patterns on DBLP. As we lack ground truth for these datasets, we explore the convergence behavior of each sampling approach on these datasets in the following way. We treat each approach's final estimate as the ground truth. Then we show how the approach converges to this estimate as a function of run time. Here, we focus on the average relative error and consider only the patterns were the Exhaustive approach failed to finish in the time limit. Figure 7 and Figure 8 show the results for this experiment. For Amazon, Enron and Facebook,

Dataset	Method	Pattern Size 4	5	6	7	8	9	10	Average
Yeast	FK-OBD	0.011	0.006	0.003	0.004	0.026	0.176	0.181	0.053
	FK-AD	0.011	0.006	0.003	0.004	0.019	0.169	0.176	0.049
	Random	0.033	0.182	0.155	0.346	0.485	0.466	0.559	0.283
WEBKB	FK-OBD	0.001	0.028	0.042	0.139	0.105	0.115	0.159	0.088
	FK-AD	0.001	0.027	0.044	0.070	0.093	0.123	0.167	0.090
	Random	0.001	0.024	0.088	0.158	0.233	0.306	0.311	0.163
IMDB	FK-OBD	0.001	0.001	0.001	0.004	0.009	0.013	0.006	0.006
	FK-AD	0.001	0.001	0.001	0.004	0.009	0.006	0.005	0.004
	Random	0.000	0.000	0.000	0.002	0.004	0.004	0.003	0.002
Facebook	FK-OBD	0.003	0.015	0.106	0.235	0.343	0.508	0.563	0.237
	FK-AD	0.003	0.015	0.118	0.241	0.366	0.448	0.553	0.243
	Random	0.005	0.087	0.312	0.549	0.622	0.656	0.780	0.331
DBLP	FK-OBD	0.000	0.000	0.000	0.000	0.001	0.003	0.008	0.003
	FK-AD	0.000	0.000	0.000	0.000	0.000	0.002	0.004	0.001
	Random	0.000	0.000	0.001	0.000	0.001	0.005	0.022	0.006

Table 9: The average KLD for each pattern size for the FK-ODB, FK-AD, and Random approaches on the Yeast, WEBKB, IMDB, Facebook and DBLP datasets. The entries with all zeros represent numbers that are smaller than 10^{-4}.

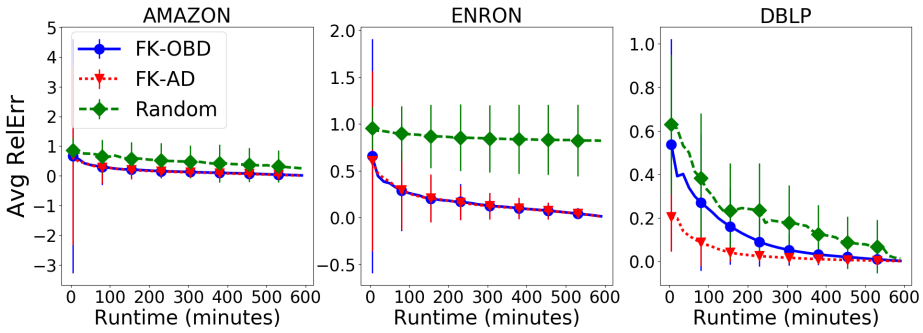

Fig. 7: The average relative error as a function of the run time (in minutes) for the FK-OBD, FK-AD, and Random approach on the Amazon, Enron, and DBLP datasets.

FK-ODB and FK-AD display nearly identical convergence behavior. On the DBLP and WEBKB datasets, FK-AD convergences faster than FK-ODB. They typically converge relatively quickly. Random also converges quickly, but typically with worse estimates than FK-AD and FK-OBD.

9.4 Experiments and Results for Q3

The performance of FK-ODB and FK-AD are very similar. One explanation is that FK-AD uses the OBD of the pattern to construct the AD. To evaluate if this is the case, we construct random arbitrary decompositions in the following way:

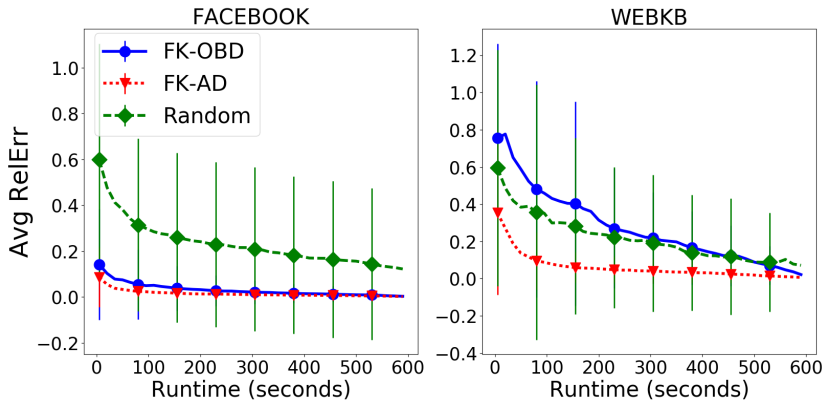

Fig. 8: The average relative error as a function of the run time (in seconds) for the FK-OBD, FK-AD, and Random approach on the Facebook and WEBKB datasets.

Dataset	FK-AD	FK-AD-10
Yeast	0.016 ± 0.027	$\mathbf{0.007 \pm 0.009}$
Facebook	0.048 ± 0.444	$\mathbf{0.046 \pm 0.445}$
WEBKB	0.032 ± 0.077	$\mathbf{0.007 \pm 0.012}$
IMDB	0.027 ± 0.074	$\mathbf{0.022 \pm 0.072}$

Table 10: The average relative error and its standard deviation for the FK-AD and FK-AD-10 on the Yeast, Facebook, WEBKB, and IMDB datasets. The average is over all pattern sizes. The best results are shown in bold.

1. Randomly select a first node in the ordering.
2. Randomly select a node such that it is adjacent to at least one previous node in the existing partial order and add it to the order.
3. Repeat step 2 until all nodes are in the order.

For each pattern, we generate ten such decompositions and compute the counts using each one and report the average over these ten runs. We denote the results for this approach as FK-AD-10. Due to the computational demands of this experiment, we only consider the following four datasets: Yeast, Facebook, WEBKB and IMDB.

Table 10 shows the average relative error for FK-AD and FK-AD-10 averaged over all pattern sizes. The results for both approaches are quite similar, with FK-AD-10 performing slightly better. Figure 9 shows the results for each pattern size. These results show that, regardless of the pattern size, FK-OBD, FK-AD and FK-AD-10 tend to perform similarly. This gives some evidence that the Fürer-Kasiviswanathan approach is robust when a non-OBD decomposition is used to estimate the counts.

Figures 10, 11, 12, and 13 show the average relative error as a function of the percentage of the Exhaustive approach's run time for various pattern sizes on the Facebook, WEBKB, Yeast, and IMDB datasets. Again, these results only consider the patterns where the Exhaustive approach finished within the allotted time so

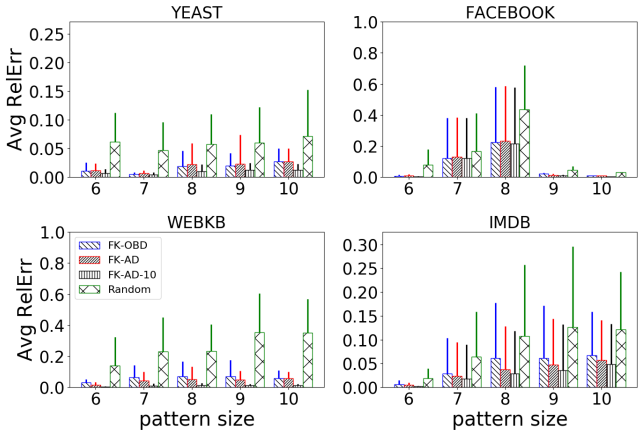

Fig. 9: The average relative error for the FK-OBD, FK-AD, FK-AD-10 and Random approaches on each pattern size for the Yeast, Facebook, WEBKB and IMDB datasets.

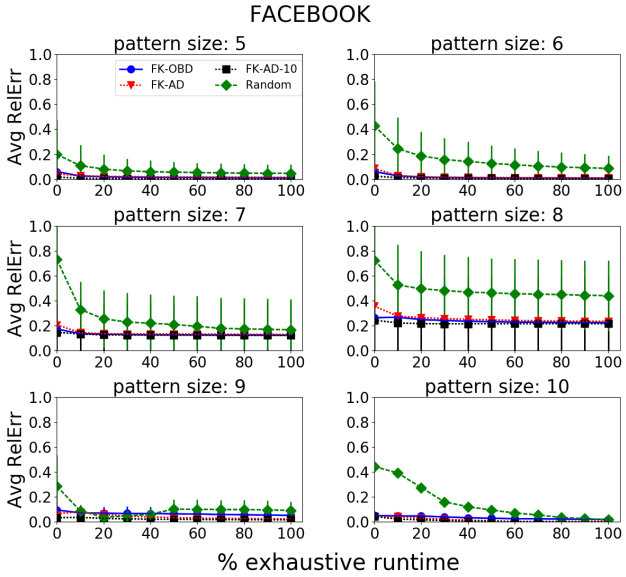

Fig. 10: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, FK-AD-10 and Random approaches on the Facebook dataset.

we know the true count. On all datasets and pattern sizes FK-AD-10 displays a similar convergence behavior to FK-OBD and FK-AD. Like in Table 10, we can see that FK-AD-10 yields similar or slightly better results than FK-OBD and FK-AD.

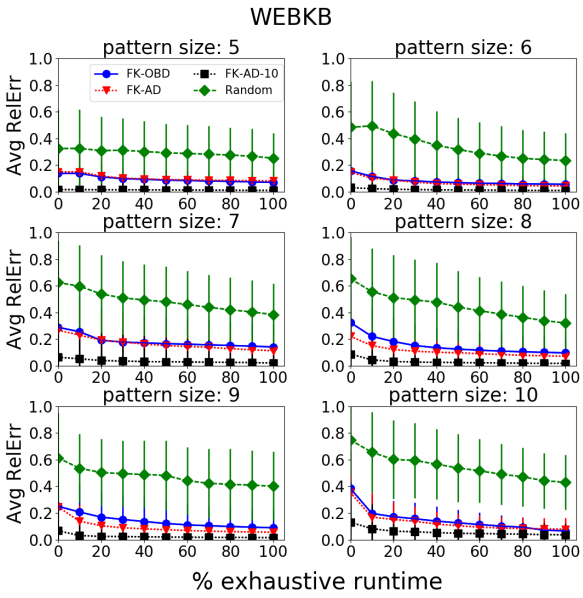

Fig. 11: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, FK-AD-10 and Random approaches on the WEBKB dataset.

9.5 Experiments and Results for Q4

The theoretical analysis of Fürer and Kasiviswanathan only applies to random graphs. To explore in a controlled setting how the algorithm performs when this condition does and does not hold, we use the synthetic datasets consisting of random and power law graphs. Figure 14 and Figure 15 show the performance of each algorithm for the KLD and average relative error, respectively. On these datasets, the Exhaustive approach always finished within the time limit. On average, FK-OBD has a better KLD and smaller relative errors than both FK-AD and Random. Figure 16 compares the average relative error for FK-OBD and FACT. FACT performs much worse than all the other methods on these datasets. These results provide some evidence that FK-OBD and FK-AD will still perform well even on non-random graphs where the theoretical analysis does not hold.

9.6 Experiments and Results for Q5

Next, we evaluate our approach in the context of weight learning and inference. We consider hand-crafted models for the WEBKB and Yeast datasets. In the WEBKB dataset, the goal is to predict label of a web page, which is a standard task. The model consists of four hand-crafted patterns. In the Yeast dataset, we predict the location of a protein. For this problem, the model consists of five hand-crafted patterns. In both models, the patterns were derived from first-order logical formulas where the target predicate (PageClass and Location) appeared exactly once. We

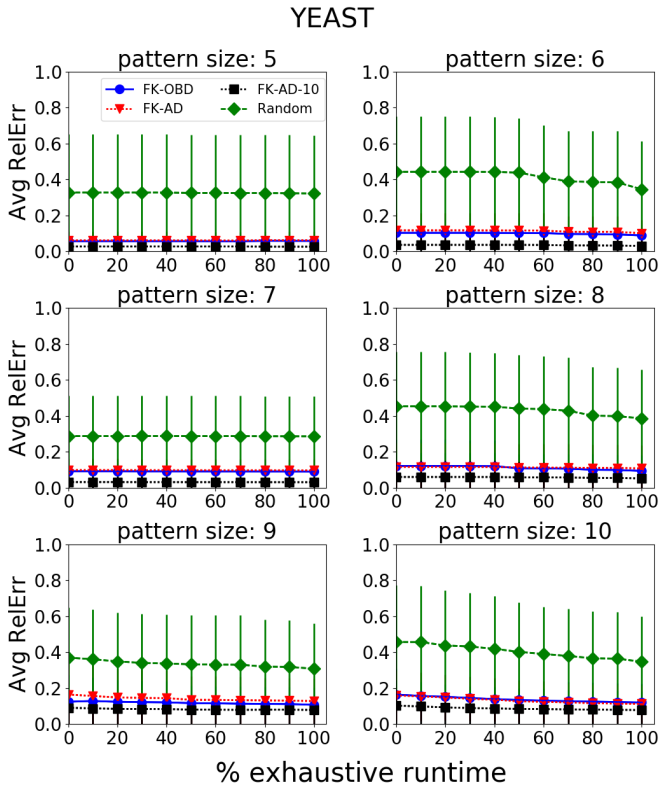

Fig. 12: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, FK-AD-10 and Random approaches on the Yeast dataset.

train a logistic regression model whose features are the counts associated with each pattern to predict the truth value of each grounding of the target predicate. Our models can be viewed as being analogous to an MLN because a discriminatively trained MLN model that only contains non-recursive clauses (which corresponds to our setting) is equivalent to a logistic regression model (Huynh and Mooney 2008).

In the WEBKB dataset, we restrict ourselves to considering 50 words (Davis and Domingos 2009). Both the WEBKB and Yeast datasets are divided into four folds, and hence we perform four-fold cross validation. Both weight learning (i.e., training the model) and inference (i.e., making the predictions on the test set) require computing counts. We compare the predictive and run time performance of using the Exhaustive approach to obtain the exact counts for both learning and inference versus using FK-OBD to obtain the approximate counts for both learning and inference. For FK-OBD, we consider the count obtained for a pattern using four different time limits: 1 second (FK-OBD-1s), 10 seconds (FK-ODB-10s), 1 minute (FK-OBD-1m), and 5 minutes (FK-OBD-5m).

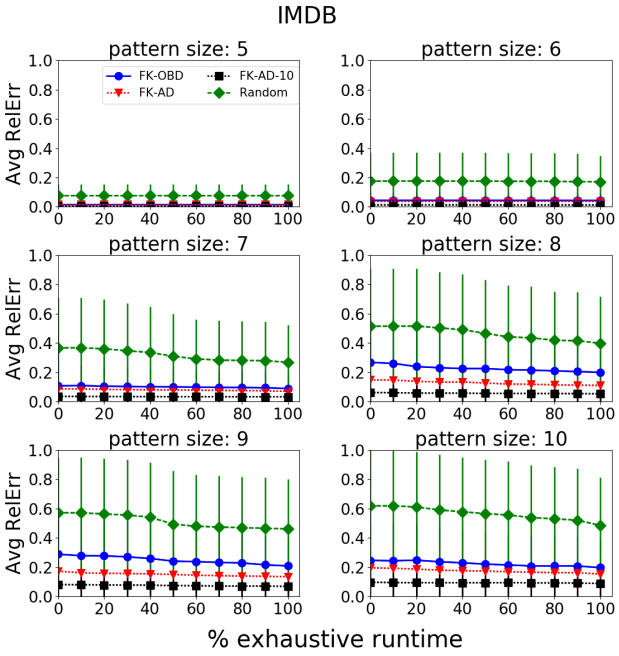

Fig. 13: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, FK-AD-10 and Random approaches on the IMDB dataset.

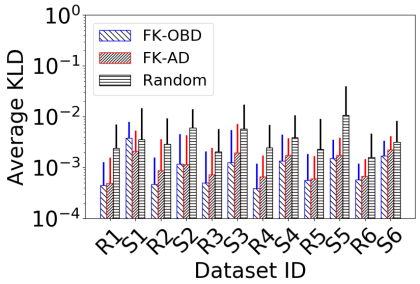

Fig. 14: The KLD averaged over all sampling iterations on the synthetic datasets. The datasets are sorted by increasing graph density. The error bars represent the standard deviation.

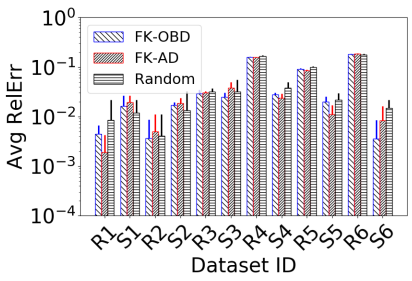

Fig. 15: The relative error averaged over all iterations of the sampling algorithms on the synthetic datasets. The datasets are sorted by increasing graph density. The error bars represent the standard deviation.

Table 11 shows the area under the ROC curve (AUC-ROC) for all approaches together with the speedup factor of the FK-ODB approaches over the Exhaustive approach for the counting time needed for learning and inference. On both datasets, the sweet spot seems to be running FK-ODB for 1 minute. There is essentially no loss of accuracy, and it achieves a 100 time speedup on WEBKB and a 10 time speedup on Yeast. Running FK-OBD for a longer time period only marginal improves predictive performance. Similarly, running FK-OBD for 10 second results in a small decline in predictive performance on WEBKB and a larger one on Yeast, albeit with a large saving in run time cost. Unsurprisingly, on these datasets, running FK-ODB for 1 second degrades the predictive performance substantially.

Dataset	Metrics	Exhaustive	FK-ODB-1s	FK-ODB-10s	FK-ODB-1m	FK-ODB-5m
WEBKB	Speedup Factor: Learning	N/A	12916.7	1291.7	215.3	43.1
	Speedup Factor: Inference	N/A	9642.6	964.3	160.7	32.1
	AUC-ROC	0.99± 0.014	0.74±0.033	0.93±0.028	0.97±0.015	0.98±0.013
Yeast	Speedup Factor: Learning	N/A	585.0	58.5	9.8	2.2
	Speedup Factor: Inference	N/A	585.7	58.6	9.8	2.1
	AUC-ROC	1.00±0.000	0.64±0.005	0.83±0.010	0.97±0.006	0.99±0.005

Table 11: The average AUC-ROC and its standard deviation when the counts required for learning and inference were obtained by the Exhaustive approach and FK-ODB with a time limit of 1 second (FK-OBD-1s), 10 seconds (FK-ODB-10s), 1 minute (FK-OBD-1m), and 5 minutes (FK-OBD-5m). Additionally, the speedup factor of the FK-ODB approaches over the Exhaustive approach for the counting time needed during both learning and inference is shown. The speedup factor for the Exhaustive approach over itself is not applicable (N/A).

9.7 Experiments and Results for Q6

Finally, we explore how the approach will handle longer patterns. Employing the same pattern generation and selection approach outlined in Subsection 9.2.1, we generate patterns containing from 11 up to 15 nodes for the DBLP and Yeast datasets. Here, we only focus on the FK-ODB and FK-AD approaches.

Figures 17 and 18 present results for this setting. Each plot shows the average relative error of FK-ODB and FK-AD when those approaches are given a fixed percentage of the run time required for the Exhaustive approach to complete. On DBLP, regardless of the pattern size, both approaches arrive at a relatively stable estimate using only a small fraction of the time needed by the Exhaustive approach. Given more time, the estimates improve slightly. On Yeast, longer pattern sizes require slightly more time to converge. On this dataset, there is not much improvement when running beyond 20 to 40 percent of the Exhaustive approach's run time. On both datasets, FK-OBD and FK-AD perform similarly.

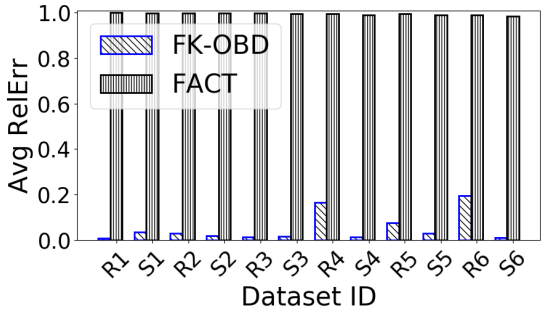

Fig. 16: The average relative error for FK-OBD and FACT on the synthetic datasets. The datasets are sorted by increasing graph density.

9.8 Discussion

Now we will revisit the experimental questions posed at the beginning of this section in light of the presented results. In terms of **Q1**, across the various datasets and pattern sizes considered, we see that the FK-OBD and FK-AD approaches result in much more accurate counts than Random and FACT when each algorithm has the same run time. Furthermore, both FK-ODB and FK-AD converge rather quickly to counts that are close to the true count found by Exhaustive search. Finally, we see that FK-OBD and FK-AD are able to learn highly accurate CPDs for LBNs on almost all datasets and pattern sizes.

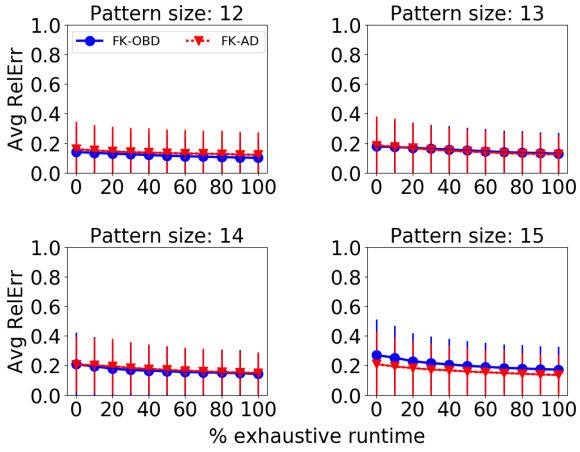

Fig. 17: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, and Random approaches on the DBLP dataset. The focus is longer pattern sizes.

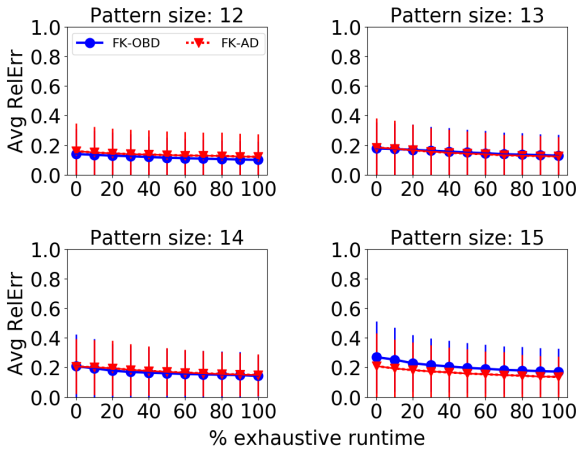

Fig. 18: The average relative error for a specific pattern size as a function of the percentage of the Exhaustive approach's run time for the FK-OBD, FK-AD, and Random approaches on the YEAST dataset. The focus is longer pattern sizes.

9.9 Discussion

Finally, we revisit the experimental questions posed at the beginning of this section in light of the presented results. In terms of **Q1**, across the various datasets and pattern sizes considered, we see that the FK-OBD and FK-AD approaches result in much more accurate counts than Random and FACT when each algorithm has the same run time. Furthermore, both FK-ODB and FK-AD converge rather quickly to counts that are close to the true count found by Exhaustive search.

For **Q2**, we can see that FK-OBD and FK-AD tend to perform similarly. Thus even when the decomposition is not an OBD the approach seems to converge in practice. One possible reason is that because the size of the AD is larger, it is faster to match. Hence, it makes more observations than FK-OBD which needs to calculate all possible combinations of values for nodes in each partition of an OBD. The larger the partition, the longer it takes to finish one iteration of sampling. This result holds for both approaches we used to construct an arbitrary decomposition (**Q3**).

The synthetic data allow us to address **Q4**. Here, we find that FK-OBD and FK-AD perform well on non-random graphs where the theoretical guarantees of the original Fürer-Kasiviswanathan algorithm are not valid. These results, in conjunction with our findings on the real-world data, indicate that FK is a viable strategy for non-random graphs.

To address **Q5**, we considered using FK-OBD to obtain the counts needed for learning and inference. We see that FK-ODB achieves significant gains in run time performance for both tasks. These gains, which are between one and two orders of magnitude, come only with a very small drop in predictive performance.

By looking at results on patterns up to length 15, we see that the behavior of FK-ODB and FK-AD are consistent with how they perform on shorter patterns. These results thus provide some evidence for a positive answer to **Q6**.

Finally, we hope that this work can benefit and be exploited by the broader community. Therefore, we have released our code and the patterns used.[9]

10 Conclusions

This paper explored the problem of estimating the number of embeddings of a pattern in a graph, which is an important task in many areas like (approximate) pattern mining and parameter estimation for statistical relational models. Specifically, the paper presented several different sampling algorithms, studied how to obtain statistics from the embeddings they generated, and performed an extensive empirical comparison.

Empirically, we observed that a strategy based on the theoretical algorithm by Fürer and Kasiviswanathan (2014) performed well, even in conditions where the requirements for its theoretical guarantees were violated. Namely, we found that the algorithm still exhibited good performance when given (1) a decomposition that is not a OBD, and (2) a data graph that is not an Erdős-Rényi random graph. We found that the FK-based approaches offered several benefits. One, they outperformed several competitors. Two, they converged to a good estimate in the early stages of sampling. Three, they were able to produce accurate estimates for longer patterns.

[9] https://dtai.cs.kuleuven.be/software/gs-srl

Acknowledgments

IR was supported by the KU Leuven Research Fund (OT/11/051). MZ was partially supported by the KU Leuven Research Fund (OT/11/051) and the Slovenian Research Agency (P2-0103). JD is partially supported by the KU Leuven Research Fund (OT/11/051, C14/17/070, C22/15/015, C32/17/036) and FWO-Vlaanderen (G.0356.12, SBO-150033).

References

Ariely D (2008) Predictably irrational: The hidden forces that shape our decisions. Harper Collins

Barabasi AL, Albert R (1999) Emergence of scaling in random networks. Science 286(5439):509–512

Baskerville K, Grassberger P, Paczuski M (2007) Graph animals, subgraph sampling, and motif search in large networks. Physical Review E 76(3):036,107

Bordino I, Donato D, Gionis A, Leonardi S (2008) Mining large networks with subgraph counting. In: Proceedings of the 2008 IEEE International Conference on Data Mining (ICDM), pp 737–742

Cordella L, Foggia P, Sansone C, Vento M (2004) A (sub)graph isomorphism algorithm for matching large graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence 26(10):1367–1372

Das M, Wu Y, Khot T, Kersting K, Natarajan S (2016) Scaling lifted probabilistic inference and learning via graph databases. In: Proceedings of the 2016 SIAM International Conference on Data Mining (SDM), pp 738–746

Davis J, Domingos P (2009) Deep transfer via second-order Markov logic. In: Proceedings of the Twenty-Sixth International Conference on Machine Learning (ICML), pp 217–224

Davis J, Burnside E, Dutra IC, Page D, Costa VS (2005) An integrated approach to learning Bayesian networks of rules. In: Proceedings of the Sixteenth European Conference on Machine Learning (ECML), pp 84–95

Di Natale R, Ferro A, Giugno R, Mongiovi M, Pulvirenti A, Shasha D (2010) SING: Subgraph search in non-homogeneous graphs. BMC Bioinformatics 11(1):96

Fierens D, Blockeel H, Ramon J, Bruynooghe M (2004) Logical Bayesian networks. In: Proceedings of the Third International Workshop on Multi-Relational Data Mining (MRDM), pp 19–30

Friedman N, Goldzsmidt M (1996) Learning Bayesian networks with local structure. In: Proceedings of the Twelfth Annual Conference on Uncertainty in Artificial Intelligence (UAI), pp 252–262

Fürer M, Kasiviswanathan SP (2014) Approximately counting embeddings into random graphs. Combinatorics, Probability & Computing 23(6):1028–1056

Getoor L, Taskar B (2007) Introduction to statistical relational learning. MIT press

Giugno R, Shasha D (2002) GraphGrep: A fast and universal method for querying graphs. In: Proceedings of the Sixteenth International Conference on Pattern Recognition (ICPR), pp 112–115

Huynh T, Mooney R (2008) Discriminative structure and parameter learning for Markov logic networks. In: Proceedings of the Twenty-Fifth International Conference on Machine Learning, pp 416–423

Inokuchi A, Washio T, Motoda H (2003) Complete mining of frequent patterns from graphs: Mining graph data. Machine Learning 50(3):321–354

Jowhari H, Ghodsi M (2005) New streaming algorithms for counting triangles in graphs. In: Proceedings of the Eleventh International Conference on Computing and Combinatorics (COCOON), pp 710–716

Kashtan N, Itzkovitz S, Milo R, Alon U (2004) Efficient sampling algorithm for estimating subgraph concentrations and detecting network motifs. Bioinformatics 20(11):1746–1758

Kersting K, De Raedt L, Kramer S (2000) Interpreting Bayesian logic programs. In: Proceedings of the AAAI-2000 Workshop on Learning Statistical Models from Relational Data, pp 29–35

Kok S, Domingos P (2005) Learning the structure of Markov logic networks. In: Proceedings of the Twenty-Second International Conference on Machine Learning (ICML), pp 441–448
Leskovec J, Faloutsos C (2006) Sampling from large graphs. In: Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp 631–636
Mewes HW, Frishman D, Gruber C, Geier B, Haase D, Kaps A, Lemcke K, Mannhaupt G, Pfeiffer F, Schüller C, Stocker S, Weil B (2000) MIPS: A database for genomes and protein sequences. Nucleic Acids Research 28(1):37–40
Pržulj N (2007) Biological network comparison using graphlet degree distribution. Bioinformatics 23(2):177–183
Ravkic I, Ramon J, Davis J (2015) Learning relational dependency networks in hybrid domains. Machine Learning 100(2-3):217–254
Richards BL, Mooney RJ (1992) Learning relations by pathfinding. In: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI), pp 50–55
Richardson M, Domingos P (2006) Markov logic networks. Machine learning 62(1-2):107–136
Shervashidze N, Vishwanathan S, Petri T, Mehlhorn K, Borgwardt K (2009) Efficient graphlet kernels for large graph comparison. In: Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS), pp 488–495
Tang J, Zhang J, Yao L, Li J, Zhang L, Su Z (2008) Arnetminer: Extraction and mining of academic social networks. In: Proceedings of the Fourteenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), pp 990–998
Ullmann JR (1976) An algorithm for subgraph isomorphism. Journal of the ACM 23(1):31–42
Van Haaren J, Kolobov A, Davis J (2015) TODTLER: Two-order-deep transfer learning. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, pp 3007–3015
Venugopal D, Sarkhel S, Gogate V (2015) Just count the satisfied groundings: Scalable local-search and sampling based inference in MLNs. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, pp 3606–3612
Wernicke S (2005) A faster algorithm for detecting network motifs. In: Proceedings of the Fifth International Workshop on Algorithms in Bioinformatics (WABI), pp 165–177
Yan X, Han J (2002) gSpan: Graph-based substructure pattern mining. In: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM), pp 721–724
Zou R, Holder LB (2010) Frequent subgraph mining on a single large graph using sampling techniques. In: Proceedings of the Eighth Workshop on Mining and Learning with Graphs (MLG), pp 171–178