HoPS: Probabilistic Subtree Mining for Small and Large Graphs

Pascal Welke
University of Bonn
Bonn, Germany
welke@cs.uni-bonn.de

Florian Seiffarth
University of Bonn
Bonn, Germany
seiffarth@cs.uni-bonn.de

Michael Kamp
Monash University
Melbourne, Australia
michael.kamp@monash.edu

Stefan Wrobel
University of Bonn and Fraunhofer IAIS
Bonn, Germany
wrobel@cs.uni-bonn.de

ABSTRACT

Frequent subgraph mining, i.e., the identification of relevant patterns in graph databases, is a well-known data mining problem with high practical relevance, since next to summarizing the data, the resulting patterns can also be used to define powerful domain-specific similarity functions for prediction. In recent years, significant progress has been made towards subgraph mining algorithms that scale to complex graphs by focusing on tree patterns and probabilistically allowing a small amount of incompleteness in the result. Nonetheless, the complexity of the pattern matching component used for deciding subtree isomorphism on arbitrary graphs has significantly limited the scalability of existing approaches. In this paper, we adapt sampling techniques from mathematical combinatorics to the problem of probabilistic subtree mining in arbitrary databases of many small to medium-size graphs or a single large graph. By restricting on tree patterns, we provide an algorithm that approximately counts or decides subtree isomorphism for arbitrary transaction graphs in sub-linear time with one-sided error. Our empirical evaluation on a range of benchmark graph datasets shows that the novel algorithm substantially outperforms state-of-the-art approaches both in the task of approximate counting of embeddings in single large graphs and in probabilistic frequent subtree mining in large databases of small to medium sized graphs.

CCS CONCEPTS

• **Mathematics of computing** → **Trees**; **Matchings and factors**; **Graph algorithms**; • **Information systems** → **Data mining**; • **Computing methodologies** → **Randomized search**; **Motif discovery**; • **Theory of computation** → **Pattern matching**.

ACM Reference Format:
Pascal Welke, Florian Seiffarth, Michael Kamp, and Stefan Wrobel. 2020. HoPS: Probabilistic Subtree Mining for Small and Large Graphs. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3394486.3403180

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08…$15.00
https://doi.org/10.1145/3394486.3403180

1 INTRODUCTION

Frequent subgraph mining is a well known technique to identify relevant patterns in graph databases. The resulting patterns can be used to define powerful domain-specific similarity functions. One common way of defining relevance is based on frequency [5] of occurrence wrt. some matching criterion. Most commonly, subgraph isomorphism or induced subgraph isomorphism are used as matching criterions, as they are intuitive and have been shown to be semantically more meaningful than, e.g., homomorphism [16]. By identifying the patterns in the mining process, frequent subgraph mining can be interpreted as a form of representation learning where each pattern represents a meaningful substructure in the input graph, which often is relevant for further analysis. Furthermore, identifying the frequency of known subgraphs up to a certain size allows to efficiently compute similarity measures between graphs which is essential in many machine learning algorithms [2].

In this work, we are interested in counting subgraph isomorphisms (i.e., not induced subgraph isomorphisms) of arbitrary size, which is a basic building block for both representation learning and frequency estimation. In particular, we are interested in a method that works both for the transactional setting, where the input is a database of many small to medium sized graphs, and for the setting where the input is one large graph. Both tasks – unfortunately – are computationally intractable [4, 6]. However, previous work has shown a theoretically and practically feasible mining system that restricts the pattern language to *trees* and allows the output to be incomplete, i.e., allows dropping a few frequent subtrees [19].

We propose a novel embedding algorithm for trees, denoted HoPS (Highly Optimistic Probabilistic Subtrees), which is named after the vine, as the embeddings grow greedily from the embedding of the root until no progress can be made. Importance sampling similar to Fürer and Kasiviswanathan [3] – and its extension to labeled graphs by Ravkic et al. [14] – provides a fast and accurate algorithm for estimating the number of subgraph isomorphisms in a large graph. Our algorithmic contribution is a linear time algorithm for sampling and counting maximum matchings arising as subproblems in each step of the algorithm, reducing the runtime from exponential to linear in the pattern degree.

Note that counting *induced* subgraph isomorphisms, is known as graphlet counting or sampling and a large number of works have addressed practically fast methods for this purpose, see [15] for a recent survey. Here, importance sampling[1] has been used as

[1] This is also known as Horvitz-Thompson estimation.

well [13]. The difference is that our algorithm considers trees of arbitrary size in labeled graphs. Moreover, the current state-of-the-art in graphlet sampling considers graphlets of up to eight vertices [1, 13], while we consider patterns of arbitrary size, evaluating our algorithm on patterns with up to 50 vertices. Koutis and Williams [10] present a randomized algorithm that approximately decides (with one-sided error) subgraph isomorphism in $O\left(m2^k k^2 \log^2 k\right)$ time, where m is the number of edges in the transaction graph and k is the number of vertices in the pattern tree. While this work was used by to identify frequent subtrees in a large graph [8], it does not fulfill our scaling requirements. The runtime of HOPS instead depends only linearly on the pattern size and the *vertex degree* of the transaction graph, not on its overall size, making it highly scalable.

HOPS can also be used to answer the simpler question of whether a pattern is present in a transaction graph or not and thus can be used inside a generic frequent subgraph mining algorithm to perform practically feasible and theoretically efficient incomplete frequent subtree mining for arbitrary transactional graph databases. Our empirical evaluation shows that this algorithm substantially outperforms the state of the art on a range of benchmark graph datasets. Our contributions are as follows:

(1) We offer a drop-in replacement for the embedding algorithm considered by Fürer and Kasiviswanathan [3] and Ravkic et al. [14] for the case that the patterns are trees. That is: Using our algorithm, the theoretical guarantees of [3] hold if the pattern vertex degree is bounded, and it can be easily integrated in the practical system of [14].
(2) Our algorithm runs in linear time in the size of the pattern (and the maximum degree of the transaction graphs) and does not restrict the vertex degree of the pattern or transaction graphs. As a result, it improves the runtime of the algorithms in [3, 14] by a factor that is exponential in the maximum vertex degree of the pattern. In particular, it does not require a maximum matching algorithm as subroutine, rendering it fast and easy to implement.
(3) We empirically evaluate our algorithm for two common graph mining settings: Similar to Ravkic et al. [14], we consider approximate counting of embeddings in a single large transaction graph and similar to [20], we consider probabilistic frequent subtree mining in databases of many small to medium sized graphs. We show that our algorithm substantially outperforms its competitors in both settings.

2 BASIC NOTIONS

An *(undirected) graph* $G = (V, E)$ consists of a finite set V of vertices and a set $E \subseteq \{X \subseteq V : |X| = 2\}$. An edge $\{u, v\} \in E(G)$ will be denoted by uv. Given $G = (V, E)$ we will often use the notation $V(G) := V$ and $E(G) := E$ to refer to the vertex or edge set of G. We consider only simple graphs, i.e., loops and parallel edges are not permitted. A *labeled* graph is a graph G together with a function $l_G : E(G) \cup V(G) \to L$ that assigns a label from some finite set L to each vertex and each edge. Labeled graphs can model chemical molecules, protein-protein interactions, social networks, the Web graph and other phenomena. A tree is a graph H that is connected and $|E(H)| = |V(H)| - 1$.

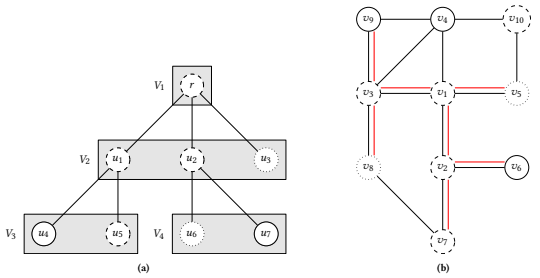

Figure 1: Rooted and labeled (different labels are indicated by the node border style) tree H with root r and OBD $V_1, \ldots, V_4$ (left). Labeled graph G with tree isomorphism indicated in red where the root node r is mapped to v_1 (right).

The set $\mathcal{N}(v) := \{w \in V(G) : vw \in E(G)\}$ is the set of *neighbors* of v. The cardinality of $\mathcal{N}(v)$ is called *degree* of v and denoted by $\delta(v)$. We denote $\Delta(G) := \max_{v \in V(G)} \delta(v)$ and $\varnothing\delta(G) = \frac{1}{|V(G)|} \sum_{v \in V(G)} \delta(v)$. A *leaf* is a vertex that has exactly one neighbor. For $V' \subseteq V$ we define $\mathcal{N}(V') = \bigcup_{v \in V'} \mathcal{N}(v) \setminus V'$.

An *independent set* S in a graph G is a subset $S \subseteq V(G)$ such that there are no edges between any $v, w \in S$. A *bipartite graph* $G = (A \dot\cup B, E)$ is a graph such that A and B are independent sets. In a *complete* bipartite graph, each vertex in A is connected to each vertex in B; hence it has $|A| \cdot |B|$ edges.

A labeled graph $H = (V(H), E(H), l_H)$ is subgraph isomorphic to a labeled graph $G = (V(G), E(G), l_G)$ if there exists an injective mapping $\varphi : V(H) \to V(G)$ such that $vw \in E(H)$ implies $\varphi(v)\varphi(w) \in E(G)$ and for all nodes $v \in V(H)$ and for all edges $vw \in E(H)$ it holds that $l_H(v) = l_G(\varphi(v))$ and $l_H(vw) = l_G(\varphi(v)\varphi(w))$. We will often call H the *pattern* graph, and G the *transaction* graph.

A *matching* in a graph G is a set $M \subseteq E(G)$ such that each vertex $v \in V(G)$ is contained in at most one edge $e \in M$. A *maximum matching* is a matching $M \subseteq E(G)$ such that all $M' \subseteq E(G)$ with $|M| < |M'|$ are not a matching. A matching $M \subseteq E$ *covers* a set of vertices $A \subseteq V(G)$, if every vertex $a \in A$ appears in some edge $e \in M$. We note that a matching M in a bipartite graph $(A \dot\cup B, E)$ that covers A can be seen as an injective mapping $\varphi : A \to B$ such that $\varphi(a) = b$ for all $ab \in M$. We will use this property frequently to extend partial subgraph isomorphisms with compatible matchings. Finally, let $[n] := \{1, \ldots, n\}$.

3 APPROXIMATE COUNTING OF SUBGRAPH ISOMORPHISMS

We first discuss the approximation algorithm proposed by Fürer and Kasiviswanathan [3] before presenting our improvements in Section 4. Since we are interested in the theoretical and *practical* aspects of the algorithm, we explain the details of an efficient implementation and analyze its runtime thoroughly. For the first time we give a tight worst case analysis of its runtime. We will subsequently present our improved HOPS algorithm in Section 4.

The algorithm estimates the number of subgraph isomorphisms from a pattern graph H to a transaction graph G using importance sampling. It requires an *ordered bipartite decomposition (OBD)* of H and can be shown to be computationally efficient if this OBD has

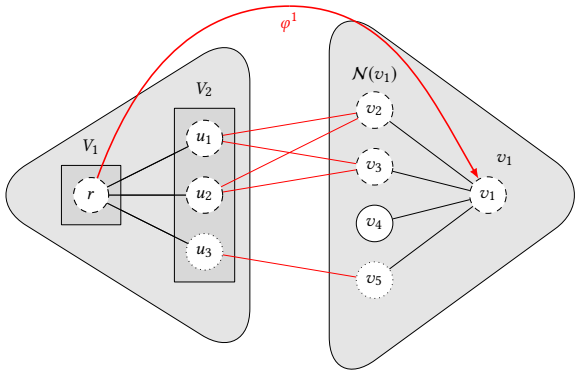

Figure 2: A bipartite matching instance created in Line 8 of Algorithm 1. Note that this instance (due to the labels) consists of three disjoint complete bipartite blocks.

bounded *width*. Furthermore, they show that this method results in a *fully polynomial randomized approximation scheme (FPRAS)* for *almost all* transaction graphs. While this result is more of theoretical interest[2], the algorithm works well in practice [14]. It is based on the simple idea that if we pick a random embedding for a pattern H with probability $p_i > 0$ from the set of all embeddings $C_H(G)$ into G and calculate p_i accurately, then the output $Z = 1/p_i$ is in expectation the number of embeddings, i.e., $\mathbb{E}[Z] = \sum_{i \in C_H(G)} p_i 1/p_i = |C_H(G)|$. The challenging part is to calculate the embedding probability of the pattern. For that, we start by introducing the decomposition that is required for the pattern.

Definition 3.1 ([3]). An *ordered bipartite decomposition* of a graph $G = (V, E)$ is a sequence $D(G) = V_1, \ldots, V_l$ of subsets of V such that

(1) $V_1, \ldots, V_l$ form a partition of V,
(2) each V_i is an independent set in G, and
(3) $\forall v \in V\ \exists k \in [l]$ such that $v \in V_i \Rightarrow \mathcal{N}(v) \subseteq \left(\bigcup_{j=1}^{i-1} V_j\right) \cup V_k$.

Several types of graphs are guaranteed to have an OBD, e.g. trees and bipartite graphs [3]. See Fig. 1a and Fig. 2 for examples. However, not all graphs have an OBD, the simplest case being the triangle graph: Property 2 requires each set in the partition to contain only a single vertex, but then Property 3 cannot be fulfilled, as each vertex has two neighbors, which are in different V_i's. Hence any ordering of the V_i's dissatisfies this condition for the first singleton in the order. Due to this fact Ravkic et al. [14] investigated whether the algorithm still works in practice for *arbitrary decompositions (AD)* that do not fulfill Property 3. Fürer and Kasiviswanathan [3] require an OBD to bound the variance of the estimator to obtain a FPRAS, but an AD suffices for the estimate to be unbiased. However, as we will show in Section 6, the choice of decomposition also has practical implications for the quality of results.

Algorithm 1 shows the pseudo code of the algorithm, relying on subroutines that we will describe below. It first computes an OBD/AD $D(H)$ of the pattern H (Line 2) and then goes into a loop

[2] It is practically infeasible to derive the required parameters for a given pair of graphs and approximation factor.

Algorithm 1 The Algorithm of Fürer and Kasiviswanathan

input: A graph H, a graph G, and an integer $k > 0$
output: An unbiased estimate of the number of subgraph isomorphisms φ from H to G.

```
1:  C := ∅
2:  D(H) := ComputeDecomposition(H)
3:  for all 1, ..., k do
4:      for all v ∈ V(H) do φ(v) := None
5:      φ(V_1), c' := DrawInitialAssignment(V_1, G)
6:      c := c'
7:      for i = 2, ..., |D(H)| do
8:          (AU̇B, E_i) := CreateMatchingInstance(V_i, G)
9:          M, c' := DrawAndCountMatchings(AU̇B, E_i))
10:         if |M| = |A| then
11:             for all (a, b) ∈ M do φ(a) = b
12:             c := c · c'
13:         else      // we could not find a subgraph isomorphism
14:             c := 0
15:             Continue in Line 16
16:     C := C ∪ {c}
17: return (1/k) Σ_{c∈C} c
```

for the importance sampling (Line 3). For this, it tries repeatedly to incrementally build a subgraph isomorphism φ from H into G starting with the first partition set V_1 of $D(H)$. Each time, it computes the inverse probability $c(\varphi)$ of finding exactly this embedding by the applied method, or zero if the method failed (cf. Lines 12 and 14). In this way, Alg. 1 returns an unbiased estimate of the number of subgraph isomorphisms from H to G: Each subgraph isomorphism φ can be found with nonzero probability $\frac{1}{c(\varphi)}$. Hence each subgraph isomorphism contributes a value of one to the expectation of the estimator. Repeating the sampling step i.i.d. for k iterations, the algorithm returns the average of the individual estimates to reduce the variance (Line 17).

To compute φ and $c(\varphi)$ for an iteration k, the algorithm greedily selects feasible injective assignments $\varphi_1, \ldots, \varphi_l$ for the partition sets $V_1, \ldots, V_l$ in the OBD/AD $D(H)$ without ever changing a decision that was made and sets $\varphi = \bigcup_{i=1}^{l} \varphi_i : V(H) \to V(G)$. Hence $\frac{1}{c(\varphi)}$ is the probability of choosing the assignment of the first partition set among all feasible assignments, multiplied by the probability of choosing the assignment of the second set among all feasible mappings given the first choice, and so on. A *feasible assignment of* V_i, is an assignment $\varphi_i : V_i \to V(G)$ such that $\varphi^i = \bigcup_{j=1}^{i} \varphi_j : \bigcup_{j=1}^{i} V_j \to V(G)$ is a subgraph isomorphism and φ_j for $j < i$ are feasible assignments selected in previous steps.

DrawInitialAssignment (Line 4) draws a feasible assignment for V_1; CreateMatchingInstance (Line 8) addresses the other V_i's. Recall that all V_i are independent sets. Hence, a random injective assignment $\varphi_1 : V_1 \to V(G)$ is feasible if for all $u \in V_1$ it holds $l(u) = l(\varphi_1(u))$. As a result, DrawInitialAssignment (Line 4) can just randomly select an initial assignment by drawing vertices without replacement from multiple bags of suitably labeled vertices.

For all V_i with $i > 1$, the feasibility of an assignment $\varphi_i : V_i \to V(G)$ does not only depend on V_i, but also on φ^{i-1}. For an extended

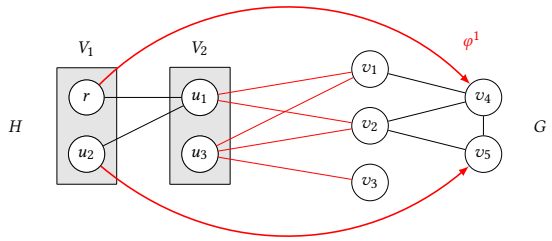

Figure 3: A pattern graph H with OBD $\{V_1, V_2\}$, transaction graph G, partial embedding $\varphi^1 : V_1 \to G$ (red arrows) and matching instance for finding a feasible φ_2.

φ^i to remain a subgraph isomorphism, we have to maintain (1) injectivity, (2) for each edge $uu' \in E(H)$ with $u \in V_i$ and $u' \in \bigcup_{j=1}^{i-1} V_j$ there must be an edge $\varphi_i(u)\varphi^{i-1}(u') \in V(G)$, and (3) compatible labels of vertices (resp. edges) and their images under φ^i.[3] We can solve this by finding a matching covering V_i in the bipartite matching instance $(V_i \dot\cup V(G) \setminus im(\varphi^{i-1}), E_i)$. Here E_i is constructed by adding uv for $u \in V_i$ and $v \in V(G) \setminus im(\varphi^{i-1})$ if

(1) $\mathcal{N}(u) \cap \bigcup_{j=1}^{i-1} V_j = \emptyset$ and $l(u) = l(v)$ or
(2) $\mathcal{N}(u) \cap \bigcup_{j=1}^{i-1} V_j \neq \emptyset$ and $l(u) = l(v)$ and for all $u'u \in E(H)$ with $u' \in \bigcup_{j=1}^{i-1} V_j$ there exists an edge $\varphi^{i-1}(u')v \in E(G)$ with $l(\varphi^{i-1}(u')v) = l(u'u)$.

This means that we can map a vertex from V_i that is not connected to any free vertex in $V(G)$ with correct label, while we need to map neighbors of vertices that are already embedded to neighbors of their images. Figure 3 shows an example of creating a matching instance for V_2 with both kinds of vertices.

Once the matching instance is created by CreateMatchingInstance (Line 8), the algorithm must draw a matching covering all vertices in V_i to extend the (partial) subgraph isomorphism φ^{i-1} to V_i and count the number of all such matchings to get an estimate of the inverse probability. [3, 14] propose to enumerate all maximum matchings in the instance to do so. Before addressing the implementation and runtime of DrawAndCountMatchings (Line 9) and the overall algorithm, we will first give an explicit example of one iteration of the algorithm.

3.1 Example

We will now make an explicit example of one iteration of the main loop of Algorithm 1. Consider the tree H with OBD $D(H) = \{V_1, V_2, V_3, V_4\}$ in Fig. 1a and the labeled graph G in Fig. 1b (node labels are indicated by the border of the vertices; for simplicity, all edges have the same label). We select an image v_1 for r in Line 4 from five candidates and set $\varphi^1(r) = v_1$. For V_2, we construct a bipartite graph between the neighbors $V_2 = \{u_1, u_2, u_3\}$ of r in H and the neighbors $\{v_2, v_3, v_4, v_5\}$ of v_1 in G in Line 8. Fig. 2 shows the corresponding matching instance. There are two maximum matchings, that cover all neighbors of r and we chose $\{u_1v_2, u_2v_3, u_3v_5\}$ at random (Line 9), extend the embedding φ^1 with the selected matching (Line 11), multiply our current estimate by two, and continue. For V_3 and V_4 there is now exactly one maximum matching in the corresponding bipartite instance, e.g. $\{u_4v_6, u_5v_7\}$ for V_4. As a result, the algorithm finds an embedding and returns $5 \cdot 2 \cdot 1 \cdot 1 = 10$ as an estimate for the number of subgraph isomorphisms. The probability that the algorithm finds this embedding is exactly, by construction, the inverse probability 1/10, namely 1/5 for choosing w_1 as image for r and 1/2 for choosing then $\{u_1v_2, u_2v_3, u_3v_5\}$. As one can see by comparing the node degrees and the labels of H and G there exists exactly one embedding of the rooted tree H in Fig. 1a into the Graph G. Hence the expected value of the number of subgraph isomorphisms is equal to one while the output for each iteration of the algorithm is either 0 or 10.

3.2 Runtime Analysis

A rough runtime analysis of Alg. 1 was given in Fürer and Kasiviswanathan [3]. They show that Alg. 1 runs in polynomial time if H has an OBD of bounded (i.e. asymptotically constant) width: The *width* $w(D)$ of an OBD (resp. AD) D is the maximum number of neighbors[4] of vertices in some V_i in any V_j with $j < i$, i.e.

$$w(V_1, \ldots, V_l) = \max_{i \in [l]} \left| \mathcal{N}(V_i) \cap \left(\bigcup_{j<i} V_j \right) \right| .$$

Furthermore, let the *size* of an OBD (resp. AD) be $s(V_1, \ldots, V_l) = l$.

Remark 1. *If H is a tree, a smallest-width OBD $D(H)$ can be constructed in linear time [3]. This can be done by choosing an arbitrary leaf as root $r \in V(H)$, $V_1 = \{r\}$, and an arbitrary top-down traversal $v_1 = r, v_2, \ldots$ of $V(G)$, excluding the leaves. We then define V_i as the set of children of v_{i-1} (see Fig. 1a for an example). Hence the width of $D(H)$ is either $\Delta(H)$ or $\Delta(H) - 1$ and the size of $D(H)$ is equal to the non-leaf vertices of H, where r is not counted as a leaf.*

We now (for the first time) give a more detailed worst-case runtime analysis of Alg. 1 for tree patterns and argue that it is tight for the implementation proposed by [14]. To this end, we need a bound on the number of maximum matchings in a bipartite matching instance. The following fact is well-known in combinatorics:

Lemma 3.2. *Let $G = (A \dot\cup B, E)$ with $|A| \leq |B|$ be a bipartite graph. Then there can be up to*

$$\prod_{i=0}^{|A|-1} (|B| - i) = \frac{|B|!}{(|B| - |A|)!} \tag{1}$$

maximum matchings. The bound is tight for complete bipartite graphs.

E.g., for a complete bipartite graph with $|A| = |B|$ there are $|A|!$ different maximum/perfect matchings. As a result, it may prove costly to explicitly enumerate all maximum matchings just to select a single one of those uniformly at random:

Lemma 3.3. *Let H be a tree with OBD $D(H)$ and G be a graph. Then the worst case runtime of Algorithm 1 as presented in [14] is*

$$O\left(k \cdot s(D(H)) \cdot \frac{\Delta(G)!}{(\Delta(G) - w(D(H))!} \cdot (w(D(H)) + \Delta(G))\right) .$$

[3] Note that there are no edges with both endpoints in V_i and that all edges between vertices in V_j and $V_{j'}$ with $j < j' < i$ were already processed in previous iterations.

[4] Note that Ravkic et al. [14] define the width of an OBD/AD as $\max_{i \in [l]} V_i$. The two definitions can, however, differ by a factor of $O(|E|)$.

Remark 2. *In the case that G is a d-regular graph with $d > \Delta(H)$ the above runtime bound is tight. Furthermore, the bound is exponential in $\Delta(H)$, and grows (for constant $\Delta(H)$) polynomially with increasing $\Delta(G) - \Delta(H)$, with $\Delta(H)$ in the exponent. Hence this implementation can only be applied to trees of bounded (small) degree.*

If H is an arbitrary pattern graph, then Algorithm 1 can be even slower. First, the worst case time complexity of computing an OBD for an arbitrary graph H is unknown. Ravkic et al. [14] give two exponential time algorithms for computing an OBD, if one exists. Second, for general OBDs (or ADs) there is no guarantee that some vertex in V_i is connected to *any* vertices in V_j with $j < i$. Hence, such vertices must be connected in the matching instances (Lines 8–9) to all uncovered vertices of G with identical label and naively implementing the maximum matching enumeration results in a runtime where each instance of $\Delta(G)$ must be replaced by $|V(G)|$. Finally, note that even sampling a maximum matching *almost* uniformly at random takes $O\left(|V(B)|^7 \cdot log\,|V(B)|\right)$ time in a bipartite graph B and an approximate count of the maximum matchings can only be obtained in time $O\left(|V(B)|^{11} \cdot (log\,|V(B)|)^3\right)$ [7].[5]

While we have assumed a d-regular G for simplicity, a similar situation occurs for large dense graphs or large sparse graphs with a high average vertex degree (with a different size of B). Ravkic et al. address this issue by choosing OBDs consisting of small sets (resp. ADs of singletons), which reduces the runtime of the algorithm. This, however, results in a high variance in the estimate, as matching vertices individually reduces the probability of finding a valid embedding, as we will show in Section 6.

4 THE HOPS EMBEDDING ALGORITHM

We now propose drop-in replacements for Algorithm 1 and analyze the resulting runtime if the pattern graph H is a tree. The algorithm, which we call HOPS[6] retains all guarantees provided by [3]. Furthermore, its worst case runtime is linear in the pattern size (instead of exponential in the pattern degree). Hence, the algorithm remains computationally efficient for tree patterns of unbounded degree. Another advantage is that it is very easy to implement; in particular, the algorithm does not require a traditional maximum matching algorithm as a subroutine. Theorem 4.1 summarizes the results of this section.

Theorem 4.1. *Let H be a tree and G be a graph. Then the HOPS algorithm, a specialization of Alg. 1, can be implemented to run in*

$$O\left(k \cdot |V(H)| \cdot \Delta(G)\right) .$$

The main insight that allows these improvements is that the bipartite instances created in Line 8 of Algorithm 1 have special structure when the pattern graph is a tree and its OBD is constructed as in Remark 1. Therefore we can sample a maximum matching uniformly at random if one exists and compute the number of such matchings, both in linear time in the *number of vertices of the bipartite matching instance*. For the remainder of this section, we assume w.l.o.g. that we are given the bipartite graph $(A \dot\cup B, E)$ with $|A| \leq |B|$ in Line 8.

Theorem 4.2. *A maximum matching in Line 9 of Algorithm 1 can be drawn uniformly at random in time $O\left(|A| + |B|\right)$ if the input pattern H is a tree.*

The proof follows directly from Lemmas 4.5 and 4.6, which will be presented in Section 4.1. But first, we prove Theorem 4.1.

Proof of Theorem 4.1. Theorem 4.1 follows from Theorem 4.2 by noting that the matching instances in Line 9 have size at most $|V_i| + \Delta(G)$. Hence one iteration requires $O\left(\sum_{V_i} \left(|V_i| + \Delta(G)\right)\right) = O\left(|V(H)| \cdot \Delta(G)\right)$, as the V_i form a partition of $V(H)$. □

4.1 Block Disjoint Bipartite Graphs

As an easy first step in sampling a maximum matching uniformly at random, we first consider the case that $(A \dot\cup B, E)$ is a complete bipartite graph. Now, sampling a maximum matching uniformly at random is simple: Just fix an arbitrary order on A and select an order of the elements of B uniformly at random. This can be achieved, e.g., by Fisher-Yates shuffle [see, e.g. 9, Chapter 3.4.2] in linear time. We construct a matching by selecting $\{a_i b_i \ : \ i \in [|A|]\}$, where a_i (resp. b_i) is the ith element in the selected order of A (resp. B). Then according to Lemma 3.2, there are $\frac{|B|!}{(|B|-|A|)!}$ such maximum matchings and each has the same probability to be generated by the above method.

For arbitrary bipartite graphs (i.e., where some vertices from A cannot be paired with some other vertices in B), the selection scheme above does not necessarily result in a valid matching. First, there may not be an edge between a_i and b_i in $(A \dot\cup B, E)$. Second, if we consider the matching M that consist of all edges $a_i b_i \in E$, then M might not be maximum. [3, 14] hence resort to explicit enumeration of all maximum matchings to sample one of them with uniform probability and obtaining their count. Recall from Lemma 3.2 that this set has factorial size. In the case of the HOPS algorithm, however, the bipartite matching instances allow for a linear time exact algorithm for the sampling and counting problem.

Definition 4.3 (Block Disjoint Bipartite Graph). A bipartite graph $(A \dot\cup B, E)$ is *block disjoint* if there exist complete bipartite graphs $(A_1 \dot\cup B_1, E_1), \ldots, (A_x \dot\cup B_x, E_x)$ such that

$$(A \dot\cup B, E) = \left(\left(\dot\bigcup_{i=1}^{x} A_i\right) \ \dot\cup \ \left(\dot\bigcup_{i=1}^{x} B_i\right), \dot\bigcup_{i=1}^{x} E_i\right)$$

is the disjoint union of these graphs.

Note that our definition of a bipartite graph allows A_i or B_i to be empty for some or even all indices i. See Figure 2 for an example of a block disjoint bipartite graph $(V_2 \dot\cup \mathcal{N}(v_1), E)$ created in Line 8 of Algorithm 1 for the tree and graph in Figure 1. Note, however, that Figure 3 shows a matching instance arising for general graphs H, G, which is not block disjoint. Next, we show that *all* bipartite matching instances arising in Algorithm 1 are indeed block disjoint if the pattern H is a tree.

Lemma 4.4. *The graph $(A \dot\cup B, E)$ arising in Line 8 of Algorithm 1 is a block disjoint bipartite graph, if H is a tree with an OBD $D(H)$ as constructed in Remark 1.*

[5] The method described by Jerrum et al. addresses perfect matchings, but there exists an easy transformation that results in $(|B|-|A|)!$ perfect matchings for each maximum matching in the original graph.

[6] For convenience, we give the detailed pseudo-code in the appendix.

Algorithm 2 Sampling a maximum matching uniformly at random for block disjoint bipartite graphs

input: A block disjoint bipartite graph $(A \dot\cup B, E)$
output: A maximum matching M drawn uniformly at random and the count of all such matchings in $(A \dot\cup B, E)$

1: Split $(A \dot\cup B, E)$ into disjoint complete bipartite graphs $(A_j \dot\cup B_j, E_j)$, with $|A_j| \leq |B_j|$
2: $M := \emptyset$; $c := 1$
3: **for all** $(A_j \dot\cup B_j, E_j)$ **do**
4: Draw a maximum matching M_j in $(A_j \dot\cup B_j, E_j)$ uniformly at random
5: $M := M \cup M_j$
6: $c := c \cdot \prod_{k=0}^{|A_j|} (|B_j| - k)$
7: **return** M, c

The proof can be found in the appendix.

Algorithm 2 shows the pseudo-code of an algorithm computing the number of maximum matchings in a block disjoint bipartite graph arising in Algorithm 1. It implements both CreateMatchingInstance and DrawAndCountMatchings for a partition set V_i, $i > 1$ that consists of the children of some vertex $v_i \in V(H)$ (cf. Remark 1). Alg. 2 splits the bipartite graph $(A \dot\cup B, E)$ into the fully connected blocks, draws a random matching in each block as described in the beginning of this section and returns the union of all these matchings. Simultaneously, it computes the number of maximum matchings in each block and returns the product of these numbers. The lemma below shows that this behavior indeed results in the required output.

Lemma 4.5. *Algorithm 2 draws a maximum matching of a block disjoint bipartite graph $(A \dot\cup B, E)$ uniformly at random and returns the number of maximum matchings.*

Proof. To prove the lemma, we show that the union of maximum matchings drawn individually for the fully connected blocks is a maximum matching in the block disjoint bipartite graph drawn uniformly at random. As the complete bipartite graphs are disconnected from each other, each maximum matching of the block disjoint bipartite graph can be partitioned into a set of maximum matchings in the blocks. Furthermore, the union of a maximum matching for each block is a maximum matching in the block disjoint bipartite graph. In fact, the number of maximum matchings in the whole graph is the product over the numbers of maximum matchings in each complete bipartite block. □

Lemma 4.6. *Algorithm 2 requires $O(|A| + |B|)$ time if the block disjoint bipartite graph $(A \dot\cup B, E)$ is given as in Lemma 4.4.*

The proof can be found in the appendix.

5 ERROR BOUNDS

The hoPS embedding algorithm is a proper drop-in replacement, so Thm. 1.2 in Fürer and Kasiviswanathan [3] can be applied.

Corollary 5.1. *Let $G \in \mathcal{G}(n,p)$ be Erdős-Rényi and H a tree with bounded vertex degree. Then, the hoPS algorithm is an FPRAS for estimating the number of copies of H in G.*

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\varnothing\delta(G)$	$\Delta(G)$	density
YEAST	16 233	18 355	2.26	124	$1.4 \cdot 10^{-4}$
DBLP	393 230	447 650	2.28	1 036	$5.7 \cdot 10^{-6}$
WEBKB	5 732	6 750	2.36	133	$2.0 \cdot 10^{-4}$
FB	28 057	112 252	8.00	1 051	$3.0 \cdot 10^{-4}$
AMAZON	334 863	925 872	5.53	549	$8.3 \cdot 10^{-6}$
ORKUT	3 072 441	117 185 083	76.28	33 313	$1.2 \cdot 10^{-5}$
LIVEJOURNAL	3 997 962	34 681 189	17.35	14 815	$2.2 \cdot 10^{-6}$

Table 1: Graph datasets

However, in practice it is infeasible to calculate the constants involved in their bound on the number of iterations k. Therefore, we bound the number of iterations required to achieve a given estimation accuracy ϵ with probability $1 - \delta$ for arbitrary graphs G based only on the number of vertices in H and the vertex degree of G and H. For that, recall that each iteration of Alg. 1 is an estimator c_i with $i \in [k]$ and final estimate of Alg. 1 is $C^k = k^{-1} \sum_{i=1}^{k} c_i$. Let $C_H(G)$ denote the set of subgraph isomorphisms from a tree H to G. Since all c_i are unbiased estimators, $\mathbb{E}[c_i] = \mathbb{E}[C^k] = |C_H(G)|$. Furthermore, for all i, $0 \leq c_i \leq U_H(G)$, where $U_H(G)$ denotes an upper bound on $|C_H(G)|$. It follows from Lemma 3.2 that

$$U_H(G) \leq |V(G)| \left(\frac{\Delta(G)!}{(\Delta(G) - \Delta(H))!} \right)^{|V(H)|-1} \leq |V(G)|\, \Delta(G)^{|V(H)|\Delta(H)} \,. \tag{2}$$

With this we can bound the estimation error for arbitrary graphs.

Proposition 5.2. *For $\epsilon, \delta > 0$, a tree H and a graph G , it holds for Alg. 1 with the hoPS embedding that*

$$\mathbb{P}\left[\left| C^k - |C_H(G)| \right| \geq \epsilon\, |C_H(G)| \right] \leq 2 \exp\left(- \frac{2k\epsilon^2\, |C_H(G)|^2}{|V(G)|\, \Delta(G)^{|V(H)|\Delta(H)}} \right).$$

In order to achieve a confidence of δ, Alg. 1 must be run with

$$k \geq \frac{|V(G)|\, \Delta(G)^{|V(H)|\Delta(H)}}{2\epsilon^2\, |C_H(G)|^2} \ln \frac{2}{\delta} \,.$$

The proof can be found in the appendix. Note that this bound is linear in $|V(G)|$ and polynomial in $\Delta(G)$ but exponential in $|V(H)|$ and $\Delta(H)$.

6 EXPERIMENTAL EVALUATION

We now show that in practice hoPS achieves low error rates, outperforms state-of-the-art algorithms, and improves on [14]. In particular, we consider two tasks: (1) Estimating the number of embeddings of tree patterns in single large graphs (Sec. 6.1) and (2) finding an approximate set of frequent subtrees in arbitrary databases of small to medium size graphs (Sec. 6.2). All experiments are performed on an Intel i7-4770 CPU with 3.40GHz and 16 GB DDR3-1600 RAM. Our code and data are available at https://github.com/pwelke/hops.

6.1 Approximate Counting in Large Graphs

In the first part of the experiments we compare hoPS to the natural baseline algorithms introduced by Ravkic et al. [14] and an exact

Graph	pattern size	#finished (#total)	∅ #embed.	∅ runtime [s]
YEAST	15	87(87)	49 236(±53 228)	2 458(±3 853)
DBLP	10	24(26)	322 879(±353 522)	2 698(±6 453)
WEBKB	10	22(92)	153 129(±173 630)	6 302(±8 450)
FB	10	24(68)	176 150(±0)	2 326(±158)

Table 2: Results of the exact algorithm

algorithm [17]. We implemented the HOPS algorithm within their framework[7]. In the second part, we investigate how HOPS scales to large unlabeled graphs from [11] (see the last three graphs from Table 1). To this end, we implemented our algorithm in the snap framework [12], as it is more memory efficient than the pure Python implementation of Ravkic et al..

Comparison to Exact and Approximate Algorithms. For exact evaluation and comparison with Ravkic et al. [14] we choose the graphs YEAST, DBLP, WEBKB and FACEBOOK (FB) (see Table 1) and pattern graphs of sizes 10 and 15 in the case of YEAST such that the exact algorithm finishes for an acceptable number of patterns (c.f. Ravkic et al. [14] and Table 2). Due to space limitations other graphs considered by Ravkic et al. [14] and smaller pattern sizes they provided are not presented in this paper. For each graph 100 patterns of the size reported in Tab. 2 which are known to appear in the graph are evaluated. As HOPS is specific for trees all the evaluations are done with tree patterns. Except for DBLP, over two thirds of all given patterns are trees, see Table 2. At first, we run the exact algorithm of Ullmann [17] with a time limit of 10h for each tree pattern for all the graph data. Only tree patterns for which the exact algorithm finishes within this time limit are used for evaluation. The average[8] runtime for finding the exact number of embeddings together with the average of the exact embedding numbers is denoted in Table 2.

As a first evaluation we compare the relative error wrt. the exact number of embeddings of the HOPS algorithm to that of OBD, AD and RANDOM introduced in [14]. OBD implements Alg. 1 with exact OBD search, AD uses a flattened OBD that contains singleton partitions. The algorithms run on each tree pattern (chosen as described above) within a time limit of 60 seconds on the corresponding graphs for the estimation. Figure 4 shows the average relative error every 2 seconds. HOPS significantly outperforms all competitors on the four datasets. After one minute the relative error on all datasets is at most half as high as the closest competitor (c.f. Table 3). This is impressive noting that the graphs have different structure. While DBLP is relatively sparse having some nodes with very high degree, YEAST, WEBKB and FB are denser graphs (c.f. Table 1). Moreover, HOPS is also more stable concerning the standard deviation of its predictions as shown by the filled areas in Fig. 4 and in Table 3. Another advantage of HOPS, explaining its good performance comparing to OBD and AD, is that the number of iterations is higher and it also more frequently finds an embedding, see Table 3. E.g., for YEAST and DBLP HOPS finds an embedding in 0.49% resp. 12.3%

[7]Code and data are available at https://dtai.cs.kuleuven.be/software/gs-srl.
[8]The average is always taken over all patterns where the exact algorithm finished within the time limit. Standard deviations are displayed in parentheses.

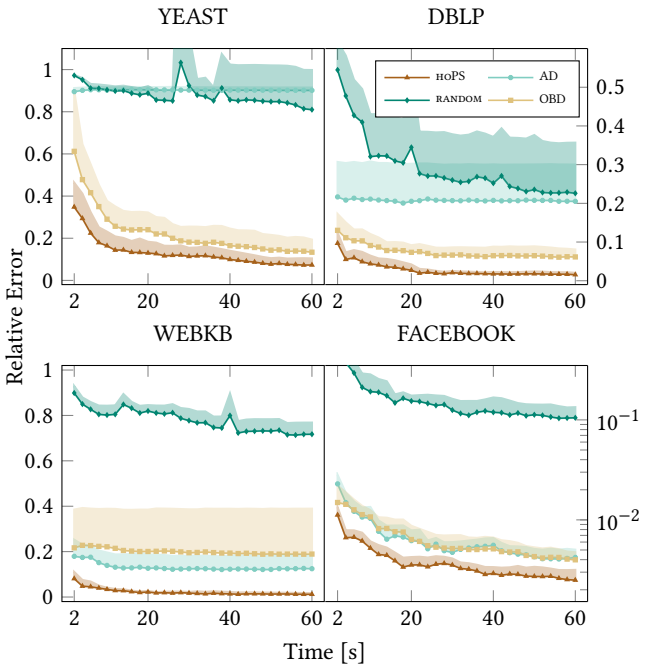

Figure 4: Comparisons of HOPS, OBD, AD and random baseline algorithm on YEAST, WEBKB and DBLP graphs

of all iterations while for OBD we have 0.19% resp. 10.4% and for AD 0.41% resp. 11.2%. AD has a comparable success rate only for WEBKB. We assume that this is the result of (1) solving matching instances instead of just embedding single neighbors as in AD and (2) ensuring locality through our choice of OBD.

Considering the convergence speed towards the expected value we looked at the relative errors of the algorithms pattern-wise. In Figure 5 each line shows the relative error on one tree pattern for the YEAST graph. Note that the error is only displayed for discrete time steps while the curves between two time steps are interpolated linearly for presentation reasons. In this case we only compared the two best algorithms on YEAST. Here, for some patterns the OBD algorithm needs a lot more time to converge. The same behaviour can also be observed in the case of the other graphs.

Finally, we note that the evaluation and hence the results differ significantly from the ones presented in [14]. First, we only look at tree patterns which is justified by [19]. Second, we consider a fixed time interval of one minute for each pattern, while Ravkic et al. normalize their plots by the runtime of the exact algorithm, which is usually infeasible in practice and hence disregarded in this paper.

Scaling Experiments. We now resort to larger graphs and larger patterns and consider the last three graphs given in Tab. 1. Due to their size we cannot provide relative error plots, since any exact algorithm is not expected to finish in acceptable time. To obtain candidate patterns, we instead sample trees uniformly at random which are likely to be frequent since G is unlabeled. To investigate how HOPS scales with the pattern size on large graphs G, we generate 50 random trees H of size $|V(H)| = 10, 20, \ldots, 50$ each and plot it against the runtime per iteration. Figure 6 (left) shows a linear

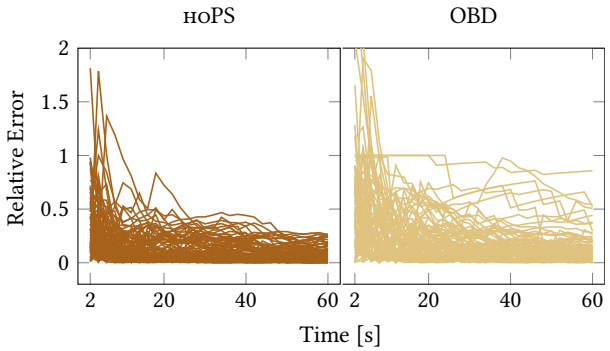

Figure 5: Pattern-wise relative error of HOPS (left) and OBD (right) on YEAST. Each line indicates one pattern.

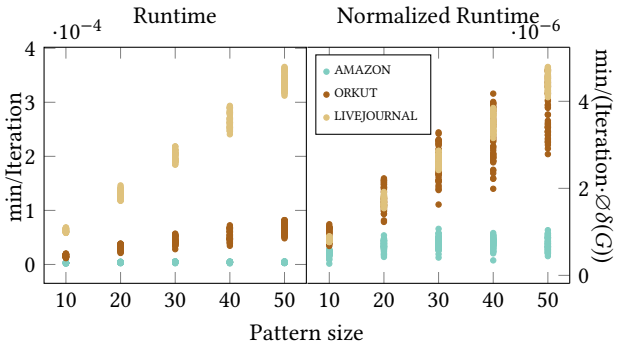

Figure 6: Runtime evaluation of HOPS for large graphs as a function of tree size

relation between the runtime and the pattern size, indicating that HOPS establishes a new state of the art with respect to pattern size.

Normalizing with the average degree of G (right) shows that the runtime is furthermore bounded linearly by the average degree of G (note that Lemma 4.6 only shows that it is bounded by the maximum degree of G). Thus, HOPS is at least as fast as the state-of-the-art in terms of G. Note that the (normalized) runtime of HOPS on the AMAZON dataset is substantially lower accross all pattern sizes. This is due to the relatively low average degree (5.53) of AMAZON which results in a high number of (fast) early terminations of HOPS, as the random trees have a typical maximum degree of 4–6.

6.2 Probabilistic Frequent Subtree Mining

As a second application of our HOPS algorithm, we evaluate its suitability for frequent subgraph mining in transactional graph databases. Here, an important task is to decide whether a candidate pattern is subgraph isomorphic to at least t graphs in a given database $D = [G_1, G_2, \ldots, G_N]$ of (small) graphs. We report results on standard benchmark graph databases used in the graph kernel community and on synthetic graphs created in [20].

HOPS can be used in this setting to decide with one-sided error, whether a pattern is present in a graph, or not. That is, if HOPS finds at least one embedding, we know that the pattern is subgraph isomorphic to a graph; if no iteration is successful, the pattern might or might not be subgraph isomorphic to a given transaction graph. Other algorithms with this behavior have been studied under the name probabilistic subtree mining [19, 20]. These methods work by drawing a set of spanning trees for each graph in the database randomly and then mining all frequent subtrees in the resulting database consisting of a forest of spanning trees for each graph. In the context of one-sided error an increased number of frequent patterns for identical frequency threshold implies a lower error, even if we do not know the true number of frequent patterns. We hence evaluate our methods w.r.t. patterns found per time and repeated the experiments in [19, 20] comparing to their probabilistic subtree (PS) and boosted probabilistic subtrees (BPS) methods.

Figure 7 shows the number of patterns found in a given time budget. Each marker on a line corresponds to a setting of the sampling parameters for the methods HOPS, PS, and BPS. HOPS outperforms the other probabilistic methods on the synthetic graph datasets and on the more complicated real world datasets (AIDS99, DD, and POKEC) by a large margin. For example, on DD HOPS requires 457 seconds to obtain 32447 patterns, PS 1039 seconds for only 29493 patterns, and BPS 1453 seconds for 29594 patterns. On NCI1, however, HOPS is outperformed by PS and BPS. An exploratory analysis of the patterns found by PS and HOPS suggests that PS finds more patterns of large maximum degree when the transaction graphs have low maximum degree (as is the case for molecular datasets).

7 CONCLUSION

We have proposed HOPS, a fast and easy-to-implement algorithm to estimate the number of subgraph isomorphisms from a tree into an arbitrary graph. We have provided theoretical guarantees on the runtime and variance of our algorithm and have shown that it outperforms its competitors by a large margin in two relevant scenarios. We note that HOPS only requires local read-only access to the transaction graph. Hence it can easily run in parallel and can be integrated in large scale distributed graph databases.

Our results rely on a fast algorithm to sample maximum matchings. As an immediate consequence of Lemma 4.6, given a block disjoint bipartite graph $(A \dot\cup B, E)$, we can preprocess it in $O(|E|)$ time to obtain the hash map representation above and are then able to (1) compute the number of maximum matchings and (2) draw a maximum matching uniformly at random in $O(|A| + |B|)$ time. It is an open question whether other classes of bipartite graphs allow linear time algorithms for both these problems.

Finally, our algorithm can be adapted to estimate the number of homomorphisms from a tree into a graph, as well. In this setting, Algorithm 2 only needs to (1) draw elements from $B \cup \{p\}$ with replacement and (2) give $|B \cup \{p\}|^{|A|}$ as the correct estimate of partial homomorphisms in a single block of a block disjoint bipartite graph. Drawing with replacement can be implemented in the same asymptotic time as shuffling, hence the runtime analysis still holds.

ACKNOWLEDGEMENTS

This research was partially funded by the Federal Ministry of Education and Research of Germany as part of the competence center for machine learning ML2R (01|S18038C) and was partially funded by the German Research Foundation (DFG) under Germany's Excellence Strategy – EXC 2070 – 390732324.

Graph	iterations / min			successful tries rate			relative error			relative std. deviation		
	HOPS	OBD	AD	HOPS	OBD	AD	HOPS	OBD	AD	HOPS	OBD	AD
YEAST	635 639	423 792	429 777	0.0049	0.0019	0.0041	0.073	0.134	0.901	0.10	0.19	0.91
DBLP	554 945	382 822	326 877	0.123	0.104	0.112	0.016	0.062	0.205	0.020	0.081	0.300
WEBKB	568 284	353 742	276 121	0.264	0.162	0.251	0.012	0.189	0.125	0.017	0.389	0.184
FACEBOOK	238 849	77 762	47 859	0.999	0.999	0.999	0.002	0.004	0.004	0.003	0.005	0.005

Table 3: Extended Results of the estimated subgraph isomorphism number for HOPS, OBD and AD with the number of algorithm tries, the success rate, the relative error after 60*s* and the relative standard deviation over the patterns after 60*s*.

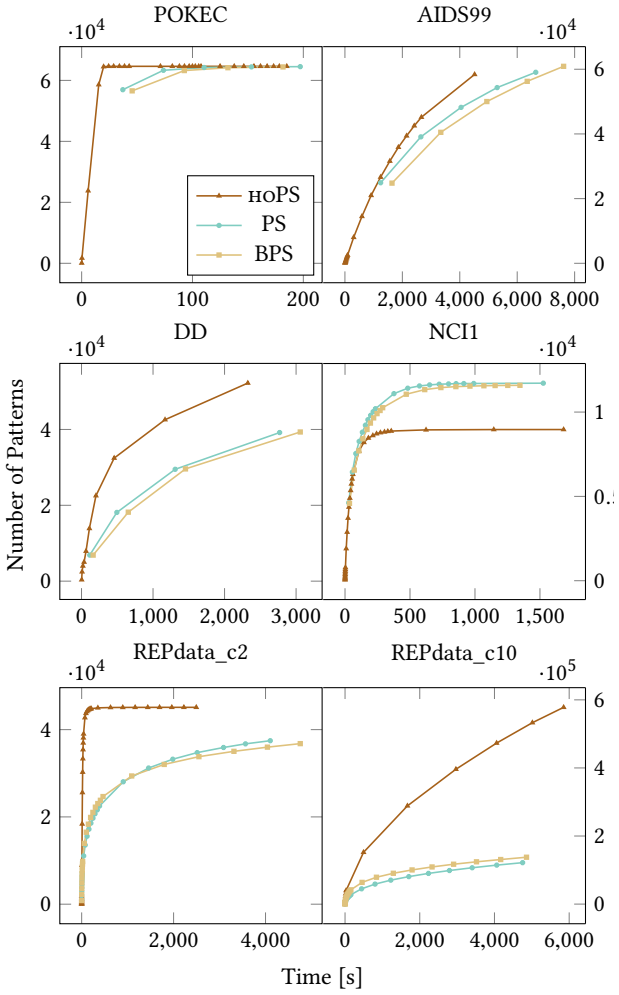

Figure 7: Recall of frequent patterns per time in probabilistic pattern mining on real world graph datasets

REFERENCES

[1] Marco Bressan, Stefano Leucci, and Alessandro Panconesi. 2019. Motivo: Fast Motif Counting via Succinct Color Coding and Adaptive Sampling. *PVLDB* 12, 11 (2019), 1651–1663. https://doi.org/10.14778/3342263.3342640

[2] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. 2005. Frequent substructure-based approaches for classifying chemical compounds. *Transactions on Knowledge and Data Engineering* 17, 8 (Aug. 2005), 1036–1050. https://doi.org/10.1109/tkde.2005.127

[3] Martin Fürer and Shiva Prasad Kasiviswanathan. 2014. Approximately Counting Embeddings into Random Graphs. *Combinatorics, Probability & Computing* 23, 6 (2014), 1028–1056. https://doi.org/10.1017/S0963548314000339

[4] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman.

[5] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. 2007. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery* 15, 1 (2007), 55–86. https://doi.org/10.1007/s10618-006-0059-1

[6] Tamás Horváth and Jan Ramon. 2010. Efficient frequent connected subgraph mining in graphs of bounded tree-width. *Theoretical Computer Science* 411, 31–33 (2010), 2784–2797. https://doi.org/10.1016/j.tcs.2010.03.030

[7] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. 2004. A polynomial-time approximation algorithm for the permanent of a matrix with nonnegative entries. *J. ACM* 51, 4 (2004), 671–697. https://doi.org/10.1145/1008731.1008738

[8] Ashraf M. Kibriya and Jan Ramon. 2013. Nearly exact mining of frequent trees in large networks. *Data Mining and Knowledge Discovery* 27, 3 (2013), 478–504. https://doi.org/10.1007/s10618-013-0321-2

[9] Donald E. Knuth. 1998. *The art of computer programming, volume 2: (2nd ed.) seminumerical algorithms.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.

[10] Ioannis Koutis and Ryan Williams. 2009. Limits and Applications of Group Algebras for Parameterized Problems. In *International Colloquium on Automata, Languages and Programming (ICALP) Proceedings, Part I (Lecture Notes in Computer Science, Vol. 5555).* Springer, 653–664. https://doi.org/10.1007/978-3-642-02927-1_54

[11] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[12] Jure Leskovec and Rok Sosič. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 1 (2016), 1.

[13] Kirill Paramonov, Dmitry Shemetov, and James Sharpnack. 2019. Estimating Graphlet Statistics via Lifting. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, (KDD) Proceedings.* ACM, 587–595. https://doi.org/10.1145/3292500.3330995

[14] Irma Ravkic, Martin Žnidaršič, Jan Ramon, and Jesse Davis. 2018. Graph sampling with applications to estimating the number of pattern embeddings and the parameters of a statistical relational model. *Data Mining and Knowledge Discovery* 32, 4 (2018), 913–948. https://doi.org/10.1007/s10618-018-0553-2

[15] Pedro Ribeiro, Pedro Paredes, Miguel E. P. Silva, David Aparicio, and Fernando Silva. 2019. A Survey on Subgraph Counting: Concepts, Algorithms and Applications to Network Motifs and Graphlets. *CoRR* abs/1910.13011 (2019), 1–35. arXiv:1910.13011 http://arxiv.org/abs/1910.13011

[16] Till Hendrik Schulz, Tamás Horváth, Pascal Welke, and Stefan Wrobel. 2018. Mining Tree Patterns with Partially Injective Homomorphisms. In *European Conference on Machine Learning and Knowledge Discovery in Databases ECML PKDD Proceedings, Part II (Lecture Notes in Computer Science, Vol. 11052).* Springer, 585–601. https://doi.org/10.1007/978-3-030-10928-8_35

[17] Julian R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42. https://doi.org/10.1145/321921.321925

[18] Takeaki Uno. 1997. Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs. In *International Symposium on Algorithms and Computation (ISAAC) Proceedings (Lecture Notes in Computer Science, Vol. 1350).* Springer, 92–101. https://doi.org/10.1007/3-540-63890-3_11

[19] Pascal Welke, Tamás Horváth, and Stefan Wrobel. 2018. Probabilistic Frequent Subtrees for Efficient Graph Classification and Retrieval. *Machine Learning* 107, 11 (2018), 1847–1873. https://doi.org/10.1007/s10994-017-5688-7

[20] Pascal Welke, Tamás Horváth, and Stefan Wrobel. 2019. Probabilistic and Exact Frequent Subtree Mining in Graphs Beyond Forests. *Machine Learning* 108, 7 (2019), 1137–1164. https://doi.org/10.1007/s10994-019-05779-1

Algorithm 3 The HOPS embedding algorithm

input: A tree H, a graph G, and an integer $k > 0$
output: An unbiased estimate of the number of subgraph isomorphisms φ from H to G.

1: $C := \emptyset$
2: Select $r \in V(H)$ uniformly at random
3: **for all** $k \in \{1, \dots, k\}$ **do**
4: **for all** $v \in V(H)$ **do** $\varphi(v) := None$
5: Draw $v \in W = \{w \in V(G) : l(w) = l(r)\}$ uniformly at random
6: $\varphi(r) := v$
7: $c := |W| \cdot \text{EmbedTree}(r, v, None)$
8: $C := C \cup \{c\}$
9: **return** $\frac{1}{|C|} \sum_{c \in C} c$

Function EmbedTree(u, v, p):

10: Initialize empty hash maps A, B
11: **for all** $a \in \mathcal{N}(u) \setminus \{p\}$ **do** $A[l(a)\_l(ua)].append(a)$
12: **for all** $b \in \mathcal{N}(v) \setminus im(\varphi)$ **do** $A[l(b)\_l(vb)].append(b)$
13: **for all** $x \in A.keys()$ **do**
14: **if** $|A[x]| \leq |B[x]|$ **then**
15: Shuffle $B[x]$
16: **for all** $i \in [|A[x]|]$ **do** $\varphi(A[x][i]) := B[x][i]$
17: $c := c \cdot \frac{|B[x]|!}{(|B[x]|-|A[x]|)!}$
18: **else** $c := 0$
19: **if** $c \neq 0$ **then**
20: **for all** $a \in \mathcal{N}(u) \setminus \{p\}$ **do**
21: $c = c \cdot \text{EmbedTree}(a, \varphi(a), u)$
22: **return** c

A THE HOPS EMBEDDING ALGORITHM

Algorithm 3 shows detailed pseudo code of the HOPS embedding algorithm for convenience of implementation. In contrast to Algorithm 1, we write it as a recursive algorithm that traverses the pattern tree H and do not explicitly compute the OBD given by Remark 1.

We use Python inspired notation for the hash maps A, B. Namely, $A[x]$ stores a list that is indexed by integers, hence $A[x][i]$ returns the ith value of that list. "_" is a special symbol that is not contained in the label set Σ, used to concatenate labels.

B ADDITIONAL PROOFS

In this section we provide the proofs of Lemma 3.3, Lemma 4.4, Lemma 4.6, and Proposition 5.2.

Proof of Lemma 3.3. Let $G = K_n$ be the complete graph on n vertices and let G, H be unlabeled and $|V(H)| << n$. Choose $D(H)$ as in Remark 1. As $V_1 = \{r\}$, we can draw an initial image of $\varphi(r) \in V(G)$ in constant time (Line 5).[9] For any V_i with $i > 2$, the bipartite matching instance in Line 8 can be constructed between V_i and the uncovered vertices in $\mathcal{N}(\varphi(v_i))$ in G. The algorithm now enumerates all maximum matchings and chooses one uniformly at random. The best known algorithm for this subtask runs in $O(|V_i| + \Delta(G))$ per matching [18]. Hence, according to Lemma 3.2, Line 9 requires up to $O\left(\frac{\Delta(G)! \cdot (\Delta(G)+|V_i|)}{(\Delta(G)-|V_i|)!}\right)$ time to finish. As neither Fürer and Kasiviswanathan [3] nor Ravkic et al. [14] give any additional insight in the matching instances considered for extension of the patterns, this is the best run time we can assume for their algorithm. The runtime of the remaining steps is dominated by this call. As a result, one iteration of the outer loop runs in

$$O\left(\sum_{V_i \in D(H)} \frac{\Delta(G)!(|V_i| + \Delta(G))}{(\Delta(G) - |V_i|)!}\right) = O\left(\sum_{v \in V(H)} \frac{\Delta(G)!(\delta(v) + \Delta(G))}{(\Delta(G) - \delta(v))!}\right)$$
$$= O\left(\Delta(G)\,|V(H)|\,\frac{\Delta(G)!}{(\Delta(G) - \Delta(H))!}\right).$$

□

Proof of Lemma 4.4. By definition of E_i in Algorithm 1 it follows that $(A \dot\cup B, E)$ is a bipartite graph. Moreover, by the definition of the tree OBD $D(H)$ the partition $A = V_i$ consists of the children of some $u \in V_j$ for $1 \leq j < i$. Now consider two vertices $a \in A$ and $b \in B$. If $l(a) = l(b)$ and $l(ua) = l(\varphi^{i-1}(u)b)$ then, by definition, $ab \in E$. Hence, for a fixed pair of labels $l, l' \in \Sigma$ the subgraph induced by the vertices $a \in A$ with $l(a) = l$ and $l(ua) = l'$ and $b \in B$ with $l(b) = l$ and $l(\varphi^{i-1}(u)b) = l'$ is a complete bipartite graph. On the other hand, if $l(a) \neq l(b)$ or $l(ua) \neq l(\varphi^{i-1}(u)b)$ then $ab \notin E$. Thus $(A \dot\cup B, E)$ separates into disjoint complete bipartite subgraphs that are indexed by pairs of labels $(l, l') \in \Sigma^2$. □

Proof of Lemma 4.6. Assuming that the labels of vertices and edges of the graphs G and H come from a fixed alphabet Σ, we can compute the disjoint blocks directly from the label information of A and B. Using two hash maps[10], $H(A)$ indexed by $(l(a), l(\varphi^{i-1}(u)a) \in \Sigma \times \Sigma$ (resp. $H(B)$ indexed by $(l(b), l(\varphi^{i-1}(u)b))$ that hold a list of vertices for each such label pair. Note that there are at most $|A|$ different indices in $H(A)$. Hence, there are linearly many keys, each of which can be accessed in constant time. To compute a random maximum matching, we iterate over all keys. For each such key $(l(a), l(\varphi^{i-1}(u)a))$, we obtain the subsets of A and B that are labeled in this way. These are exactly the left and right hand side of the complete bipartite block that corresponds to $(l(a), l(\varphi^{i-1}(u)a))$. Hence, shuffling the larger of the two lists of vertices and assigning vertices as described in the proof of Lemma 4.5 yields the desired result. Hence, as hashing all (linearly many) label combinations and iterating over them can be done in linear time, and as the subsets stored in the hash maps as lists form a partition of A (resp. B), the whole algorithm can be implemented to run in linear time. □

Proof of Proposition 5.2. Assume that $|C_H(G)| > 0$, then the theorem follows directly from Hoeffding's inequality, using that $0 \leq c_i \leq U_H(G)$ and that $\mathbb{E}[C^k] = |C_H(G)|$, i.e.,

$$\mathbb{P}\left[\left|C^k - |C_H(G)|\right| \geq \epsilon\,|C_H(G)|\right] \leq 2\exp\left(-\frac{2k\epsilon^2\,|C_H(G)|^2}{U_H(G)}\right)$$

For $|C_H(G)| = 0$, the output of Alg. 1 will always be zero, so the error probability is $\mathbb{P}\left[C^k = 0 \middle| |C_H(G)| \neq 0\right] \leq \mathbb{P}\left[C^k = 0 \middle| |C_H(G)| = 1\right]$, which is bounded by the error probability for $\epsilon = 1$ and $|C_H(G)| = 1$. Solving for k and using Eq. 2 yields the result. □

[9] Assuming usual graph data structures. Using appropriate index structures, the same is possible for labeled graphs.

[10] Alternatively, one can implement the algorithm in-place using sorting.