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

Probabilistic Frequent Subtrees for Efficient Graph Classification and Retrieval

Pascal Welke · Tamás Horváth · Stefan Wrobel

Abstract Frequent subgraphs proved to be powerful features for graph classification and prediction tasks. Their practical use is, however, limited by the computational intractability of pattern enumeration and that of graph embedding into frequent subgraph feature spaces. We propose a simple probabilistic technique that resolves both limitations. In particular, we restrict the pattern language to trees and relax the demand on the completeness of the mining algorithm, as well as on the correctness of the pattern matching operator by replacing transaction and query graphs with small random samples of their spanning trees. In this way we consider only a random subset of frequent subtrees, called probabilistic frequent subtrees, that can be enumerated efficiently. Our extensive empirical evaluation on artificial and benchmark molecular graph datasets shows that probabilistic frequent subtrees can be listed in practically feasible time and that their predictive and retrieval performance is very close even to those of complete sets of frequent subgraphs. We also present different fast techniques for computing the embedding of unseen graphs into (probabilistic frequent) subtree feature spaces. These algorithms utilize the partial order on tree patterns induced by subgraph isomorphism and, as we show empirically, require much less evaluations of subtree isomorphism than the standard brute-force algorithm. We also consider partial embeddings, i.e., when only a part of the feature vector has to be calculated. In particular, we propose a highly effective practical algorithm that significantly reduces the number of pattern matching evaluations required by the classical min-hashing algorithm approximating Jaccard-similarities.

Pascal Welke
Department of Computer Science, University of Bonn, Germany
E-mail: welke@uni-bonn.de

Tamás Horváth
Department of Computer Science, University of Bonn and Fraunhofer IAIS, Sankt Augustin, Germany
E-mail: tamas.horvath@cs.uni-bonn.de

Stefan Wrobel
Fraunhofer IAIS, Sankt Augustin and Department of Computer Science, University of Bonn, Germany
E-mail: stefan.wrobel@iais.fraunhofer.de

1 Introduction

A common paradigm in distance-based learning is to embed the instance space into some appropriately chosen feature space equipped with a metric and to define the dissimilarity between instances by the distance of their images in the feature space. In this work we deal with the special case that the instances are arbitrary (labeled) graphs and the feature space is the d-dimensional *Hamming space* (i.e., $\{0,1\}^d$) spanned by the elements of a pattern set of cardinality d for some $d > 0$. The crucial step of this generic approach is the appropriate choice of the pattern (or feature) set. Indeed, the quality (e.g., predictive performance) of this method applied to some particular problem strongly depends on the semantic relevance of the patterns considered, implying that one might be interested in pattern languages of high expressive power. Our goal in this paper is to use *frequent* subgraphs as features, *without* any structural restriction on the graph class defining the instance space. This is motivated, among others, by the observation that frequent subgraph based learners (see, e.g., Deshpande et al, 2005) are of remarkable predictive performance for example on the ligand-based virtual screening problem (Geppert et al, 2008).

Our goal involves two steps solving the following computational problems:

(i) Pattern mining: *Given* a (training) database of arbitrary graphs, *compute* the set $\mathcal{F}$ of all frequent subgraphs with respect to some user specified frequency threshold.
(ii) Graph embedding: *Given* an unseen graph (usually drawn from the same distribution as the training data set), *compute* its embedding into the Hamming space spanned by the elements of $\mathcal{F}$.

For the case that the embedding operator is defined by subgraph isomorphism and that there is no restriction on the transaction and query graphs, both steps are computationally intractable. The pattern mining problem (i) has gained lots of attention (see, e.g., Deshpande et al, 2005; Nijssen and Kok, 2005; Chi et al, 2005; Zhao and Yu, 2008), resulting in several practical systems for graph databases restricted in different ways. The graph embedding problem (ii) is, however, often neglected in the literature, though it is crucial to the ability of applying the pattern set generated in the first step.

To arrive at a *practically* fast system, we restrict the pattern language to *trees.* This restriction alone, however, does not resolve the complexity problems above. Regarding problem (i), mining frequent subtrees from arbitrary graphs is *not* possible in output polynomial time (unless P=NP, Horváth et al, 2007). Regarding problem (ii), deciding subgraph isomorphism from a tree into a graph is NP-complete (Garey and Johnson, 1979), implying that computing the embedding of arbitrary graphs into feature spaces spanned by tree patterns is computationally intractable. To overcome these limitations, we give up the demand on the completeness of the mining algorithm and that on the correctness of the pattern matching operator (i.e., subgraph isomorphism).

Regarding the mining algorithm, for each training graph in the mining step we first take a set of k spanning trees generated uniformly at random, where k is some user specified parameter, and replace each training graph with a random forest obtained by the vertex disjoint union of its k random spanning trees. We then calculate the set of frequent subtrees for this forest database for some user specified frequency threshold. Clearly, the output of this probabilistic technique is always *sound* (any tree found to be frequent by this algorithm is a frequent subtree in the original dataset), but *incomplete* (the algorithm may miss frequent subtrees). Since frequent subtrees in forests can be generated with polynomial delay (see, e.g., Horváth and Ramon, 2010), our frequent pattern generation algorithm runs in time polynomial in the combined size of the training dataset $\mathcal{D}$ and the set of frequent subtrees generated, as long as k is bounded by a polynomial of the size of $\mathcal{D}$. Our extensive empirical evaluation demonstrates that the above idea results in a practically feasible mining algorithm. In particular, we show that the *recall*, i.e., the fraction of frequent patterns retrieved by our probabilistic mining technique is high even for small values of k and that the retrieved set of patterns is very stable. (Notice that *precision* is always 100% for the soundness of the algorithm.)

We follow a similar strategy for the pattern matching operator used in the embedding step: For an unseen graph G and a set $\mathcal{F}$ of tree patterns enumerated in the mining step, we generate a set $\mathfrak{S}_k(G)$ of k random spanning trees of G and compute the set of all $T \in \mathcal{F}$ that are subgraph isomorphic to $\mathfrak{S}_k(G)$. The incidence vector of this set defines the embedding of G into the Hamming space spanned by $\mathcal{F}$. On the one hand, in this way we decide subgraph isomorphism from a tree into a graph with one-sided error, as only a negative answer may be erroneous, i.e., when T is subgraph isomorphic to G, but not to $\mathfrak{S}_k(G)$. On the other hand, this probabilistic pattern matching test can be performed in polynomial time (Shamir and Tsur, 1999), in contrast to the correct pattern evaluation. We show that our probabilistic algorithm decides subgraph isomorphism from T into G correctly with high probability if T is frequent in G and k is chosen appropriately.

Using the probabilistic technique sketched above, we can compute the embedding of a graph in polynomial time by deciding subgraph isomorphism for *all* trees in the pattern set. This brute-force algorithm can be accelerated by reducing the number of subgraph isomorphism tests. Utilizing that subgraph isomorphism induces a partial order on the pattern set and that it is anti-monotone with respect to this order, we can infer for certain patterns whether or not they match a graph from the evaluations already performed for other patterns. We propose two such strategies. One is based on a greedy *depth-first search* traversal, the other uses *binary search* on paths in the pattern poset. We show empirically that both algorithms drastically reduce the number of embedding operator evaluations compared to the baseline obtained by levelwise search.

The high dimensionality of the resulting feature space often still results in practically infeasible time and space complexity for distance-based learning methods. Time and space can, however, be significantly reduced by using *min-hashing* (Broder, 1997), an elegant and powerful probabilistic technique for the approximation of Jaccard-similarity. For the feature set formed by the set of all paths up to a *constant* length, min-hashing has already been applied to graph similarity estimation by performing the embedding *explicitly* (Teixeira et al, 2012). We show for the more general case of tree patterns of *arbitrary* size that the min-hash sketch of a given graph can be computed without calculating the embedding explicitly. We utilize the fact that we are interested in the first occurrence of a pattern in some permutation of the pattern set; once we have found it, we can stop the calculation, as all patterns after it are irrelevant for min-hashing. Beside this straightforward speed-up of the algorithm, the computation of the min-hash sketch can further be accelerated by utilizing once more the anti-monotonicity of subgraph isomorphism on the pattern set. These facts allow us to define a linear order on the patterns to be evaluated and to avoid redundant subtree isomorphism tests.

The experimental results presented in Section 5 clearly demonstrate that using our technique, the number of subtree isomorphism tests can dramatically be reduced with respect to the min-hash algorithm performing the embedding explicitly. It is natural to ask how the predictive performance of the approximate similarities compares to the exact ones. We show for molecular benchmark graph datasets that even for a few random spanning trees per chemical compound, remarkable precisions of the active molecules can be obtained by taking the k nearest neighbors of an active compound for $k = 1, \ldots, 100$ and that these precision values are close to those obtained by the *full* set of frequent subtrees. In a second experimental setting, we analyze the predictive power of support vector machines using our approximate similarities and show that it compares to that of state-of-the-art related methods. The stability of our incomplete probabilistic technique is explained by the fact that a subtree generated by our method is frequent not only with respect to the training set, but, with high probability, also with respect to the set of spanning trees of a graph.

Parts of this paper have already been published in (Welke et al, 2016b) and (Welke et al, 2016a). In particular, our probabilistic subtree mining method already appeared in (Welke et al, 2016b) while the min-hash sketch computation was presented in (Welke et al, 2016a). In addition to the results of these two papers we propose two novel algorithms to efficiently compute the full feature embeddings of arbitrary graphs with respect to a set of tree patterns. Using the anti-monotonicity of the pattern matching operator on the pattern poset, these novel algorithms significantly accelerate the computation of the complete embedding vector in practice.

The rest of the paper is organized as follows: We introduce the necessary notions and fix the notation in Section 2. Our probabilistic frequent subtree mining algorithm is presented in Section 3. The problem of embedding unseen graphs into (probabilistic frequent) subtree feature spaces is studied in Section 4. We report extensive experimental results with the proposed probabilistic technique in Section 5. Finally, in Section 6 we conclude and mention some interesting directions for further research.

2 Notions

In this section we collect the necessary notions and fix the notation. The set $\{1, \ldots, n\}$ will be denoted by $[n]$ for all $n \in \mathbb{N}$. The following basic concepts from graph theory are standard (see, e.g., Diestel, 2012). An *undirected* (resp. *directed*) *graph* G is a pair (V, E), where V (the vertex set) is a finite set and E (the edge set) is a subset of the family of 2-subsets of V (resp. $E \subseteq V \times V$). The set of vertices (resp. edges) of a graph G is denoted by $V(G)$ (resp. $E(G)$). When G is clear from the context, n (resp. m) denotes $|V(G)|$ (resp. $|E(G)|$). Unless otherwise stated, by graphs we mean undirected graphs. A *forest* is a graph that contains no cycle; an *unrooted* (or *free*) tree is a connected forest. A subgraph H of G is a graph with $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$. A spanning tree T of a *connected* graph G is a subgraph of G with $V(T) = V(G)$ that is a tree. For simplicity, we restrict the description of our method to unlabeled connected graphs and discuss in Section 6 how our results can be generalized to disconnected graphs.

Among the classical pattern embedding (or matching) operators, subgraph isomorphism is the most widely used one in frequent pattern mining. For this reason, in the next section we will present our method also for *subgraph isomorphism.* Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be undirected graphs. They are *isomorphic* if there exists a bijection $\varphi : V_1 \to V_2$ with $\{u, v\} \in E_1$ if and only if $\{\varphi(u), \varphi(v)\} \in E_2$ for all $u, v \in V_1$. G_1 is *subgraph isomorphic* to G_2, denoted $G_1 \preccurlyeq G_2$, if G_2 has a subgraph that is isomorphic to G_1; $G_1 \prec G_2$ denotes that $G_1 \preccurlyeq G_2$ and G_1 is not isomorphic to G_2. It is a well-known fact that the subgraph isomorphism problem is NP-complete. This negative result holds even for the case that the patterns are restricted to trees.

For any finite graph class $\mathcal{F}$ containing no two isomorphic graphs, $(\mathcal{F}, \preccurlyeq)$ is a partially ordered set (or *poset*). Since $\mathcal{F}$ is finite, $(\mathcal{F}, \preccurlyeq)$ can be represented by a *directed acyclic* graph $(\mathcal{F}, E)$ with $(T_1, T_2) \in E$ if and only if $T_1 \prec T_2$ and there is no $T \in \mathcal{F}$ with $T_1 \prec T \prec T_2$ for all $T_1, T_2 \in \mathcal{F}$. We will make use of the fact that any directed acyclic graph has at least one topological order. In particular, a *topological order* on $(\mathcal{F}, E)$ is a total order $(\mathcal{F}, \sqsubset)$ on $\mathcal{F}$ satisfying the following property: For all $T_1, T_2 \in \mathcal{F}$, $T_1 \sqsubset T_2$ whenever $(T_1, T_2) \in E$ (or equivalently, $T_1 \preccurlyeq T_2$).

A common way of defining the similarity between two graphs is to take the Jaccard-similarity of their images in the Hamming-cube $\{0, 1\}^{|\mathcal{F}|}$ spanned by the elements of some finite feature set $\mathcal{F}$. The binary feature vectors can then be regarded as the incidence vectors of subsets of $\mathcal{F}$. Given two binary feature vectors $\mathbf{f_1}$ and $\mathbf{f_2}$ representing the sets S_1 and S_2, respectively, their *Jaccard-similarity* is defined by

$$\textsc{Sim}_{\text{Jaccard}}(\mathbf{f_1}, \mathbf{f_2}) := \textsc{Sim}_{\text{Jaccard}}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

with $\textsc{Sim}_{\text{Jaccard}}(\emptyset, \emptyset) := 0$ for the degenerate case. As long as the feature vectors are low dimensional (i.e., $|\mathcal{F}|$ is small), the Jaccard-similarity can quickly be calculated. If, however, they are high dimensional, it can be approximated with the following fast probabilistic technique based on *min-hashing* (Broder, 1997): For a permutation π of $\mathcal{F}$ and feature vector $\mathbf{f}$, define $h_\pi(\mathbf{f})$ to be the index of the first entry with value 1 in the permuted order of $\mathbf{f}$. One can show that the following correspondence holds for the feature vectors $\mathbf{f_1}$ and $\mathbf{f_2}$ above (see Broder, 1997, for the details):

$$\textsc{Sim}_{\text{Jaccard}}(S_1, S_2) = \Pr\left[h_\pi(\mathbf{f_1}) = h_\pi(\mathbf{f_2})\right] \quad ,$$

where the probability is taken by selecting π uniformly at random from the set of all permutations of $\mathcal{F}$. This allows for the following approximation of the Jaccard-similarity between $\mathbf{f}_1$ and $\mathbf{f}_2$: Generate a set $\pi_1, \ldots, \pi_K$ of permutations of the feature set uniformly and independently at random and return K'/K, where K' is the number of permutations π_i with $h_{\pi_i}(\mathbf{f}_1) = h_{\pi_i}(\mathbf{f}_2)$. The approximation of the Jaccard-similarity with min-hashing results in a fast algorithm if the embedding into the feature space can be computed quickly.

Algorithm 1 Probabilistic Frequent Subtree Mining

```
input: graph database $\mathcal{D} \subseteq \mathcal{G}$, frequency threshold $\theta \in (0, 1]$, and $k > 0$ integer
output: a random subset of the set of frequent subtrees of $\mathcal{D}$
1: $\mathcal{D}' := \emptyset$
2: for all $G \in \mathcal{D}$ do
3:     sample $k$ spanning trees of $G$ uniformly at random
4:     add the forest $\mathfrak{S}_k(G)$ of those trees to $\mathcal{D}'$
5: list all subtrees with frequency at least $\theta$ in $\mathcal{D}'$
```

3 Mining Probabilistic Frequent Subtrees

In this section we define *probabilistic frequent subtree* feature spaces, i.e., feature spaces spanned by certain random subsets of frequent subtrees, and discuss some of their computational aspects. In the next section we present fast methods for computing total and partial embeddings of graphs into probabilistic frequent subtree feature spaces. As mentioned earlier, we consider only Hamming (or binary) feature spaces, i.e., the underlying instance space is mapped to the vertices of the d-dimensional unit hypercube, where d is the cardinality of the feature set considered.

To arrive at the definition, consider first the *exact* frequent subtree feature space spanned by *all* frequent subtrees of a graph dataset. Working with such feature spaces raises the following two computational problems:

(P1) The Frequent Subtree Mining Problem: *Given* a finite set $\mathcal{D}$ of graphs and a frequency threshold $\theta \in (0, 1]$, *generate* the set $\mathcal{F}$ of frequent *trees*, i.e., *all* trees T with $|\{G \in \mathcal{D} : T \preccurlyeq G\}|/|\mathcal{D}| \geq \theta$.
(P2) The Subtree Isomorphism Problem: *Given* a tree T and a graph G, *decide* whether or not $T \preccurlyeq G$.

The second problem appears in the support counting step of all algorithms solving (P1) with the generate-and-test paradigm. It is also essential to the step of computing the embedding of unseen graphs into the Hamming feature space spanned by $\mathcal{F}$. Since we have no restrictions on $\mathcal{D}$ and G, both problems above are computationally intractable. In particular, unless P = NP, (P1) cannot be solved in output polynomial time (Horváth et al, 2007) and (P2) is NP-complete. To overcome these limitations, we give up the demand on the completeness of (P1) and that on the correctness of the subtree isomorphism test for (P2), resulting in practically effective algorithms.

Regarding the relaxation of (P1), for each graph $G \in \mathcal{D}$ we consider a forest $\mathfrak{S}_k(G)$ formed by the vertex disjoint union of k *random* spanning trees of G and solve (P1) for the forest database obtained (cf. Algorithm 1). The Hamming feature space spanned by the output patterns will be referred to as *probabilistic frequent subtree* (PFS) feature space. In this way, we effectively reduce the problem of mining frequent subtrees in *arbitrary* graphs to that in forests. In contrast to the computational intractability of (P1), this relaxed problem can be solved with polynomial delay (see, e.g., Chi et al, 2005; Horváth and Ramon, 2010). In addition to $\mathcal{D}$ and θ in problem (P1), the input contains an additional parameter $k \in \mathbb{N}$ as well (cf. Algorithm 1). It specifies the upper bound on the number of spanning trees to be generated for the transaction graphs. Clearly, for any $\mathcal{D}$, θ, and k, Algorithm 1 is *sound*, i.e, its output is always a subset of the set of frequent trees in $\mathcal{D}$. However, it will not necessarily find all frequent patterns, i.e., it is *incomplete* in general. Thus, on the one hand we obtain a polynomial delay mining algorithm that is fast for small values of k, on the other hand, however, we disregard frequent patterns. Another advantage of our technique is that it assumes neither explicitly nor implicitly any structural restriction on the input graphs. A probabilistic justification of this approach together with an appropriate choice of k is given below. As we empirically demonstrate in Section 5 on various datasets, the predictive and retrieval power of probabilistic frequent subtree features are very close to those of exact ones.

Regarding the relaxation of (P2), we allow the algorithm deciding subgraph isomorphism to be *incorrect* by using a similar probabilistic approach as above. More precisely, for a tree T and a graph G, we generate a random forest $\mathfrak{S}_k(G)$ as in Algorithm 1 and regard T as a subtree of G if $T \preccurlyeq \mathfrak{S}_k(G)$. In what follows, the relation $T \preccurlyeq \mathfrak{S}_k(G)$ will be referred to as T *probabilistically matches* G. Clearly, T is subgraph isomorphic to G whenever it probabilistically matches G; otherwise T may or may not be subgraph isomorphic to G. Thus, we decide subtree isomorphism with one-sided error. Note that the relaxation on (P2) is used only for the embedding of unseen graphs, as the relaxation on (P1) implies that we have to calculate the support count of a tree in a database of forest transactions, which can be done in polynomial time.

In the application of probabilistic frequent subtrees, the incompleteness of Algorithm 1 and the incorrectness of the probabilistic embedding sketched above raise two important questions:

1. How *stable* is the output of Algorithm 1 and what is its *recall* with respect to *all* frequent subtrees? (Note that precision is always 1 for the soundness of the algorithm.)
2. How good is the predictive performance of probabilistic frequent subtrees?

Regarding the first question, we show in Section 5 on artificial and real-world chemical graph datasets that (i) the output is very stable even for $k = 1$ and (ii) more than 75% of the frequent patterns can be recovered by using only $k = 10$ random spanning trees per graph. The high stability and recall results together indicate that the probabilistic embedding of G calculated by the above method has a small Hamming distance to the exact one defined by the set of *all* frequent subtrees.

Regarding the second question above, we experimentally show in Section 5 on different real-world benchmark chemical graph datasets that the predictive performance of our probabilistic approach is comparable to those obtained by the FSG algorithm (Deshpande et al, 2005), using correct embedding. This holds not only for the set of all frequent trees, but also for the full set of frequent *subgraphs*.

3.1 μ-Important Subtrees

The rationale behind our probabilistic technique is as follows. For a graph G, let $\mathfrak{S}(G)$ be the set of *all* spanning trees of G. For the remainder of this section, we will regard $\mathfrak{S}_k(G)$ as a *set* of k spanning trees of G. Note that this is equivalent to considering it as a forest in the following sense: There exists a spanning tree

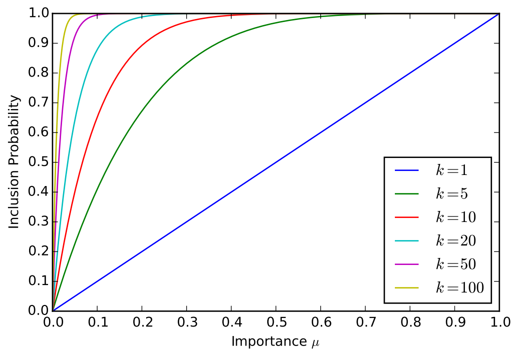

Fig. 1: The function $1-(1-\mu)^k$ for different values of k.

$S \in \mathfrak{S}_k(G)$ (as a set) such that $H \preccurlyeq S$ if and only if $H \preccurlyeq \mathfrak{S}_k(G)$ (as a forest). Using these notions, a tree T is *μ-important* in G if

$$\frac{|\{S \in \mathfrak{S}(G) : T \preccurlyeq S\}|}{|\mathfrak{S}(G)|} \geq \mu \enspace .$$

Thus, the probability that a μ-important tree in G is subtree isomorphic to a spanning tree of G generated uniformly at random is at least μ. Notice that $\mu = 1$ for any subtree of the forest formed by the set of bridges of G (i.e., by the edges that do not belong to any cycle in G). Let $\mathfrak{S}_k(G)$ denote a sample of k spanning trees of G generated independently and uniformly at random and let T be a μ-important tree in G. Then

$$\Pr\left[\exists S \in \mathfrak{S}_k(G) \text{ such that } T \preccurlyeq S\right] \geq 1-(1-\mu)^k \enspace . \tag{1}$$

The bound in (1) implies that for any graph G and μ-important tree pattern T in G for some $\mu \in (0,1]$, and for any $\delta \in (0,1)$,

$$\Pr\left[\exists S \in \mathfrak{S}_k(G) \text{ such that } T \preccurlyeq S\right] \geq 1-\delta \tag{2}$$

whenever

$$k \geq \frac{1}{\mu} \ln \frac{1}{\delta} \enspace , \tag{3}$$

where (3) is obtained from (1) and (2) by the inequality $1-x \leq e^{-x}$. (See, also, Figure 1 for the function $1-(1-\mu)^k$ for different values of k). Thus, if k is appropriately chosen, we have a probabilistic guarantee in terms of the confidence parameter δ that all μ-important tree patterns will be considered with high probability. Putting the three facts above together, we have the following claim:

Proposition 1 *For any graph G, let T be a μ-important tree in G for some $\mu \in (0,1]$ and let $\delta \in (0,1)$. Then for any $k \geq \frac{1}{\mu} \ln \frac{1}{\delta}$,*

$$\Pr\left[T \preccurlyeq \mathfrak{S}_k(G)\right] \geq 1-\delta \enspace .$$

As an example, 20 random spanning trees suffice to correctly process a 0.15-important tree pattern with probability 0.95. Clearly, a smaller value of μ results in a larger feature set.

3.2 Implementation Issues and Runtime Analysis

Line 3 of Algorithm 1 can be implemented using the algorithm of Wilson (1996), which has an expected runtime of $O\left(n^3\right)$ in the worst case, where n is the number of vertices in G. In fact, it is conjectured to be much smaller for most graphs (Wilson, 1996). Thus, the sampling step of our algorithm runs in expected $O\left(kn^3\right)$ time. If we do not require the spanning trees to be drawn uniformly, we can improve on this time and achieve a deterministic $O\left(km \log n\right)$ runtime, where m denotes the number of edges in G. This is achieved by choosing a *random* permutation of the edge set of a graph and then applying Kruskal's minimum spanning tree algorithm using this edge order. It is not difficult to see that this technique can generate random spanning trees with non-uniform probability. Each spanning tree has, however, a nonzero probability of being selected in this way. As our experimental results on molecular graphs of pharmacological compounds show, non-uniformity has no significant impact on the predictive performance of the graph kernel obtained.

For a practical improvement of the runtime of our algorithm, we note that some spanning trees in $\mathfrak{S}_k(G)$ might be redundant: Since isomorphic spanning trees yield the same subtrees, it suffices to keep only one spanning tree from each equivalence class. The set of all sampled spanning trees in $\mathfrak{S}_k(G)$ *up to isomorphism* can be computed from $\mathfrak{S}_k(G)$ using some canonical string representation for trees and a prefix tree as data structure (see, e.g., Chi et al, 2005, for more details on canonical string representations for trees). For each tree, this can be done in $O\left(n \log n\right)$ time by computing first the tree center and then applying a canonical string algorithm for rooted trees as in (Chi et al, 2005). These canonical strings are then stored in and retrieved from a prefix tree in time linear in their size. We implemented this method as an extension of Line 4.

Finally we note that for Line 5, we can use almost any one of the existing algorithms generating frequent connected subgraphs (i.e., subtrees) from forest databases (see, e.g., Chi et al, 2005, for an overview on this topic). We recall that most practical systems actually do not guarantee polynomial delay (or even output polynomial time).

4 Embeddings into Subtree Feature Spaces

Typical applications of feature spaces for learning and information retrieval tasks rely on calculating some similarity measure between points in the underlying feature space. Many of such algorithms require the input data to be embedded into the feature space *explicitly*. In this section we consider explicit embeddings into Hamming feature spaces spanned by tree patterns. Thus, the results of this section can naturally be applied to the case of PFS feature spaces as well. We distinguish two scenarios. The first one is when the feature vector of a graph G must be computed *completely*, i.e., for all tree patterns T in the feature set it must be decided whether $T \preccurlyeq G$, or not. The second one is when it suffices to compute the feature vector *partially*. For the latter scenario we have a particular interest in the application of the *min-hashing* algorithm (Broder, 1997) to PFS feature spaces. This algorithm computes arbitrary close approximations of the *Jaccard-similarity* between two binary vectors (or equivalently, between two sets). The Jaccard-similarity is a widely used measure applied among others in information retrieval, kernel methods, clustering, etc. Our motivation of considering the Jaccard-similarity is that it has been applied successfully also in graph mining (see, e.g., Teixeira et al, 2012).

For both scenarios we present algorithms that significantly reduce the number of pattern matching evaluations over the standard brute-force algorithms and hence accelerate the speed of explicit embeddings. We utilize the fact that subgraph isomorphism induces a (natural) partial order on trees. In Section 4.1 we first discuss complete embeddings and then, in Section 4.2, partial embeddings for min-hashing in subtree feature spaces.

4.1 Complete Embeddings into Subtree Feature Spaces

In this section we deal with the problem of computing *complete* embeddings into Hamming feature spaces spanned by tree patterns. The algorithms presented can be applied to the special case of probabilistic frequent subtrees as well. More precisely, we consider the following problem:

(P3) Exact Embedding into Subtree Feature Spaces: *Given* a set $\mathcal{F}$ of tree patterns and a graph G, *compute* the incidence vector $\mathbf{f}$ of the set $\{T \in \mathcal{F} : T \preccurlyeq G\}$.

We regard $\mathcal{F}$ as the poset $(\mathcal{F}, \preccurlyeq)$ and assume without loss of generality that the empty tree $\perp$ is an element of $\mathcal{F}$ and that $\mathcal{F}$ is closed under taking subgraphs modulo isomorphism. We also assume that the poset $(\mathcal{F}, \preccurlyeq)$ is provided as a directed acyclic graph $F = (\mathcal{F}, E)$ such that for all $T_1, T_2 \in \mathcal{F}$ holds $(T_1, T_2) \in E$ if and only if $|V(T_2)| = |V(T_1)| + 1$, and $T_1 \preccurlyeq T_2$.

Clearly, (P3) is NP-complete. Therefore, we relax it in a way similar to the relaxation of (P2) in the previous section and approximate the incidence vector $\mathbf{f}$ of $\{T \in \mathcal{F} : T \preccurlyeq G\}$ by the incidence vector $\mathbf{f}'$ of $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}_k(G)\}$. While this relaxation makes (P3) computationally tractable, each invocation of the probabilistic matching operator adds a non-negligible amount of work of complexity $O\left(kn^{2.5}/\log n\right)$, corresponding to the complexity of subgraph isomorphism from trees into forests (Shamir and Tsur, 1999). This super-quadratic complexity motivates us to minimize the number of calls of the probabilistic matching operator while computing $\mathbf{f}'$. We present three algorithms for computing $\mathbf{f}'$ that significantly reduce the number of probabilistic pattern matching evaluations in practice with respect to the brute-force algorithm calling this operator $|\mathcal{F}|$-times. We note that all three methods can be applied without any change to other embedding operators as well, as long as they are anti-monotone with respect to the partial order induced on $\mathcal{F}$.

Notice that computing $\mathbf{f}'$ for G is equivalent[1] to computing all frequent subtrees of the graph database $\mathcal{D} = \{G\}$ for frequency threshold 1, where the pattern language is restricted to $\mathcal{F}$. As a baseline approach we generate the set of matching patterns with *levelwise search* (Mannila and Toivonen, 1997), i.e., with breadth-first search traversal of F starting at $\perp$ and pruning by utilizing the anti-monotonicity of the embedding operator on $(\mathcal{F}, \preccurlyeq)$. This algorithm, referred to as Levelwise, evaluates the probabilistic pattern matching operator exactly for all patterns that probabilistically match G and for all patterns in the negative border, i.e., which do not probabilistically match G, but all of their subgraphs in $\mathcal{F}$ do.

[1] In the sense that there exist polynomial time reductions between the two problems.

Levelwise is optimal in the sense that it evaluates only those non-matching patterns that are in the negative border (Mannila and Toivonen, 1997). However, one call to the embedding operator is required for each matching pattern. As F is given explicitly, we can reduce this number by leveraging the anti-monotonicity of the embedding operator downwards as well: If a pattern T matches G all of its subgraphs match G as well and therefore need not be evaluated explicitly. Note that the number of such subgraphs can be exponential in the size of T. In F, there is a directed path from each such pattern to T. Hence, a traversal strategy that visits large patterns before the evaluation of all of their subgraphs can reduce the number of calls to the embedding operator. This idea can be implemented in several ways; we will present two different such traversal strategies.

We first consider a simple *greedy* search instead of the levelwise traversal of F, that already works quite well in practice, as we will see in Section 5.2. It reduces the number of embedding operator evaluations on matching patterns by traversing the supergraphs of a matching pattern before the evaluation of their subgraphs. However, it might evaluate non-matching patterns as well that are beyond the negative border. Our experiments indicate that the number of matches that are *not* explicitly evaluated usually outweighs that of non-matching patterns beyond the negative border that are evaluated by the greedy search.

Our implementation of this greedy strategy, called Greedy, is depicted in Algorithm 2. The algorithm iterates through every pattern from small to large and starts a depth-first search (DFS) traversal on each pattern for which the outcome of the embedding operator is yet unknown and backtracks once a non-matching pattern is found. While Greedy is running, it updates the state of patterns according to the anti-monotonicity of subgraph isomorphism. It encodes the value of the embedding operator on a pattern in a ternary *state* variable, which can take the values *match*, *noMatch*, and *unknown*. Due to the fact that we mark subgraphs of matching patterns as matches, it will likely happen that the state of some or all direct supergraphs of a pattern is already known during backtracking. However, the state of some other (larger) supergraphs might still be unknown. Hence a single invocation of a DFS starting at $\perp$ would not suffice to guarantee that every pattern has been visited. Note that the state of each pattern changes from *unknown* to either *match* (Line 10) or *noMatch* (Line 12) whenever the embedding operator is evaluated in Line 8. Due to Line 7 this means that the operator is evaluated at most once for each pattern. The remaining runtime of Greedy is bounded by $O(|E(F)|)$: The recursion on the edges only happens when the embedding operator is evaluated on a pattern, which can happen only once as we have seen above. Second, Lines 10 and 12 can be implemented by a BFS or DFS that only traverses patterns in F (respectively the reverse graph of F) whose *state* is *unknown*.

As a second idea to use anti-monotonicity for pruning non-matching patterns as well as matching patterns, we propose BinarySearch described in Algorithm 3. The algorithm iteratively searches longest paths in the part of the directed graph F for which the value of the embedding operator is still unknown. Such a directed path P in F corresponds to a chain in the partial order $(\mathcal{F}, \preccurlyeq)$. Due to the anti-monotonicity of the embedding operator for a fixed graph G there are three cases: (1) All patterns in P match G, (2) no patterns in P match G, or (3) there is a unique pattern T in P whose descendants in P all match and whose successors in P all do not match G. BinarySearch regards such a path P in F as an array and searches T in $O(\log |V(P)|)$ time, all the while maintaining the deducible state of

Algorithm 2 Greedy

```
Input: A graph $G$ and a directed graph $F = (\mathcal{F}, E)$ representing the poset $(\mathcal{F}, \preccurlyeq)$
Output: $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}_k(G)\}$

 1: set $state[T] := unknown$ for all $T \in \mathcal{F}$
 2: fix $\mathfrak{S}_k(G)$
 3: for $T \in \mathcal{F}$ in a topological order do
 4:     DFS($\mathfrak{S}_k(G), T, state, F$)
 5: return $\{T \in \mathcal{F} : state[T] = match\}$

 6: procedure DFS($\mathfrak{S}_k(G), T, state, F$)
 7:     if $state[T] = unknown$ then
 8:         if $T \preccurlyeq \mathfrak{S}_k(G)$ then
 9:             for $(T, T') \in \mathcal{N}(T)$ do DFS($\mathfrak{S}_k(G), T', state, F$)
10:             set $state[T'] = match$ for all $T'$ that can reach $T$ in $F$.
11:         else
12:             set $state[T'] = noMatch$ for all $T'$ reachable from $T$ in $F$.
```

Algorithm 3 BinarySearch

```
Input: A graph $G$ and a directed graph $F = (\mathcal{F}, E)$ representing the poset $(\mathcal{F}, \preccurlyeq)$
Output: $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}_k(G)\}$

 1: set $state[T] := unknown$ for all $T \in \mathcal{F}$
 2: fix $\mathfrak{S}_k(G)$
 3: for $T \in \mathcal{F}$ in a topological order do
 4:     if $state[T] = unknown$ then
 5:         let $P$ be a longest path in $F$ starting at $T$ such that
                $\forall T' \in V(P)\ state[T'] = unknown$
 6:         BinarySearch($\mathfrak{S}_k(G), P, state, F$)
 7: return $\{T \in \mathcal{F} : state[T] = match\}$

 8: procedure BinarySearch($\mathfrak{S}_k(G), P, state, F$)
 9:     $min := 1$
10:     $max := length(P)$
11:     while $min \leq max$ do
12:         let $i := \lfloor (min + max)/2 \rfloor$
13:         let $T := P[i]$
14:         if $state[T] = unknown$ then
15:             if $T \preccurlyeq \mathfrak{S}_k(G)$ then
16:                 set $state[T'] = match$ for all $T'$ that can reach $T$ in $F$.
17:             else
18:                 set $state[T'] = noMatch$ for all $T'$ reachable from $T$ in $F$.
19:         if $state[T] = match$ then
20:             $min := i + 1$
21:         else
22:             $max := i - 1$
```

the patterns in F. It is noteworthy that long paths are beneficial for the runtime of this algorithm, as the difference between $\log x$ and x increases with growing x.

Using similar arguments as in the discussion of Algorithm 2 above, one can show that Algorithm 3 is correct and evaluates the matching operator at most once for each tree pattern. A longest path starting at a given pattern in the part of F where the *state* of patterns is *unknown* can be implemented by a DFS. However, in contrast to the traversal of F to maintain the *state*, there is no guarantee that a given edge is traversed at most once and in total no better bound than

$O\left(|E(F)|^2 + |V(F)| \cdot f(G)\right)$ can be given for the runtime of Algorithm 3. During our empirical evaluation, however, we saw that the runtime of BinarySearch was still dominated by the calls to the matching operator.

4.2 Min-Hashing in Subtree Feature Spaces

In this section we discuss another application of probabilistic frequent subtrees by considering the problem of computing the *Jaccard-similarity* between feature vectors in the Hamming space spanned by probabilistic frequent subtrees. The Jaccard-similarity (often also called Tanimoto kernel) is a well-established and commonly used similarity measure in subgraph feature spaces (see, e.g., Teixeira et al, 2012; Gärtner et al, 2003). Despite the redundancies among the subgraph features it has a number of successful practical applications (see, e.g., Willett, 2006, for its application in computational chemistry). More precisely, we consider the following problem:

(P4) The Jaccard-Similarity Problem: *Given* a set $\mathcal{F}$ of probabilistic frequent subtrees and two graphs G_1, G_2 with random forests $\mathfrak{S}_k(G_1), \mathfrak{S}_k(G_2)$, respectively, *compute* the Jaccard-similarity

$$\textsc{Sim}_{\text{Jaccard}}(\mathbf{f}_1, \mathbf{f}_2) \ ,$$

where $\mathbf{f}_i$ is the incidence vector of the set of trees in $\mathcal{F}$ that are subgraph isomorphic to $\mathfrak{S}_k(G_i)$ $(i = 1, 2)$.

Instead of using the naive brute-force algorithm, i.e., performing first the explicit embeddings of $\mathfrak{S}_k(G_1)$ and $\mathfrak{S}_k(G_2)$ into the Hamming space spanned by $\mathcal{F}$ and calculating then the exact value of $\textsc{Sim}_{\text{Jaccard}}(\mathbf{f}_1, \mathbf{f}_2)$, we follow Broder's probabilistic *min-hashing* technique (Broder, 1997) sketched in Section 2. Though the description below is restricted to tree shaped patterns, the approach can naturally be adapted to any partially ordered pattern language and anti-monotone embedding operator.

Min-hashing was originally applied to text documents using q-shingles as features (i.e., sequences of q contiguous tokens for some $q \in \mathbb{N}$), implying that one can calculate the explicit embedding in linear time by shifting a window of size q through the document to be embedded. In contrast, a naive algorithm embedding the forest $\mathfrak{S}_k(G)$ generated for a graph of size n into the Hamming space spanned by $\mathcal{F}$ would require $O\left(|\mathcal{F}|kn^{2.5}/\log n\right)$ time by using the subtree isomorphism algorithm of Shamir and Tsur (1999). This is practically infeasible when the cardinality of $\mathcal{F}$ is large, which is typically the case. Another difference between the two application scenarios is that while the set of q-shingles for text documents forms an anti-chain (i.e., the q-shingles are pairwise incomparable), subgraph isomorphism induces a natural *partial order* on $\mathcal{F}$. The transitivity of subgraph isomorphism allows us to safely ignore features from $\mathcal{F}$ that do not influence the outcome of min-hashing, resulting in a much faster algorithm.

To adapt the min-hashing technique to the situation that the patterns form a nontrivial partial order and embedding computation is expensive, we proceed as follows: In a preprocessing step, directly after the generation of $\mathcal{F}$, we generate K random permutations $\pi_1, \ldots, \pi_K : \mathcal{F} \to [|\mathcal{F}|]$ of $\mathcal{F}$ (see Broder et al, 2000, for the details) and fix them for computing the min-hash values that will be used for similarity query evaluations (cf. Section 2). We assume that our algorithm will be applied to a large number of transaction graphs and that the runtime of computing the embeddings will dominate the overall time complexity. Hence we can allow preprocessing time and space that is polynomial in the size of the pattern set $\mathcal{F}$. Therefore, we explicitly compute and store $\pi_1, \ldots, \pi_K$, and do not apply any implicit representations of them. This is particularly true, as we compute $\mathcal{F}$ explicitly in the preprocessing step and spend time that is polynomial in $\mathcal{F}$ anyway.

For a graph G with a random forest $\mathfrak{S}_k(G)$ and for a permutation π of $\mathcal{F}$, let

$$h_\pi(G) = \operatorname*{argmin}_{T \in \mathcal{F}} \{\pi(T) : T \preccurlyeq \mathfrak{S}_k(G)\} \quad .$$

The *sketch* of G with respect to $\pi_1, \ldots, \pi_K$ is then defined by

$$\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G) = (h_{\pi_1}(G), \ldots, h_{\pi_K}(G)) \quad .^2$$

The rest of this section is devoted to the following problem: Given $\pi_1, \ldots, \pi_K$, G with forest $\mathfrak{S}_k(G)$ as above, compute $\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G)$. The first observation that leads to an improved algorithm computing $\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G)$ is that for any $i \in [K]$, π_i may contain trees that can never be the first matching patterns according to π_i. Indeed, suppose we have two patterns $T_1, T_2 \in \mathcal{F}$ with $T_1 \preccurlyeq T_2$ and $\pi_i(T_1) < \pi_i(T_2)$. Then for $\mathfrak{S}_k(G)$ we have either

1. $T_1 \preccurlyeq \mathfrak{S}_k(G)$ and hence T_2 is not the first matching pattern in π_i or
2. $T_1 \not\preccurlyeq \mathfrak{S}_k(G)$ and hence $T_2 \not\preccurlyeq \mathfrak{S}_k(G)$ by the transitivity of subgraph isomorphism.

For both cases, T_2 will never be the first matching pattern according to π_i and can therefore be omitted from this permutation. Algorithm 4 implements this idea for a single permutation π of $\mathcal{F}$. It filters the permutation π and returns an *evaluation sequence* σ by traversing π in order and removing all patterns for which Case 1 or 2 hold. This evaluation sequence can be substituted for the permutation to compute the min-hash values, as stated in the following lemma:

Lemma 1 *Let $\sigma = \langle T_1, \ldots, T_l \rangle$ be the output of Algorithm 4 for a permutation π of $\mathcal{F}$. Then, for any graph G with $\mathfrak{S}_k(G)$,*

$$h_\pi(G) = \operatorname*{argmin}_{T_i \in \sigma} \{i : T_i \preccurlyeq \mathfrak{S}_k(G)\} \quad .$$

Proof Let $H = h_\pi(G)$, i.e., $H = \operatorname{argmin}_{T \in \mathcal{F}} \{\pi(T) : T \preccurlyeq \mathfrak{S}_k(G)\}$ and let $H' = \operatorname{argmin}_{T_i \in \sigma}\{i : T_i \preccurlyeq \mathfrak{S}_k(G)\}$. The output of Algorithm 4 only contains elements of π (Line 7) and maintains their order, i.e., if T comes before T' in σ, then $\pi(T) < \pi(T')$. Hence, $\pi(H) \leq \pi(H')$. It only remains to show that H is contained in σ. If H is not appended to σ in Line 7 then *visited*$(H) = 1$ must have held in Line 4. Hence, there must have been a T before H in π such that $T \preccurlyeq H$. However, $H \preccurlyeq \mathfrak{S}_k(G)$ implies $T \preccurlyeq \mathfrak{S}_k(G)$, contradicting our assumption $H = h_\pi(G)$. □

Algorithm 4 runs in time $O(|\mathcal{F}|)$. Loop 5 can be implemented by a DFS that does not recurse on the visited neighbors of a vertex. In this way, each edge of F is visited exactly once during the algorithm.

[2] In practice, we do not store the patterns in $\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G)$ explicitly. Instead, we define some arbitrary total order on $\mathcal{F}$ and represent each pattern by its position according to this order.

Algorithm 4 POSET-PERMUTATION-SHRINK

```
Input: directed graph $F = (\mathcal{F}, E)$ representing a poset $(\mathcal{F}, \preccurlyeq)$ and permutation $\pi$ of $\mathcal{F}$
Output: evaluation sequence $\sigma = \langle T_1, \ldots, T_l \rangle \in \mathcal{F}^l$ for some $0 < l \leq |\mathcal{F}|$ with $\pi(T_i) < \pi(T_j)$
    for all $1 \leq i < j \leq l$

1: Initialize $\sigma :=$ empty list
2: Initialize $visited(T) := 0$ for all $T \in \mathcal{F}$
3: for all $T \in \mathcal{F}$ in the order of $\pi$ do
4:     if $visited(T) = 0$ then
5:         for all $T' \in \mathcal{F}$ (including $T$) that are reachable from $T$ in $F$ do
6:             set $visited(T') := 1$
7:         append $T$ to $\sigma$
8: return $\sigma$
```

We now turn to the computation of $\text{SKETCH}_{\pi_1,\ldots,\pi_K}(G)$ for a graph G with $\mathfrak{S}_k(G)$. A straightforward implementation of calculating $\text{SKETCH}_{\pi_1,\ldots,\pi_K}(G)$ for the evaluation sequences $\sigma_1, \ldots, \sigma_K$ computed by Algorithm 4 for $\pi_1, \ldots, \pi_K$ just loops through each evaluation sequence, stopping each time the first match is encountered. This strategy can further be improved by utilizing the fact that a pattern T may be evaluated redundantly more than once for a graph G with forest $\mathfrak{S}_k(G)$, if T occurs in more than one permutation before or as the first match. Lemma 2 below formulates necessary conditions for avoiding redundant subgraph isomorphism tests. To this end, let $|\sigma|$ denote the number of elements in an evaluation sequence σ.

Lemma 2 *Let G be a graph with $\mathfrak{S}_k(G)$ and let $\sigma_1, \ldots, \sigma_K$ be the evaluation sequences computed by Algorithm 4 for the permutations $\pi_1, \ldots, \pi_K$ of $\mathcal{F}$. Let $\mathfrak{A}$ be an algorithm that correctly computes $\text{SKETCH}_{\pi_1,\ldots,\pi_K}(G)$ by evaluating subgraph isomorphism in the pattern sequence $\Sigma = \langle \sigma_1[1], \ldots, \sigma_K[1], \sigma_1[2], \ldots, \sigma_K[2], \ldots \rangle$. Then $\mathfrak{A}$ remains correct if for all $i \in [K]$ and $j \in [|\sigma_i|]$, it skips the evaluation of $\sigma_i[j] \preccurlyeq \mathfrak{S}_k(G)$ whenever one of the following conditions holds:*

1. *$\sigma_i[j'] \preccurlyeq \mathfrak{S}_k(G)$ for some $j' \in [j-1]$,*
2. *there exists a pattern T before $\sigma_i[j]$ in Σ such that $\sigma_i[j] \preccurlyeq T$ and $T \preccurlyeq \mathfrak{S}_k(G)$,*
3. *there exists a pattern T before $\sigma_i[j]$ in Σ such that $T \preccurlyeq \sigma_i[j]$ and $T \not\preccurlyeq \mathfrak{S}_k(G)$.*

Proof If Condition 1 holds then the min-hash value for permutation π_i has already been determined. If $\sigma_i[j] \preccurlyeq T$ and $T \preccurlyeq \mathfrak{S}_k(G)$ then $\sigma_i[j] \preccurlyeq \mathfrak{S}_k(G)$ by the transitivity of subgraph isomorphism. For the same reason, if $T \preccurlyeq \sigma_i[j]$ and $T \not\preccurlyeq \mathfrak{S}_k(G)$ then $\sigma_i[j] \not\preccurlyeq \mathfrak{S}_k(G)$. Hence, if Condition 2 or 3 holds then $\mathfrak{A}$ can infer the answer to $\sigma_i[j] \preccurlyeq G$ without explicitly performing the subtree isomorphism test. □

Algorithm 5 computes the sketch for a graph G with $\mathfrak{S}_k(G)$ along the conditions formulated in Lemma 2. Similarly to the algorithms in Section 4.1 it maintains a state for all $T \in \mathcal{F}$ defined as follows: *unknown* encodes that $T \preccurlyeq G$ is unknown, *match* that $T \preccurlyeq \mathfrak{S}_k(G)$, and *noMatch* that $T \not\preccurlyeq \mathfrak{S}_k(G)$.

Theorem 1 *Algorithm 5 is correct, i.e., it returns $\text{SKETCH}_{\pi_1,\ldots,\pi_K}(G)$. Furthermore, it is non-redundant, i.e., for all patterns $T \in \mathcal{F}$, it evaluates at most once whether or not $T \preccurlyeq \mathfrak{S}_k(G)$.*

Proof The correctness is immediate from Lemmas 1 and 2. Regarding non-redundancy, suppose $T \preccurlyeq \mathfrak{S}_k(G)$ has already been evaluated for some pattern $T \in \mathcal{F}$ with

Algorithm 5 MIN-HASH SKETCH

```
Input: graph $G$ with forest $\mathfrak{S}_k(G)$, directed graph $F = (\mathcal{F}, E)$ representing a poset $(\mathcal{F}, \preccurlyeq)$
    and $K$ evaluation sequences $\sigma_1, \ldots, \sigma_K$ computed by Algorithm 4 for the permutations
    $\pi_1, \ldots, \pi_K$ of $\mathcal{F}$
Output: $\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G)$

 1: Initialize $sketch := [\bot, \ldots, \bot]$
 2: Initialize $state(T) := unknown$ for all $T \in \mathcal{F}$
 3: for $i = 1$ to $|\mathcal{F}|$ do
 4:     for $j = 1$ to $K$ do
 5:         if $|\sigma_j| \geq i \land sketch[j] = \bot$ then
 6:             if $state[\sigma_j[i]] \neq unknown$ then
 7:                 if $state[\sigma_j[i]] = match$ then $sketch[j] = \sigma_j[i]$
 8:             else if $\sigma_j[i] \preccurlyeq \mathfrak{S}_k(G)$ then
 9:                 $sketch[j] = \sigma_j[i]$
10:                 Set $state[T'] = match$ for all $T'$ that can reach $T$ in $F$
11:             else
12:                 Set $state[T'] = noMatch$ for all $T'$ reachable from $T$ in $F$
13: return $sketch$
```

$T = \sigma_i[j]$. Then, as $T \preccurlyeq T$, for any $\sigma_{i'}[j'] = T$ after $\sigma_i[j]$ in Σ either Condition 2 or 3 holds and hence $T \preccurlyeq \mathfrak{S}_k(G)$ will never be evaluated again. □

Once the sketches are computed for two graphs G_1, G_2, their Jaccard-similarity with respect to $\mathcal{F}$ can be approximated by the fraction of identical positions in these sketches. (We define the similarity of G_1 and G_2 with $\textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G_1) = \textsc{Sketch}_{\pi_1,\ldots,\pi_K}(G_2) = (\bot, \ldots, \bot)$ by 0.)

5 Experimental Evaluation

To evaluate the methods described in Sections 3 and 4, we have conducted various experiments on different real-world and artificial datasets. In Section 5.1 we first consider the probabilistic frequent subtree mining algorithm and demonstrate that, except for small molecular graphs, it outperforms ordinary frequent subgraph mining in *runtime*. We analyze also its *recall* and *stability* with respect to the *complete* set of frequent tree patterns. In Section 5.2 we then empirically investigate our algorithms from Section 4 concerning complete and partial embedding into PFS feature spaces. We evaluate their *speed* measured by the number of subtree isomorphism tests performed. We show that our methods drastically reduce this number compared to the brute-force and the levelwise algorithms discussed in Section 4. In Section 5.3 we finally evaluate the *predictive* and *retrieval* performance of probabilistic frequent subtrees applied in kernel and other distance-based methods. Our experimental results clearly show that PFS feature spaces perform only slightly worse than ordinary frequent subgraph feature spaces (using the same frequency thresholds). This minor drawback of probabilistic frequent subtrees is, however, compensated by their *high recall* and *stability*, as well as by the *superior runtime* results of our methods.

Datasets: For the experiments we have used the chemical graph datasets MUTAG, PTC, DD, NCI1, and NCI109[3], NCI-HIV[4], and ZINC[5].

- MUTAG is a dataset of 188 connected compounds labeled according to their mutagenic effect on Salmonella Typhimurium.
- PTC contains 344 connected molecular graphs, annotated with respect to the carcinogenicity in mice and rats.
- DD consists of $1,187$ protein structures, of which $1,157$ are connected. Labels differentiate between enzymes and non-enzymes.
- NCI1 and NCI109 contain $4,110$ resp. $4,127$ compounds, of which $3,530$ resp. $3,519$ are connected. Both are balanced sets of chemical molecules labeled according to their activity against non-small cell lung cancer (resp. ovarian cancer) cell lines.
- NCI-HIV consists of $42,687$ compounds, of which $39,337$ are connected. The molecules are annotated with respect to their activity against the human immunodeficiency virus (HIV). In particular, they are labeled by "active" (A), "moderately active" (M), or "inactive" (I). While the five datasets above all have a balanced class distribution, the class distribution of NCI-HIV is heavily skewed: Only 329 molecules (i.e., less than 1%) are in class A, 906 in class M, and the remaining $38,102$ in class I.
- ZINC is a subset of $8,946,757$ ($8,946,755$ connected) unlabeled, so called 'Lead-Like' molecules from the *zinc* database of purchasable chemical compounds. The molecules in this subset have a molar mass between 250 and 350 g/mol and have an average number of vertices and edges 43 and 44, respectively.
- Additionally, we generated artificial datasets consisting of unlabeled sparse graphs of varying number of vertices and edges that were generated in the Erdős-Rényi random graph model (Erdős and Rényi, 1959). The datasets generated are of different structural complexity defined by the *expected edge factor* $q = \frac{m}{n}$ (recall that n is the number of vertices and m the number of edges). For a given q, each graph G in the corresponding dataset is generated as follows: We first draw the number n of vertices uniformly at random between 2 and 50, set the Erdős-Rényi edge probability parameter $p = \frac{2q}{n-1}$, and then generate G on n vertices in the usual way with this value of p. If the resulting graph is connected, we add it to the dataset.

Since we described our methods for connected graphs, disconnected graphs have been omitted. We have restricted the maximum size of a tree pattern to 10 vertices, as this bound seemed consistently optimal for the predictive/ranking performance for all chemical datasets used in our experiments. (We stress that our method is not restricted to constant-sized tree patterns.) The same observation was reported by Teixeira et al (2012), as well. All our experiments were conducted on an Intel i7 CPU with 3.40GHz and 16GB of RAM.

[3] All these five datasets were obtained from `http://www.di.ens.fr/~shervashidze/`.
[4] `https://cactus.nci.nih.gov/`
[5] `http://zinc.docking.org/subsets/lead-like`

5.1 Mining Probabilistic Frequent Subtrees

In this section we empirically evaluate our probabilistic frequent subtree mining algorithm described in Section 3. We first present results demonstrating that it is faster up to an order of magnitude than any comparable frequent subgraph mining algorithm and that it is able to mine frequent patterns also in situations where state-of-the-art algorithms do not. We then empirically demonstrate that the recall of its output is high with respect to the set of all frequent subtrees. Finally we give empirical evidence that probabilistic frequent subtrees are stable under resampling of the random spanning trees. The establishment of these three properties of probabilistic frequent subtrees is the first step towards considering PFS feature spaces as candidates for graph classification and retrieval tasks.

Runtime We first compare the runtime of the FSG algorithm[6] (Deshpande et al, 2005) with that of our algorithm on artificial datasets and on subsets of the ZINC dataset. For our algorithm we report the combined time for sampling and frequent pattern generation. In particular, we use the method of Wilson (1996) in Line 3 of Algorithm 1 to generate random spanning trees and an efficient implementation of a frequent tree mining algorithm[7] to list frequent subtrees in Line 5.

Figure 2 shows the runtime on artificial datasets for expected edge factors q varying between 1.0 and 5.0. (Note the log scale for the y-axis.) We report average execution times over three runs for computing the set of frequent patterns and that of probabilistic frequent subtrees for various numbers of random spanning trees (k). It turns out that FSG is very sensible to the parameter q. In order to be able to get any result in reasonable time, we had to restrict the number of graphs in each dataset to 50. Still we had to terminate FSG in several cases where it took more than 24 hours (86,400s), which was consistently the case once q exceeded 1.8. Up to 50 sampled spanning trees, our probabilistic approach is always faster; for $q > 1.4$ our method still terminates in less than a second for all k, while FSG was aborted after a day without finishing.

Figure 3 reports the runtime results (in seconds) on a subset of 1,000 molecules of the ZINC dataset for FSG and for our algorithm with $k \in \{1, 5, 20, 50\}$. In contrast to the runtime on artificial datasets, our method is faster only for $k = 1$, while being slower than FSG even for the case of $k = 5$. To this end, we note that the average edge factor (cf. the definition of q), i.e., $\frac{1}{|\mathcal{D}|} \sum_{G \in \mathcal{D}} \frac{|E(G)|}{|V(G)|}$ of chemical datasets $\mathcal{D}$ is very low: It is 1.04 for both NCI-HIV and ZINC. We therefore assume that FSG is highly optimized for structurally very simple *labeled* graphs, where it has a competitive advantage over our method.

[6] `http://glaros.dtc.umn.edu/gkhome/pafi/overview`

[7] We experimented with several publicly available frequent tree mining algorithms for forest databases but, somewhat surprisingly, they were not able to beat the speed of FSG on forest datasets resulting from the spanning tree sampling step. Our implementation generates frequent trees with levelwise search. It uses the algorithm of Shamir and Tsur (1999) as the subroutine for the support counting step. In this way, we are able to guarantee pattern enumeration in incremental polynomial time. Though we used several standard optimizations (e.g., evaluating the embedding operator only on the intersection of the support sets of the parent patterns), our implementation can further be improved.

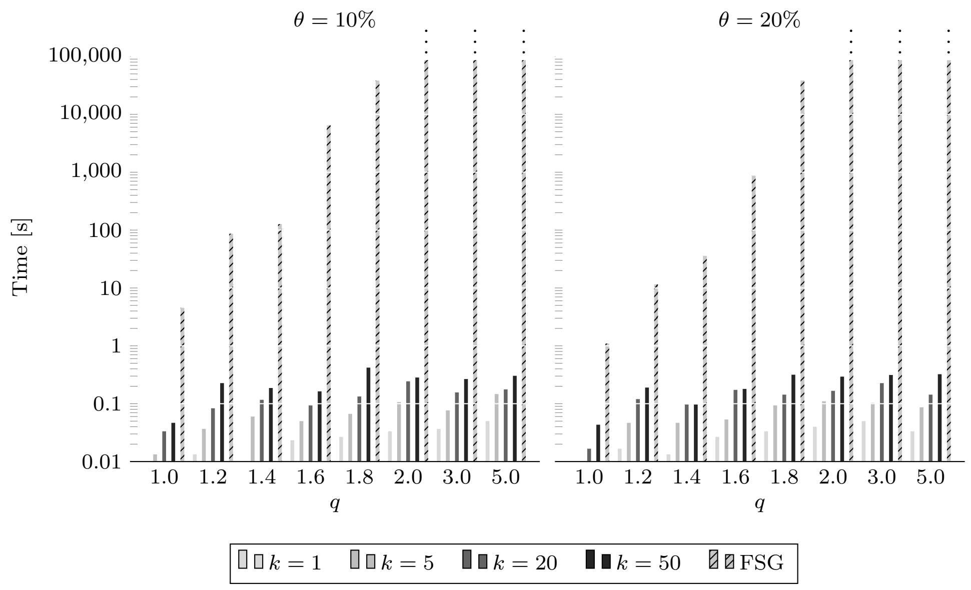

Fig. 2: Runtime of our method, compared to FSG on artificial datasets of varying expected edge factor q. Dots over bars signal that the run was terminated after 24 hours.

Recall As discussed in Section 3, for any graph database the pattern set $\mathcal{F}$ found by our probabilistic mining algorithm is a subset of all frequent subtrees $\mathcal{F}_T$, which in turn is a subset of all frequent subgraphs. We now analyze the recall of our method, i.e., the amount of frequent subtree patterns that are found when applying Algorithm 1 for various values of k and θ. To this end, let $R(k, \theta) := \frac{|\mathcal{F}|}{|\mathcal{F}_T|}$ be the fraction of θ-frequent tree patterns found by Algorithm 1 for k random spanning trees. Using the FSG algorithm, on each dataset we first compute all frequent connected patterns, including non-tree patterns as well, and then filter out all frequent subgraphs that are not trees.

Figure 4 shows the recall $R(k, \theta)$ of our method for one run on artificial datasets and for frequency thresholds 10% and 20%. It is restricted to expected edge factors $q \leq 1.8$, as FSG was not able to compute the full set of frequent patterns within a day beyond this value. Even for a single spanning tree (i.e., for $k = 1$), the recall is always above 20%; in most cases actually above 40%. The recall for $k = 5$ sampled spanning trees is drastically higher than for $k = 1$; in fact the increase in recall between $k = 5$ and $k = 50$ is much lower. This suggests that $k = 5$ might be a good compromise in the trade-off between runtime and accuracy of our method.

For NCI-HIV and ZINC, we sample 10 subsets of 100 graphs each and report the average value of $R(k, \theta)$ and its standard deviation. The results on the two datasets can be found in Table 1 for different values of k and for frequency thresholds 5%, 10%, and 20%. We have found that at least 95% of all frequent subgraphs are trees. One can also observe that the fraction of the retrieved tree patterns grows rapidly with the number of random spanning trees sampled per

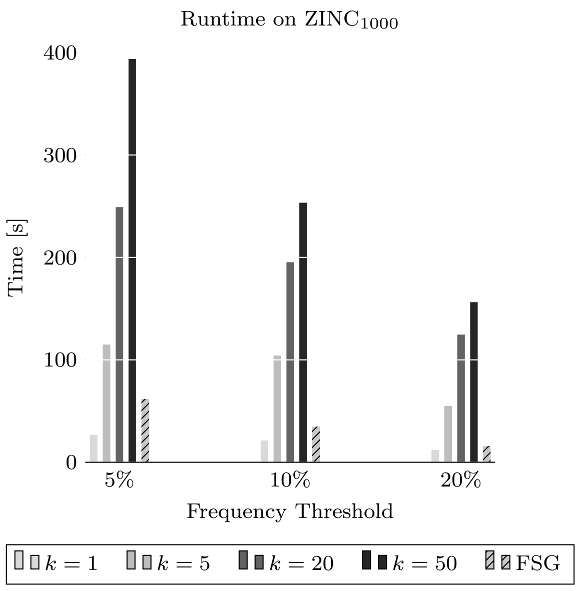

Fig. 3: Runtime results in seconds for our method and FSG, for different frequency thresholds $\theta \in \{5\%, 10\%, 20\%\}$.

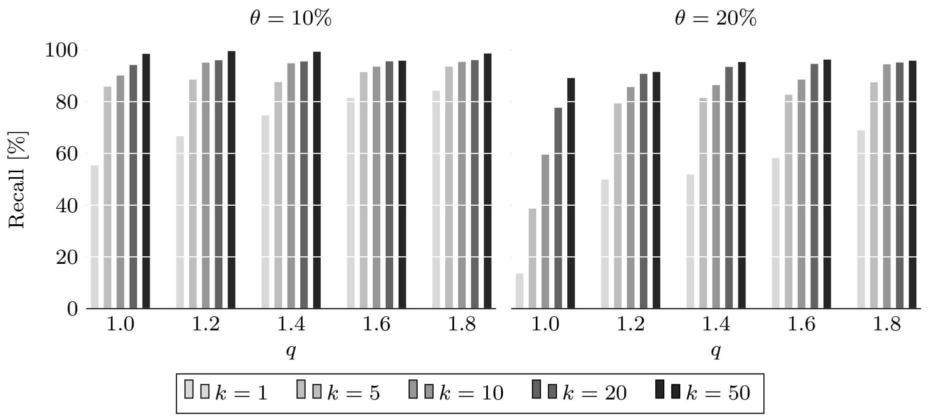

Fig. 4: Recall of our method on artificial graph databases with varying expected edge factor q, for frequency thresholds 10% and 20%.

graph. Sampling 10 spanning trees per graph already results in around 90% recall for the ZINC dataset and in a recall of 80% for the NCI-HIV dataset.

Stability The results above indicate that a relatively high recall of the frequent tree patterns can be achieved even for a very small number of random spanning trees. We now report empirical results showing that the output pattern set of Algorithm 1

	θ	$k = 1$	$k = 2$	$k = 3$	$k = 10$	$k = 20$
NCI-HIV	5%	20.13 ± 1.20	35.53 ± 1.34	46.48 ± 0.51	78.32 ± 0.85	91.11 ± 1.29
	10%	20.26 ± 2.22	34.45 ± 1.42	45.40 ± 1.59	79.94 ± 1.82	92.44 ± 1.34
	20%	24.45 ± 1.38	39.76 ± 1.68	50.41 ± 1.14	83.38 ± 1.40	94.72 ± 1.31
ZINC	5%	36.80 ± 0.87	56.70 ± 1.65	68.42 ± 0.94	92.50 ± 0.45	97.92 ± 0.55
	10%	32.77 ± 1.89	51.36 ± 1.84	64.47 ± 1.40	92.49 ± 1.18	86.70 ± 22.83
	20%	31.03 ± 2.59	48.99 ± 3.05	61.41 ± 3.41	90.53 ± 1.28	97.89 ± 0.40

Table 1: Recall with standard deviation of the probabilistic tree patterns on the NCI-HIV and ZINC datasets for frequency thresholds 5%, 10%, and 20%.

Iteration	1	2	3	4	5	6	7	8	9	10	$\overline{\mathfrak{S}(G)}$
NCI-HIV	3920	20	5	10	14	7	2	6	7	2	169
$ZINC_{40k}$	9898	18	17	11	10	22	7	7	9	1	36

(a) NCI-HIV and ZINC for $\theta = 10\%$ and $k = 1$ sampled tree.

Iteration	1	2	3	4	5	$\overline{\mathfrak{S}(G)}_m$
$q = 1.0$	692	2	5	8	3	7
$q = 1.2$	750	2	0	0	11	55
$q = 1.4$	806	18	0	0	0	2267
$q = 1.6$	824	1	0	0	0	$3.4e^5$
$q = 1.8$	824	2	0	0	0	$8.7e^7$
$q = 2.0$	850	0	0	1	0	$19e^9$
$q = 3.0$	814	26	1	4	0	$99e^{15}$
$q = 5.0$	822	4	0	0	20	$11e^{22}$

(b) Random graphs with different edge factors q for $\theta = 10\%$ and $k = 10$ sampled trees.

Table 2: Repetitions of the probabilistic frequent subtree mining experiment. The numbers reported are the number of probabilistic patterns that were not in the union of all probabilistic patterns found up to the current iteration. $\overline{\mathfrak{S}(G)}$ denotes the median number of spanning trees per graph in the data set for comparison.

is quite stable (i.e., independent runs of our probabilistic frequent tree mining algorithm yield similar sets of frequent patterns). To empirically demonstrate this advantageous property, we run Algorithm 1 several times on the same values of the parameters k and θ and observe how the union of the probabilistic tree patterns grows.

To this end, we fix two sets of graphs, each of size approximately $40,000$, as follows: We take all connected graphs in NCI-HIV, as well as a random subset ZINC$_{40k}$ of ZINC that contains $40,000$ graphs. We run Algorithm 1 10 times for the datasets obtained with parameters $k = 1$ and $\theta = 10\%$. Each execution results in a set $\mathcal{F}_i$ of probabilistic subtree patterns, from which we define $U_i = \bigcup_{j=0}^{i} \mathcal{F}_j$ with $\mathcal{F}_0 := \emptyset$. Table 2 (a) reports $|\mathcal{F}_i \setminus U_{i-1}|$, i.e., the number of *new* probabilistic subtree patterns found in iteration i for $i = 1, \ldots, 10$ on the left. For an initial number of $3,920$ (NCI-HIV) and $9,898$ (ZINC$_{40k}$) probabilistic patterns, the number of newly discovered patterns drops to at most 22 in the upcoming iterations.

We observed this behavior consistently on the artificial graphs (over all observed edge factors, all numbers of sampled spanning trees, and all frequency

Dataset	k	θ	$\|\mathcal{F}\|$	Levelwise	Greedy	BinarySearch
MUTAG	5	10%	452	206.38	116.12	131.07
MUTAG	10	10%	543	244.11	148.02	163.04
MUTAG	15	10%	562	254.86	148.98	167.66
MUTAG	20	10%	573	260.18	151.82	173.91
PTC	5	10%	1,430	321.04	175.32	193.84
PTC	5	1%	9,619	734.79	411.26	472.86
PTC	10	10%	1,566	354.20	191.70	209.26
PTC	20	10%	1,712	376.65	206.36	228.60
DD	5	10%	8,111	3,547.22	3,883.22	3,591.63
DD	10	10%	18,137	6,670.93	7,417.47	6,731.36
DD	20	10%	33,100	11,005.49	12,091.99	11,091.27
NCI1	5	10%	1,819	431.19	284.74	303.03
NCI1	5	1%	21,306	900.68	617.95	675.61
NCI1	20	10%	2,441	557.70	364.23	392.65
NCI109	5	10%	2,182	462.62	306.05	330.39
NCI109	5	1%	19,099	886.06	607.39	670.34
NCI109	20	10%	2,907	598.36	391.59	422.38

Table 3: Average number of subtree isomorphism tests per graph of the algorithms from Section 4.1 on different datasets and corresponding pattern sets $\mathcal{F}$ for varying number k of random spanning trees and frequency thresholds θ.

thresholds). Table 2 (b) shows the results for $\theta = 10\%$, $k = 10$, and 5 iterations on the right. Similarly to the evaluation of the recall above, each artificial dataset consists of 50 graphs. To put our recall and stability experiments into context, note that the median[8] number of spanning trees per graph is depicted in Table 2, as well.

These results together clearly show that the generated feature set does *not* depend too much on the particular spanning trees selected at random. Overall this means that independent runs of our algorithm yield similar feature sets on the same data. This observation, combined with the remarkable recall results of the previous experiment, is essential; high recall and stability together indicate that the predictive performance of any (computationally intractable) *exact* frequent subtree based method can closely be approximated by our (computationally feasible) *probabilistic* frequent subtree based algorithm, even for small values of k. We will further study the predictive performance of probabilistic frequent subtrees in Section 5.3.

5.2 Fast Graph Embedding into PFS Feature Spaces

We now empirically investigate the *speedup* of the methods proposed in Section 4.1 for computing complete and partial embeddings into PFS feature spaces. (We recall that the methods in Section 4.1 are not specific to probabilistic frequent tree patterns.) The main goal of the methods considered was to reduce the number of subgraph isomorphism tests during the computation of the complete feature vector or the min-hash sketch for a query graph. We assess their effectiveness from this

[8] We use the median, as there are some graphs with excessively many spanning trees that distort the average.

aspect by investigating the average number of subtree isomorphism evaluations (i.e., Is $T \preccurlyeq \mathfrak{S}_k(G)$?) per graph on various real-world datasets. To this end, we implemented our algorithms in C using the subtree isomorphism algorithm by Shamir and Tsur (1999) and the sampling algorithm by Wilson (1996).

We start by investigating our methods computing complete embeddings. To obtain probabilistic frequent subtree pattern sets, we have applied our frequent subgraph mining method from Section 3 with different values of k and θ to a randomly sampled subset of 10% of the graphs in each dataset. Using the resulting set of probabilistic trees and the same k, we computed the binary feature vector for each graph in the datasets and calculated the average number of calls to the pattern matching operator $T \preccurlyeq \mathfrak{S}_k(G)$. Table 3 shows the average number of subtree isomorphism tests per graph for the LEVELWISE, GREEDY, and BINARYSEARCH algorithms (cf. Section 4.1). For comparison, we report the cardinality of each pattern set as well (column $|\mathcal{F}|$), which is the number of pattern matching evaluations performed by the brute-force embedding algorithm. It can be seen that GREEDY performs best in general, evaluating the matching operator on average only on 19.78% of all patterns. BINARYSEARCH evaluates 20.49%, while LEVELWISE 27.47% of all patterns per graph on average. The ranking of the methods is consistent over all datasets, except for DD, where the ranking is reversed; here, LEVELWISE evaluates less patterns than BINARYSEARCH which, in turn, evaluates less patterns than GREEDY. Overall, however, we can conclude that GREEDY and BINARYSEARCH, which prune both negative and positive patterns, outperform the methods not pruning at all (brute-force) or pruning only negative patterns (LEVELWISE). This is a significant improvement in light of the speed $O\left(n^{2.5}/\log n\right)$ of the fastest subtree isomorphism algorithm (Shamir and Tsur, 1999).

We now compare our min-hash sketching technique (Algorithm 5) designed for probabilistic frequent subtree patterns with the best *naive* complete embedding algorithm from Table 3. It is important to note that our algorithm may perform more subgraph isomorphism tests than the naive algorithm. This is due to the fact that, in contrast to the naive algorithm, we do not traverse F systematically, but randomly based on the selected permutations. Table 4 shows the average number of subtree isomorphism tests per graph together with the cardinality of the pattern set, for the same datasets and pattern sets as in Table 3. Column "best naive" shows the average number of evaluations performed by the best method from Table 3. The last four columns are the results of our algorithm for sketch size $K =$ 32, 64, 128, and 256 respectively. One can see that Algorithm 5 (columns MH32–MH256) performs dramatically less subtree isomorphism tests than the brute-force algorithm (column $|\mathcal{F}|$) and that it outperforms also the best algorithm for complete embedding computation in all cases, except for $\theta = 1\%$. MH32 evaluates the matching operator on average on 4.74% of all patterns, while MH256 evaluates on average 12.92%. For example, on DD for $k = 10$ and $\theta = 10\%$, the best naive algorithm (LEVELWISE) evaluates subtree isomorphism for 11,005 patterns per graph on average, which is roughly one third of the total pattern set ($|\mathcal{F}|$), while our method evaluates subtree isomorphism 345 times on average for sketch size 32, ranging up to 2,190 times for sketch size 256. Compared to that, the best naive algorithm performs 6.6 (resp. 1.8) times as many subtree isomorphism tests as our method for $K = 32$ (resp. $K = 256$). Again, this is a significant improvement in light of the high runtime complexity of the subtree isomorphism test.

Dataset	k	θ	$\lvert\mathcal{F}\rvert$	best naive	MH32	MH64	MH128	MH256
MUTAG	5	10%	452	116.12	49.93	68.24	96.12	127.42
MUTAG	10	10%	543	148.02	42.77	63.77	90.57	125.39
MUTAG	15	10%	562	148.98	45.39	65.96	94.87	133.91
MUTAG	20	10%	573	151.82	55.34	76.32	105.15	135.11
PTC	5	10%	1,430	175.32	70.07	102.62	121.12	156.12
PTC	5	1%	9,619	411.26	236.31	327.27	475.35	611.92
PTC	10	10%	1,566	191.70	79.63	108.59	109.44	147.91
PTC	20	10%	1,712	206.36	17.60	25.81	31.49	39.62
DD	5	10%	8,111	3,547.22	260.47	486.09	846.09	1,374.76
DD	10	10%	18,137	6,670.93	317.82	568.23	1,072.58	1,936.42
DD	20	10%	33,100	11,005.49	344.59	653.66	1,242.03	2,190.15
NCI1	5	10%	1,819	284.74	89.12	137.75	185.22	221.21
NCI1	5	1%	21,306	617.95	615.62	920.17	1,227.52	1,378.18
NCI1	20	10%	2,441	364.23	115.07	183.54	220.14	255.58
NCI109	5	10%	2,182	306.05	115.62	170.43	206.23	254.70
NCI109	5	1%	19,099	607.39	532.38	727.15	1,057.18	1,348.27
NCI109	20	10%	2,907	391.59	110.42	175.76	226.07	284.92

Table 4: Average number of subtree isomorphism tests per graph needed to compute min-hash sketches for different datasets and corresponding pattern sets $\mathcal{F}$ for varying number of random spanning trees (k) and frequency thresholds θ. We report the average number of subtree isomorphism tests evaluated by the best naive method computing a complete embedding for each graph and by Algorithm 5 for $K = 32$, 64, 128, and 256 (last four columns).

5.3 Predictive and Retrieval Performance

Finally, we look at the suitability of PFS feature spaces for graph classification and retrieval tasks. We show that the predictive performance of probabilistic frequent subtree kernels is comparable to that of frequent subgraph kernels and that min-hashing in PFS feature spaces only slightly decreases the performance of Hamming feature spaces spanned by *complete* sets of frequent subgraphs. In fact, the probabilistic frequent subtree and min-hash kernels yield results that are comparable to the rbf-kernel over frequent subgraphs. To measure the retrieval performance of probabilistic frequent subtrees, we use exact and approximate Jaccard-similarities over PFS feature spaces to retrieve the closest molecules given a positive query molecule. We show that the fraction of the closest molecules that are positive is much higher than the baseline. These results together indicate that PFS feature spaces are well-suited to express semantically relevant concepts in chemical graph datasets.

Graph Classification We start by an empirical analysis of the predictive performance of PFS feature spaces in the context of graph classification. We also consider the Jaccard-similarity. It induces a kernel on sets, which is a special case of the Tanimoto kernel (see, e.g., Ralaivola et al, 2005). Interestingly, its approximation based on min-hashing is a kernel as well. Hence, we can use probabilistic frequent subtrees and min-hash sketches in PFS feature spaces together with these two kernels in support vector machines to learn a classifier. We use 5-fold cross-validation and report the average area under the ROC curve obtained using libSVM (Chang and Lin, 2011) for the datasets MUTAG, PTC, DD, NCI1, and NCI109. We omit the results with NCI-HIV because LibSVM was unable to process the Gram matrix for this dataset. We note, however, that our algorithm required less than 10 (resp. 26) minutes for sketch size $K = 32$ (resp. $K = 256$) for computing the Gram matrix for the full set of NCI-HIV, while this time was 5.5 hours for the exact Jaccard-similarity. The runtime of the preprocessing step to compute a set of probabilistic frequent subtrees on a sample of the database is not counted for both cases, by noting that they were less than three minutes each.

To this end, we fixed the number of random spanning trees per graph to $k = 5$ and sampled 10% of the graphs in a dataset to obtain the probabilistic frequent subtree patterns of up to 10 vertices. In Table 5 we report the results for $\theta = 10\%$ for our min-hash method with sketch sizes K varying between 32 and 256 (first four rows), for exact Jaccard-similarity (row "PFS (Jacc)"), and for the rbf-kernel (row "PFS (rbf)"), all using probabilistic frequent subtrees generated with the parameters above. A lower frequency threshold is practically unreasonable e.g. for MUTAG, as it contains only 188 compounds. We compare the results obtained with frequent subgraph patterns (FSG) (Deshpande et al, 2005) using the full set of frequent connected subgraphs of up to 10 vertices with respect to the *full* datasets (i.e., not only for a sample of 10%) using the Jaccard (row "FSG (Jacc)") and rbf (row "FSG (rbf)") kernels. We also report results obtained by the Hash-kernel (row "HK") (Shi et al, 2009), which uses count-min sketching on random induced subgraphs up to size 9.

One can see that the results of MH256 are close to those obtained by exact Jaccard-similarities over probabilistic frequent subtrees (PSF (Jacc)), which, in turn, are close to those obtained by exact Jaccard-similarities over all frequent subgraphs (FSG (Jacc)). Thus, the min-hash kernel in PFS feature spaces performs only slightly worse than in ordinary frequent subgraph feature spaces (cf. MH256 vs. FSG (Jacc)). One can also observe that the min-hash kernel outperforms the rbf-kernel in PFS feature spaces in all datasets, except for DD (cf. MH256 vs. PSF (rbf)). It also outperforms the rbf-kernel in frequent subgraph feature spaces on all datasets, except for NCI1 (cf. MH256 vs. FSG (rbf)). While the Hash-kernel is the best by a comfortable margin on MUTAG, the contrary is true for DD (cf. MH256 vs. HK). Most notably, it could not provide any result for NCI1 and NCI109 in practically reasonable time.

We also conducted these experiments for $k = 20$ random spanning trees. For identical frequency threshold, the AUC improved by 3% on MUTAG, while only slightly changing for the other datasets. Similar results to those in Table 5 were obtained when reducing the frequency threshold of the methods to 1%: The AUC improved roughly by 1%, processing time and memory consumption, however, drastically increased.

Overall, we can conclude that (1) the predictive performance of PFS feature spaces is comparable to that of frequent subgraph features spaces for molecular graph mining, (2) Jaccard-similarities (more precisely, the Jaccard-kernel) is a powerful similarity measure for chemical graphs, and (3) the min-hash kernel in PFS feature spaces is a valid competitor to the rbf-kernel in frequent subgraph feature spaces.

Positive Instance Retrieval Finally we use a simple setup to evaluate the retrieval performance of min-hashing in PSF feature spaces by comparing it to exact Jaccard-similarity in PFS feature spaces, as well as to the path min-hash kernel (Teixeira

θ	Method	MUTAG	PTC	DD	NCI1	NCI109
10%	$MH32$	87.84	58.97	77.58	77.36	77.48
10%	$MH64$	87.73	58.68	79.91	78.04	79.54
10%	$MH128$	87.59	56.97	82.07	79.94	79.94
10%	$MH256$	87.78	57.18	83.58	80.76	81.72
10%	PFS (Jacc)	89.04	57.72	85.38	82.28	82.41
10%	FSG (Jacc)	89.84	60.60	84.54	82.97	82.31
10%	PFS (rbf)	84.22	54.17	84.67	79.09	78.05
10%	FSG (rbf)	87.34	56.76	82.20	81.66	81.55
	HK	93.00	62.70	81.00	n/a	n/a

Table 5: AUC values for our method (MH) for sketch sizes $K = 32, 64, 128, 256$, $k = 5$ spanning trees per graph, and frequency threshold $\theta = 10\%$ to obtain the feature set. "n/a" indicates that Shi et al (2009) did not provide results for the respective datasets.

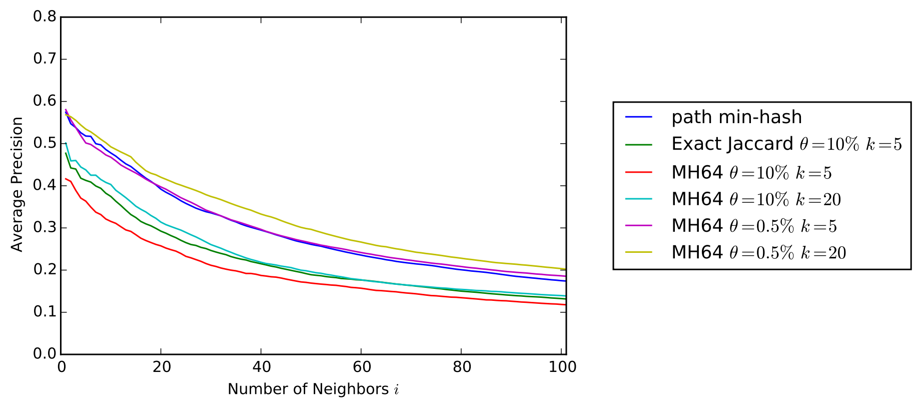

Fig. 5: Average fraction of "active" molecules among the i nearest neighbors of positive molecules in NCI-HIV dataset for path min-hash (Teixeira et al, 2012), exact Jaccard-similarity for frequent probabilistic tree patterns, and for our method with $K = 64$.

et al, 2012). For the evaluation we use the highly skewed NCI-HIV dataset. For each molecule of class A (i.e., "active") of NCI-HIV, we retrieve its i nearest neighbors (excluding the molecule itself) from the dataset and take the fraction of the neighbors of class A. This measure is known in the Information Retrieval community as *precision at i*. As a baseline, a random subset of molecules from NCI-HIV is expected to contain less than 1% of active molecules due to the highly skewed class distribution. All methods show a drastically higher precision for the closest up to 100 neighbors on average than this baseline.

Figure 5 shows the average precision at i (taken over all 329 active molecules) for i ranging from 1 to 100. The number k of sampled spanning trees per graph, as well as the frequency threshold θ has a strong influence on the quality of our method. To obtain our results, we have sampled 5 (resp. 20) spanning trees for each graph and used a random sample of $4,000$ graphs to obtain pattern sets for thresholds $\theta = 10\%$ and $\theta = 0.5\%$ respectively. We plot the min-hash-based precision for the four feature sets obtained in this way by our algorithm as a function of i for sketch size $K = 64$. We have compared this to the precision obtained by the exact Jaccard-similarity for $\theta = 10\%$ and $k = 5$, as well as to the precision obtained by path min-hash (Teixeira et al, 2012), both for the same sketch size $K = 64$.

The average precision obtained for the exact Jaccard-similarities is slightly better than that of path min-hash. While our method performs comparably to path min-hash for $\theta = 0.5\%$ and $k = 5$, for $\theta = 0.5\%$ and $k = 20$ spanning trees it outperforms all other methods.

We were not able to compute the precisions for $\theta = 1\%$ or for $k = 20$ sampled spanning trees for the exact Jaccard-similarity. The Python implementation we used to calculate the similarity computations for exact Jaccard-similarity was not able to deal with the high dimensionality of the feature space, independently of the sparsity of the feature vectors. This indicates that the space required to compute the Jaccard-similarity is crucial for high-dimensional feature spaces.

6 Concluding Remarks

We proposed simple probabilistic techniques to resolve the computational intractability of problems related to frequent subgraph feature spaces. In particular, we gave up the requirement of completeness to arrive at an efficient probabilistic frequent subtree mining algorithm for *arbitrary* graph databases and a corresponding efficient probabilistic embedding method for unseen graphs that is not correct. Our empirical results on various real-world benchmark graph datasets show that the resulting PFS feature space with the incorrect embedding operator is stable and expressive enough in terms of predictive and retrieval performance compared to the frequent subgraph feature spaces with the correct embedding operator. In contrast to ordinary frequent subgraphs, probabilistic frequent subtrees can be computed much faster and the computation has a much smaller memory footprint in all stages.

Our probabilistic mining and embedding techniques can naturally be extended to disconnected transaction graphs as well. Indeed, for a disconnected transaction graph G we can consider its *spanning forests*, each consisting of a random spanning tree for each connected component of G. The forest $\mathfrak{S}_k(G)$ generated in Algorithm 1 then consists of k such spanning forests. Since any subgraph isomorphism preserves connectivity, it remains true that any tree pattern that is subgraph isomorphic to $\mathfrak{S}_k(G)$ is also subgraph isomorphic to G.

Though the probabilistic pattern matching operator can be evaluated in polynomial time, each of its invocations during embedding into PFS feature spaces induces a non-negligible amount of work. To accelerate the embedding, we introduced different strategies to practically reduce the number of such calls by utilizing the anti-monotonicity of subgraph isomorphism on the tree pattern poset. In particular, if one is interested in the Jaccard-similarity between two graphs then min-hash sketches can be computed very efficiently in this way. We empirically demonstrated the effectiveness of our algorithms, resulting in a theoretically efficient and practically effective system to embed arbitrary graph databases into PSF feature spaces.

Our algorithms to compute (partial) embeddings for a given tree pattern set and for subtree isomorphism can easily be adapted to any finite pattern set and pattern matching operator (e.g., exact subgraph isomorphism or graph homomorphism) if the pattern matching operator induces a partial order on the pattern set in which it is (anti-)monotone. While the number of evaluations of the pattern matching operator can drastically be reduced in this way, the complexity of the algorithm depends on that of the pattern matching operator. The one-sided error of our probabilistic subtree isomorphism test seems to have no significant effect on the experimental results. This raises the question whether we can further relax the correctness of subtree isomorphism resulting in an algorithm that runs in at most sub-quadratic time, without any significant negative effect on the predictive/retrieval performance.

Acknowledgments

The authors thank the anonymous reviewers for their careful reviews and useful suggestions.

References

Broder AZ (1997) On the resemblance and containment of documents. In: Compression and Complexity of SEQUENCES Proceedings, IEEE, IEEE Comput. Soc, pp 21–29, DOI 10.1109/sequen.1997.666900

Broder AZ, Charikar M, Frieze AM, Mitzenmacher M (2000) Min-wise independent permutations. Journal of Computer and System Sciences 60(3):630–659, DOI 10.1006/jcss.1999.1690

Chang CC, Lin CJ (2011) Libsvm: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2(3):1–27, DOI 10.1145/1961189.1961199

Chi Y, Muntz RR, Nijssen S, Kok JN (2005) Frequent subtree mining - an overview. Fundamenta Informaticae 66(1–2):161–198

Deshpande M, Kuramochi M, Wale N, Karypis G (2005) Frequent substructure-based approaches for classifying chemical compounds. Transactions on Knowledge and Data Engineering 17(8):1036–1050, DOI 10.1109/tkde.2005.127

Diestel R (2012) Graph Theory, 4th Edition, Graduate texts in mathematics, vol 173. Springer

Erdős P, Rényi A (1959) On random graphs. Publicationes Mathematicae 6:290–297

Garey MR, Johnson DS (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman

Gärtner T, Flach P, Wrobel S (2003) On graph kernels: Hardness results and efficient alternatives. In: Schölkopf B, Warmuth MK (eds) Annual Conference on Computational Learning Theory and Kernel Workshop, (COLT/Kernel) Proceedings, Springer, Lecture Notes in Computer Science, vol 2777, pp 129–143, DOI 10.1007/978-3-540-45167-9_11

Geppert H, Horváth T, Gärtner T, Wrobel S, Bajorath J (2008) Support-vector-machine-based ranking significantly improves the effectiveness of similarity searching using 2d fingerprints and multiple reference compounds. Journal of Chemical Information and Modeling 48(4):742–746, DOI 10.1021/ci700461s

Horváth T, Ramon J (2010) Efficient frequent connected subgraph mining in graphs of bounded tree-width. Theoretical Computer Science 411(31–33):2784–2797, DOI 10.1016/j.tcs.2010.03.030

Horváth T, Bringmann B, Raedt LD (2007) Frequent hypergraph mining. In: Muggleton S, Otero RP, Tamaddoni-Nezhad A (eds) Inductive Logic Programming (ILP) Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 4455, pp 244–259, DOI 10.1007/978-3-540-73847-3_26

Mannila H, Toivonen H (1997) Levelwise search and borders of theories in knowledge discovery. Data Mining and Knowledge Discovery 1(3):241–258, DOI 10.1023/a:1009796218281

Nijssen S, Kok JN (2005) The gaston tool for frequent subgraph mining. Electronic Notes in Theoretical Computer Science 127(1):77–87, DOI 10.1016/j.entcs.2004.12.039

Ralaivola L, Swamidass SJ, Saigo H, Baldi P (2005) Graph kernels for chemical informatics. Neural Networks 18(8):1093–1110, DOI 10.1016/j.neunet.2005.07.009

Shamir R, Tsur D (1999) Faster subtree isomorphism. Journal of Algorithms 33(2):267–280, DOI 10.1006/jagm.1999.1044

Shi Q, Petterson J, Dror G, Langford J, Smola AJ, Vishwanathan SVN (2009) Hash kernels for structured data. Journal of Machine Learning Research 10:2615–2637, DOI 10.1145/1577069.1755873

Teixeira CHC, Silva A, Jr WM (2012) Min-hash fingerprints for graph kernels: A trade-off among accuracy, efficiency, and compression. Journal of Information and Data Management 3(3):227–242, URL `http://seer.lcc.ufmg.br/index.php/jidm/article/view/199`

Welke P, Horváth T, Wrobel S (2016a) Min-hashing for probabilistic frequent subtree feature spaces. In: Calders T, Ceci M, Malerba D (eds) Discovery Science (DS) Proceedings, Lecture Notes in Computer Science, vol 9956, pp 67–82, DOI 10.1007/978-3-319-46307-0_5

Welke P, Horváth T, Wrobel S (2016b) Probabilistic frequent subtree kernels. In: Ceci M, Loglisci C, Manco G, Masciari E, Ras ZW (eds) New Frontiers in Mining Complex Patterns (NFMCP) Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 9607, pp 179–193, DOI 10.1007/978-3-319-39315-5_12

Willett P (2006) Similarity-based virtual screening using 2d fingerprints. Drug discovery today 11(23):1046–1053

Wilson DB (1996) Generating random spanning trees more quickly than the cover time. In: Miller GL (ed) ACM Symposium on the Theory of Computing (STOC) Proceedings, ACM, pp 296–303, DOI 10.1145/237814.237880

Zhao P, Yu JX (2008) Fast frequent free tree mining in graph databases. World Wide Web 11(1):71–92, DOI 10.1007/s11280-007-0031-z