Efficient Frequent Subtree Mining Beyond Forests

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt
von

Pascal Welke

aus
Bonn

Bonn, 2018

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

Diese Arbeit wurde von Dr. Tamás Horváth und Prof. Dr. Stefan Wrobel betreut

1. Gutachter: Prof. Dr. Stefan Wrobel
2. Gutachter: Prof. Dr. Christian Bauckhage

Tag der Promotion: 20.3.2019
Erscheinungsjahr: 2019

Pascal Welke

Department of Computer Science III
welke@uni-bonn.de

Declaration

I, Pascal Welke, confirm that this work is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at the point of their use. A full list of the references employed has been included.

Acknowledgments

This thesis would not have been possible without the support of many people. In particular, I would like to thank my supervisors Tamás Horváth and Stefan Wrobel. Without them, this thesis would not exist. I don't know many people that have the opportunity to talk to their supervisor as frequently as I did with Tamás. I know even less[1], who additionally got invited to several intense weeks of discussions, writing, no Internet, and great food and wine in Nemesvita. I'd like to thank Stefan for the freedom, the generous support of my work, and for creating our motivating working environment.

During the course of my work on this thesis I was surrounded by a great group of people. CAML became KDML, countless coffees were imbibed, discussions were had and slowly an idea for this thesis manifested. Thank you, Krisztian Buza, Thomas Gärtner, Laurentiu Ilici, Michael Kamp, Olana Missura, Daniel Paurat, Till Schulz, Florian Seiffarth, Katrin Ullrich, and all the great people at Fraunhofer IAIS. A special thank you also goes to Ionut Andone, Konrad Błaszkiewicz, and Alexander Markowetz for the fruitful distractions.

I'd like to thank Lars Borutzky, Moritz Fürneisen, Tamás Horváth, Michael Kamp, Rajkumar Ramamurthy, Florian Seiffarth, Till Schulz, and Stefan Welke, who have read various draft versions of my thesis and gave valuable feedback, pointed out errors, and asked ~~nasty~~ helpful questions. All remaining mistakes are of course my own responsibility.

Last but not least, I want to thank my parents Daniela and Stefan Welke and my girlfriend Hanna Hünert for everything. You are the best!

[1] All of them were supervised by Tamás

Abstract

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. If the instances are graphs, then frequent connected subgraphs are a well-suited pattern language to define such feature spaces. Identifying the set of frequent connected subgraphs and subsequently computing embeddings for graph instances, however, is computationally intractable. As a result, existing frequent subgraph mining algorithms either restrict the structural complexity of the instance graphs or require exponential delay between the output of subsequent patterns. Hence distance-based learners lack an efficient way to operate on arbitrary graph data. To resolve this problem, in this thesis we present a mining system that gives up the demand on the completeness of the pattern set to instead guarantee a polynomial delay between subsequent patterns. Complementing this, we devise efficient methods to compute the embedding of arbitrary graphs into the Hamming space spanned by our pattern set. As a result, we present a system that allows to efficiently apply distance-based learning methods to arbitrary graph databases.

To overcome the computational intractability of the mining step, we consider only frequent *subtrees* for arbitrary graph databases. This restriction alone, however, does not suffice to make the problem tractable. We reduce the mining problem from arbitrary graphs to forests by replacing each graph by a polynomially sized forest obtained from a random sample of its spanning trees. This results in an incomplete mining algorithm. However, we prove that the probability of missing a frequent subtree pattern is low. We show empirically that this is true in practice even for very small sized forests. As a result, our algorithm is able to mine frequent subtrees in a range of graph databases where state-of-the-art exact frequent subgraph mining systems fail to produce patterns in reasonable time or even at all. Furthermore, the predictive performance of our patterns is comparable to that of exact frequent connected subgraphs, where available.

The above method considers polynomially many spanning trees for the forest, while many graphs have exponentially many spanning trees. The number of patterns found by our mining algorithm can be negatively influenced by this exponential gap. We hence propose a method that can (implicitly) consider forests of exponential size, while remaining computationally tractable. This results in a higher recall for our incomplete mining algorithm. Furthermore, the methods extend the known positive results on the tractability of *exact* frequent subtree mining to a novel class of transaction graphs. We conjecture that the next natural extension of our results to a larger transaction graph class is at least as difficult as proving whether **P** = **NP**, or not.

Regarding the graph embedding step, we apply a similar strategy as in the mining step. We represent a novel graph by a forest of its spanning trees and decide whether the frequent trees from the mining step are subgraph isomorphic to this forest. As a result, the embedding computation has one-sided error with respect to the exact subgraph isomorphism test but is computationally tractable. Furthermore, we show that we can leverage a partial order on the pattern set. This structure can be used to reduce the runtime of the embedding computation dramatically. For the special case of Jaccard-similarity between graph embeddings, a further substantial reduction of runtime can be achieved using min-hashing. The Jaccard-distance can be approximated using small sketch vectors that can be computed fast, again using the partial order on the tree patterns.

Die Welt geht nicht unter,
wenn wir nicht vollständig
aufzählen.

(Tamás Horváth)

Contents

1. Introduction

Since the advent of the first digital computers in the second half of the twentieth century there has been a gap between the data analysis tasks one would like to solve and the capabilities of existing systems. The amount of data available to computers always exceeded the processing power of even the most advanced computers. Progress in the development of hardware and software did little to solve this issue.[1] That is, despite the ongoing digitalization of our society today and the resulting simplification of the data collection process, it is more and more difficult to analyze the data automatically to derive knowledge from it. In particular, this data comes in various forms, such as discrete or real valued vectors (sales transactions, sensor measurements), text, audio, video (communication content), or relations among entities (communication metadata, access patterns, similarities between objects). Due to this variability and volume of data it gets increasingly difficult to derive valuable insights from it.

In the 1990s the field of data mining and knowledge discovery developed. One of its goals is to devise algorithms to identify interesting patterns in data and to *learn* from data, i.e., to automatically synthesize programs (called models) that capture some relations among data.[2] The models can then be applied to novel data, for example to predict whether some access pattern is fraudulent, or not. One quite general class of such methods works on data viewed as a collection of examples, or *transactions* (like sales transactions in a store) from a (usually unknown and possibly infinite) set equipped with some measure of similarity and tries to infer an unknown target variable. These *distance-based* learning methods assume that the target variables of similar examples behave similarly as well (i.e., that we can learn from the behavior of close-by instances).

If the data can be represented as discrete or real valued vectors of fixed length, various similarity measures (such as, for example, the Euclidean distance) are available. If, however, the data is text, video, or of multi-relational nature, defining a suitable similarity measure for a given learning task is a difficult problem on its own. A common approach is to embed such structured data into a space spanned by some feature set that is equipped with an easy-to-compute similarity function. It is of course crucial and nontrivial to choose a suitable feature set and corresponding similarity function for a given learning task. Indeed, the quality of this method applied to some particular problem strongly depends on the semantic relevance of the features considered, implying that one might be interested in feature languages of high expressive power.

[1] This is of course partly due to human nature: If technical advances have solved certain issues, people immediately strive to address even larger problems.

[2] Some authors consider both steps to be equivalent (e.g. Shawe-Taylor and Cristianini, 2004) and do not distinguish between patterns and models.

In this thesis, we focus on the case that the data at hand consists of relational structures. More precisely, we consider graphs, i.e., each object in the data represents a set of entities and there are some binary relations among the entities. Graphs are a powerful representation language; communication among people or distributed systems, social networks, or protein-protein interactions are some examples of using graphs as data model. In particular, we will pay special attention to chemical molecules, which can also be represented as graphs. Here the entities are atoms and the binary relations correspond to bonds between atoms. Many other applications are possible, e.g., we can view the web browsing of a user as a walk in a graph where entities are web pages and relations correspond to links, etc. In fact, even text, pictures, or videos can be represented as graphs.

While graphs easily model many real world applications, working with graphs poses a unique set of challenges. With the great expressive power of graphs comes great computational complexity: Many questions that are trivial to answer for data which is represented in a different form are not known to be answerable efficiently for graphs. It is not even known whether graph isomorphism, i.e., if two graphs are equal except for reordering their vertices, can be decided in polynomial time. As a result, defining a suitable similarity measure for a given set of graphs is a nontrivial task.[3] Furthermore, regarding the embedding idea described above, there is no straightforward fixed set of features for arbitrary graphs that is semantically meaningful. A common way to define such features is to choose a set of (semantically relevant) graph patterns to span the feature space and to check whether these features "appear" in the transaction graphs, or not. After this step we are in the world of fixed length (binary) vectors and can apply any similarity function available, such as the Euclidean, Hamming, or Jaccard distances.

The most natural and practically relevant definition of one graph appearing in another is that of subgraph isomorphism. This problem, however, is **NP**-complete. As a result, we cannot expect to be able to compute the embedding of a given arbitrary transaction graph in feasible time. Regarding the choice of the pattern set, a common approach is to find the set of graph patterns that appear frequently in a given (training) database of graphs. Since the first application of these so called *frequent subgraphs* as features to molecule classification (Deshpande et al, 2005), many further studies have empirically demonstrated a remarkable predictive performance of frequent patterns on real-world datasets. In fact, as shown for instance by Bringmann et al (2006) in the context of correlated pattern mining, even very *simple* patterns, such as paths or trees often suffice to obtain considerable predictive accuracy. However, despite the structural simplicity of trees, even frequent subtrees *cannot* be generated in output polynomial time for *arbitrary* transaction graphs (unless $\mathbf{P} = \mathbf{NP}$) (Horváth and Ramon, 2010). Furthermore, the subgraph isomorphism problem from a tree into a graph and hence the embedding computation remains **NP**-complete.

This complexity limitation prohibits frequent pattern mining in practically feasible time even for relatively simple transaction graph databases. In fact, all previous works regarding frequent subgraph mining that we are aware of focus on tree databases (Chi et al, 2003, 2004a) or on the domain of chemical graphs (Borgelt and Berthold, 2002; Borgelt

[3] A metric that respects graph isomorphism is at least as difficult to compute as graph isomorphism itself.

et al, 2005; Nijssen and Kok, 2005; Rückert and Kramer, 2004; Yan and Han, 2002; Zhao and Yu, 2008), where the transactions are quite restricted in their structural complexity. This is not surprising: For tree transactions, frequent subgraphs (trees in this case) can be generated efficiently (Chi et al, 2005) and the embedding computation can be done in polynomial time (Matula, 1968). Chemical graphs, on the other hand, have several structural properties that practically result in good performance of the state-of-the-art mining algorithms. Once we leave the domain of trees or chemical graphs, however, our experiments indicate that to the best of our knowledge there exists no system that can generate frequent subgraphs in practically feasible time and space. We experimented, e.g., with neighborhood graphs extracted from social networks or with certain artificially generated graphs. That is, for no apparent[4] reason the memory requirements of the mining algorithms blow up exponentially, or the algorithms spend more than a day without outputting any patterns on a dataset of 50 small random graphs to identify the frequent patterns (cf. Section 4.2) while databases of over 30 000 similarly sized chemical graphs can be processed in a matter of minutes (Nijssen and Kok, 2005).

However, robust mining and embedding algorithms whose runtimes do *not* depend on certain, typically unknown implicit characteristics of the data, but on some user specified parameters, are of high value. That is, in many applications the transaction graphs have no (known) specific structural properties that could be utilized by the mining algorithm. In contrast, all frequent subgraph mining tools we are aware of are explicitly or implicitly engineered towards certain structural properties of the input graph databases. They are therefore either not applicable for general graph databases, or cannot guarantee worst-case runtimes.

In light of these difficulties and requirements, this thesis is dedicated to the identification of a set of pattern graphs that is of high practical relevance and can at the same time be *efficiently* computed for *arbitrary* transaction graph databases. We also discuss the efficient embedding computation for a (novel) transaction graph class given such a set of patterns. With these two steps, we present a system to define a suitable feature space given a sample of some graph distribution and then to compute the embeddings for unseen graphs. This allows to train and apply distance-based models on graph transactions. To overcome the computational complexity of the above two problems, we propose to focus on *tree patterns* and generate only a *random* subset of frequent subtrees, called *probabilistic* frequent subtrees that can be generated with polynomial delay. To compute the embeddings of transaction graphs in the feature space spanned by these probabilistic frequent subtrees, we employ a similar technique for the subgraph isomorphism problem. This results in a computationally efficient algorithm that computes a sound but incomplete random embedding vector for arbitrary graphs. As a side-effect of our technique, we extend the positive results on efficient *exact* frequent subtree mining to a novel graph class. Before discussing the contributions of this thesis in more detail in Section 1.2, let us quickly demonstrate the suitability of our proposed approach on chemical data below.

[4] We are aware of the reason, see Section 3.1.1.

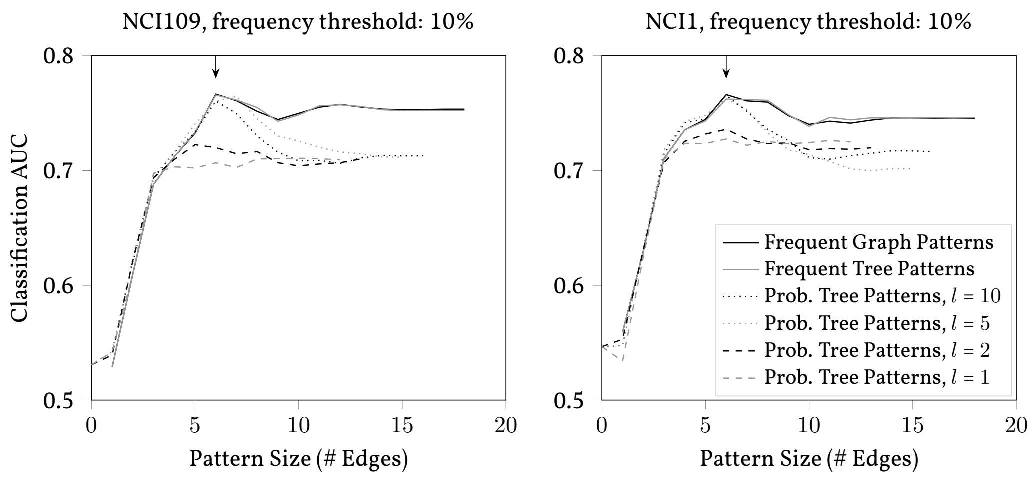


Figure 1.1.: Predictive performance of a SVM trained and evaluated on different feature sets. The two plots show the AUC values (y-axis) that are achieved for features corresponding to frequent (probabilistic) subgraph patterns up to a certain number of vertices (x-axis). Each line corresponds to a certain type of patterns. The arrows from the top of the plot indicate the best performance of the frequent subgraph pattern based classificators.

1.1. A Motivating Experiment

As a motivation for our probabilistic frequent subtree techniques we investigate a predictive chemistry task for molecular databases. On the one hand, this may seem contradictory to our claim above that traditional frequent subgraph mining systems work well on such graphs and that we develop a method to go beyond this application scenario. On the other hand, we wish to compare our method against the "ground-truth" of exact frequent subgraph and frequent subtree based learning methods. Hence we naturally have to restrict our comparison to domains where these two feature sets can be computed within feasible time and memory constraints.

Recall that we want to represent data that consists of multiple graphs in a feature space that is spanned by some set of graph patterns $\mathcal{P}$. More exactly, we represent each graph G by a subset of the patterns in $\mathcal{P}$ that match G (i.e., that are subgraph isomorphic to G); this representation can be stored as a binary vector of fixed length. To do this, we fix a set of graphs (called *database*) and compute the sets of

- all frequent subgraphs,
- all frequent subtrees, and
- our probabilistic frequent subtrees

in the database for a fixed frequency threshold of 10%. We have observed similar qualitative behavior for other frequency thresholds and do not want to convolute this motivation by showing multiple similar plots. We then consider the feature representations of the graphs in the database with respect to the three pattern sets above.

Figure 1.1 shows the results on NCI1 and NCI109. Both datasets consist of chemical molecules and the task is to predict whether they are active against certain types of cancer cells. See Section 2.4 for a more detailed description of these datasets. We report the predictive performances of support vector machines (SVM) (Cortes and Vapnik, 1995) which we measure by the area under the ROC curve (AUC) averaged over a three-fold cross validation.[5] Furthermore, we defer the discussion of the various pattern sets to the technical part of this thesis. For now, it suffices to know that our method has a parameter l that influences its error and runtime in both the pattern mining and graph embedding steps. Figure 1.1 shows plots for our method with parameter $l \in \{1, 2, 5, 10\}$. It contains a plot for each dataset and a line in each such plot for each generated pattern set. A point on such a line corresponds to the AUC value (y-axis) of an SVM classifier trained on feature representations based on patterns up to and including a certain number of edges (x-axis).

The predictive performance of frequent tree patterns (green) and frequent subgraph patterns (blue) is almost identical over both datasets and all pattern sizes. This motivates and justifies our initial simplification of the frequent subgraph mining problem (FCSM) to the frequent subtree mining problem (FTM). Furthermore, the predictive performance of the patterns first increases with the pattern size, reaching its optimum for maximum pattern sizes between 5 and 9 edges, and then decreases. This behavior is consistent through both datasets and we observed it consistently on a number of other datasets and across various frequency thresholds. The predictive performance of our probabilistic subtree patterns shows a similar behavior as a function of the pattern size. Increasing the sampling parameter l generally increases the predictive performance. In fact, for $l = 10$ the best AUC score of the frequent subgraph pattern based classifier can be matched by our method.

We draw three main conclusions from this experiment that motivate us[6] to pursue probabilistic frequent subtree patterns:

1. Our probabilistic patterns are well-suited as a basis for predictive tasks in the context of chemical graph databases.

2. Frequent pattern based classifiers do not seem to benefit from patterns that are too large. On chemical graphs, pattern sizes up to 9 edges per pattern give good predictive performance and larger patterns tend to decrease speed as well as accuracy. This seems to be a general (although not systematically investigated) trend in graph mining, extending beyond frequent pattern mining and classification, e.g. also to cyclic patterns (Horváth et al, 2004) or Weisfeiler-Lehman features (Shervashidze et al, 2011) in a regression setting (Ullrich et al, 2016).

[5] We used libSVM (Chang and Lin, 2011) with a linear kernel function. We did not optimize the soft margin parameter for each feature representation, but kept it in its default setting to speed up the computation and to avoid overfitting by chance.

[6] and hopefully the reader, as well

3. Our method works well in practical scenarios even for relatively small values of the parameter l. This indicates that it can be applied in practically reasonable time and space to distance-based learning problems that were not feasible before.

1.2. Contributions

As already mentioned, to use distance-based learning methods on graphs, a common paradigm also followed in this thesis is to embed the instance space (of graphs) into some appropriately chosen feature space equipped with a metric. In particular, we focus on embedding (labeled) graphs into the d-dimensional *Hamming space* (i.e., $\{0,1\}^d$) spanned by the elements of a pattern set of cardinality d for some $d > 0$. This thesis presents methods to use *frequent subgraphs* as features, *without* any structural restriction on the transaction graph class defining the instance space.[7] 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. Nevertheless the pattern mining problem (i) has gained lots of attention (see Chapter 3), 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 mining step (i) to unseen graphs. We describe our technical contributions to address both computational problems below.

The overall contribution of this thesis is a robust system to map arbitrary graphs to a (learned) Hamming space of fixed dimension, therefore allowing to easily apply distance-based learning methods on graph datasets. By *robust* we mean that the runtime of our system depends *not* on certain, typically unknown implicit characteristics of the data, but that it is polynomial in some user specified parameters, the size of the data, and the size of the output. Such a system is of high value for practical problems: Often the transaction graphs have no (known) specific structural properties that could be utilized by the mining or embedding algorithm. Our algorithm is robust because its delay is bounded by a polynomial which depends only on the number and size of the input graphs and on a sampling

[7] In fact, we restrict our description to connected transaction graphs. This, however, is only done for ease of explanation. All our techniques can easily be extended to disconnected transaction graphs.

parameter. The sampling parameter can be used to control the trade-off between recall and time complexity. In contrast, all frequent subgraph mining tools from other groups are explicitly or implicitly engineered towards certain structural properties of the transaction graphs and have exponential delay in the worst case. Furthermore, they usually neglect the graph embedding step (ii). As a result, these systems are not applicable in such a general scenario. Their runtime or memory requirements might explode for certain datasets and novel graphs cannot be embedded in the feature space spanned by the frequent patterns in feasible time.

1.2.1. Efficient Frequent Subtree Mining

To arrive at a theoretically efficient and practically fast algorithm for the pattern mining problem (i), we restrict the pattern language to *trees*. This restriction alone, however, does not resolve the complexity problems above. Mining frequent subtrees from arbitrary graphs is *not* possible in output polynomial time (unless **P** =**NP**, Horváth et al, 2007). To overcome this limitation, we give up the demand on the completeness of the mining algorithm. Instead, we propose to efficiently compute a subset of all frequent subtrees, which we call *probabilistic frequent subtrees*.

As a first theoretical contribution, we formalize a relaxed frequent subtree mining problem and give sufficient conditions for the existence of an efficient mining algorithm. To this end, we extend the generic algorithm in (Horváth and Ramon, 2010) to the case that (i) pattern and transaction graphs have different characteristics and (ii) only a well defined subset of all frequent patterns shall be enumerated. In particular, we consider the case that the transactions are *arbitrary* graphs and the patterns are trees. Subsequently we propose two novel mining algorithms that are based on sampling spanning trees of the transaction graphs.

Probabilistic Frequent Subtrees

The basic insight leveraged by our techniques is the following: A tree is subgraph isomorphic to a graph if and only if it is subtree isomorphic to one of the graph's spanning trees. Our first algorithm therefore generates probabilistic frequent subtrees in the following simple way: It replaces each transaction graph in the input database by a forest formed by the vertex disjoint union of a *random* subset of its spanning trees. Using, e.g., the levelwise search algorithm (Mannila and Toivonen, 1997), our algorithm generates the set of frequent connected subgraphs (i.e., subtrees) for the forest database obtained. Spanning trees can uniformly be sampled in polynomial time (Wilson, 1996) and subgraph isomorphism from a tree into a forest can be decided in polynomial time (Matula, 1968). Hence the extended generic algorithm of (Horváth and Ramon, 2010) can enumerate probabilistic frequent subtrees with polynomial delay in this way if for each transaction graph, the number of spanning trees in the sample is bounded by a polynomial of the graph's size.

The output of our method is sound (that is, all patterns printed are frequent subtrees), but not necessarily complete (i.e., some frequent subtrees may not be enumerated). Regarding the recall of our method (that is, the fraction of frequent subtrees retrieved by our algorithm), we show that the set of frequent subtrees is well approximated if the number of sampled spanning trees is chosen appropriately. We prove that the probability for a frequent pattern H to be among the probabilistic frequent subtrees is high if the pattern is *important*. That is, if H is frequent in the spanning trees of sufficiently many transaction graphs.

Our extensive empirical evaluation demonstrates that the above idea results in a practically feasible mining algorithm. We show that probabilistic frequent subtrees can be enumerated for social and random graph databases where state-of-the-art exact frequent subgraph mining systems are not able to produce results in practically feasible time. Furthermore, we demonstrate that the *recall* of our probabilistic mining technique is high even for small numbers of sampled spanning trees and that the retrieved set of patterns is very stable. (Notice that *precision* is always 100% for the soundness of the algorithm.) Furthermore, we show that the predictive performance of probabilistic frequent subtrees is comparable to that of the full set of frequent subgraphs and frequent subtrees on chemical datasets.

Boosted Probabilistic Frequent Subtrees

As a next step, we go beyond the limitation of processing polynomially many spanning trees per graph only. We present an algorithm able to generate probabilistic frequent subtrees from arbitrary graphs with polynomial delay by considering a potentially *exponentially* large implicit subset of the spanning trees for each graph in the database. Our boosted mining algorithm is based on a novel pattern matching algorithm. For a tree pattern H and a transaction graph G, it (i) partitions G into a certain set of induced subgraphs, (ii) considers a (random) subset of *local* spanning trees for each induced graph, and (iii) decides whether H is subtree isomorphic to one of the *global* spanning trees of G obtained by combining its local spanning trees in an appropriate way.

Our pattern matching algorithm traverses a rooted tree generated for G in a bottom-up manner and computes the final solution from partial solutions calculated before. The nodes of the rooted tree controlling the evaluation are constructed from the articulation vertices of G. Each node v of such a tree is associated with a set of spanning trees of a certain induced subgraph containing v. Our technique requires an efficient combination of these local spanning trees, as well as a careful assembly of certain partial subtree isomorphisms which can be computed efficiently. We prove that our algorithm decides subgraph isomorphism from a tree pattern H to $\mathfrak{S}$ correctly, where $\mathfrak{S}$ is the set of spanning trees of G that can be obtained from combinations of the local spanning trees. Our algorithm runs in time *polynomial* in the combined size of H, G, and the number of local spanning trees that it considers. The significance of this result is that the number of global spanning trees in $\mathfrak{S}$ can be *exponential* in the number of local spanning trees. This property has immediate consequences to probabilistic frequent subtree mining.

By considering exponentially many (implicit) global spanning trees instead of polynomially many ones, our technique has an improved performance in terms of *recall* over the simple algorithm described above. This improvement is only marginal on molecular graph datasets, due to the relatively simple graph structure of pharmacological compounds (cf. Horváth and Ramon, 2010; Horváth et al, 2010). On *threshold graphs*, however, which have applications among others in *spectral clustering* (see, e.g., von Luxburg, 2007), the boosted mining algorithm results in a much higher recall compared to the simple one. This also results in practical speedups, as it increases the number of patterns that are found in a given time budget with respect to the simple algorithm.

Exact Frequent Subtree Mining

The boosting technique mentioned above has implications for exact frequent subtree mining as well. We first note that despite more than two decades of research there are only a few non-trivial theoretical results concerning the complexity of frequent subgraph mining. In particular, if the transaction graphs are restricted to forests then frequent connected subgraphs (i.e., trees) can be generated with polynomial delay (see, e.g., Chi et al, 2005). Using the positive result of Matoušek and Thomas (1992), one can show that for graphs of bounded tree-width (Robertson and Seymour, 1986a) and bounded degree, frequent connected subgraphs can be generated with polynomial delay. In fact, frequent connected subgraphs can be listed in incremental polynomial time for graphs of bounded tree-width without restricting the vertex degree of the patterns (Horváth and Ramon, 2010). As a byproduct of our approach, we extend the known positive complexity results on frequent subgraph mining by a new one formulated for a graph class that is of theoretical as well as practical interest.

Our probabilistic frequent subtree mining algorithms solve the *exact* frequent subtree mining problem correctly (i.e., soundly and completely) if *all* spanning trees are considered for each database graph. As a result, frequent trees can be mined with polynomial delay if the number of spanning trees of the transaction graphs is bounded by a polynomial of their size by using the frequent subtree mining algorithm described above. The efficiency follows from the fact that all spanning trees of a graph can be listed with polynomial delay (Read and Tarjan, 1975) together with the positive result on frequent subtree mining in forest transaction databases.

The second (boosted) mining algorithm presented in this thesis extends this positive result to databases where transaction graphs may have exponentially many spanning trees. In particular, our boosted frequent pattern mining algorithm is correct and efficient (i.e., it has polynomial delay) if the transaction graphs have polynomially many *local* spanning trees. We call such graphs *locally easy*. The number of global spanning trees of a graph G might be exponential in the number of local spanning trees of G. As a result, the work in this thesis substantially extends the known positive results on efficient frequent subtree mining.

The class of locally easy graphs has some interesting properties that are both of theoretical and practical relevance. First, it is *orthogonal* to all graph classes that are defined by a constant upper bound on some *monotone* graph property (e.g., graphs of bounded tree-width); a graph property is called *monotone* if it is closed under taking subgraphs. By "orthogonality" we mean that the class always contains an infinite number of graphs that are not contained in the other graph class. The previously known positive results on efficient frequent subgraph mining, however, require the transaction graph class to be monotone.

Second, the class of locally easy graphs includes a number of interesting and practically relevant graph classes. *Forests* and *pseudoforests* in which every connected component has at most one cycle constitute two subclasses of locally easy graphs. Further subclasses can be defined by bounding the maximum number of biconnected blocks sharing a vertex by a constant. For example, *cactus* graphs (i.e., in which all edges belong to at most one simple cycle) of bounded block degree are locally easy. Note that even for cactus graph transactions (without bounding the block degree by a constant) frequent subtrees can be mined in incremental polynomial time (Horváth and Ramon, 2010). However, it is unknown whether this is possible with polynomial delay. We conjecture that generalizing our positive result on polynomial delay mining of frequent subtrees to the first natural graph class beyond locally easy graphs is at least as difficult as solving the **P** vs. **NP** problem. Our positive result on mining locally easy graphs is thus another step towards exploring the border between tractable and intractable fragments of the frequent pattern mining problem.

1.2.2. Fast Computation in Probabilistic Subtree Feature Spaces

We follow a similar strategy for the pattern matching operator used in the embedding step (ii) (see page 6): For an unseen graph G and a set $\mathcal{F}$ of tree patterns enumerated in the mining step, we generate a set $\mathfrak{S}(G)$ of (random) spanning trees of G and compute the set of all $H \in \mathcal{F}$ that are subgraph isomorphic to $\mathfrak{S}(G)$. The incidence vector of this set defines the embedding (vector) 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 H is subgraph isomorphic to G, but not to $\mathfrak{S}(G)$. On the other hand, this probabilistic pattern matching test can be performed in polynomial time while correct pattern evaluation is computationally intractable (that is, **NP**-complete). We prove that our probabilistic algorithm decides subgraph isomorphism from H into G correctly with high probability if H is frequent in G and the number of sampled spanning trees is chosen appropriately.

Using our probabilistic technique, we can compute the embedding vector of a graph in polynomial time by deciding subgraph isomorphism (with one-sided error) 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 partially ordered set of patterns. We show empirically that both algorithms drastically reduce the number of embedding operator evaluations compared to the baseline obtained by levelwise search.

In a last step, we improve the speed and space consumption of the above method by applying min-hashing (Broder, 1997). Each graph is represented by a small sketch vector that can be used to approximate Jaccard-distances. We show that the min-hash sketch of a given graph with respect to a set of tree patterns 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 this first one 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.

Our experimental results demonstrate that the proposed technique can dramatically reduce the number of subtree isomorphism tests, compared to an algorithm performing the embedding explicitly. We also show that even for a few random spanning trees per chemical compound, remarkable precisions of the active molecules can be obtained in a highly imbalanced chemical dataset by taking the i nearest neighbors of an active compound. Finally, we show that the predictive power of support vector machines using our approximate similarities compares favorably 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. While the presented embedding techniques are applied to probabilistic frequent subtree patterns in this thesis, they can be employed in other partially ordered pattern sets and monotone embedding operators.

1.3. Outline

The remainder of this thesis is structured as follows:

In Chapter 2 we introduce the necessary notions and notation. Among them, we define a quite general frequent subgraph mining problem and discuss its computational complexity in Section 2.2. It turns out that this complexity is related to the complexity of the HamiltonianPath problem. We discuss its complexity for an important special case in Appendix A. Chapter 3 presents related approaches for frequent subgraph mining and for the subgraph isomorphism problem.

The main contributions of this thesis are presented in the next three chapters. In Chapter 4 we introduce our relaxation of frequent subtree mining and show that it gives rise to an efficient algorithm for arbitrary graph transaction databases in theory and practice. Next, we investigate a more intricate algorithm that is able to decrease the error of our method with respect to the full set of frequent patterns for some types of graph databases in Chapter 5. In Chapter 6, we show how to efficiently compute the embedding vector for a graph, given a set of tree patterns. While our methods developed in the previous chapters can be used to do this in polynomial time using a brute-force approach, we propose several methods that speed up this computation by using a natural partial order on the pattern set. Finally, Chapter 7 concludes the thesis.

1.4. Previously Published Work

This thesis is based on joint work with Tamás Horváth and Stefan Wrobel which has already been published elsewhere. These publications are, in detail:

Pascal Welke, Tamás Horváth, Stefan Wrobel (2015) On the complexity of frequent subtree mining in very simple structures. In: Jesse Davis, Jan Ramon (eds) Inductive Logic Programming (ILP) Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 9046, pp 194–209, DOI 10.1007/978-3-319-23708-4_14

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

Pascal Welke, Tamás Horváth, Stefan Wrobel (2016b) Probabilistic frequent subtree kernels. In: Michelangelo Ceci, Corrado Loglisci, Giuseppe Manco, Elio Masciari, Zbigniew Ras (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

Pascal Welke (2017) Simple necessary conditions for the existence of a Hamiltonian path with applications to cactus graphs. CoRR abs/1709.01367, URL http://arxiv.org/abs/1709.01367

Pascal Welke, Tamás Horváth, Stefan Wrobel (2018) Probabilistic frequent subtrees for efficient graph classification and retrieval. Machine Learning 107(11):1847–1873, DOI 10.1007/s10994-017-5688-7

Pascal Welke, Tamás Horváth, Stefan Wrobel (2019) Probabilistic and exact frequent subtree mining in graphs beyond forests. Machine Learning DOI 10.1007/s10994-019-05779-1, (online)

2. Preliminaries

In this chapter we collect all necessary terminology and notation used in this thesis. We assume that the reader is aware of standard mathematical concepts like sets, vectors, functions, et cetera. We also require knowledge of complexity theory, in particular familiarity with the O-notation and the complexity classes **P** and **NP**. For the sake of clarity, however, we introduce our notation and recall some basic notions from graph theory (see, e.g., Diestel, 2012; Korte and Vygen, 2012) in Section 2.1. Subsequently, we formally define the pattern mining problem considered in this work in Section 2.2. As this problem is a listing problem, we define complexity classes for such problems, as they are less standard. We provide a generic algorithm and efficiency conditions in Section 2.2.1 and discuss the complexity of the pattern mining problem in Section 2.2.2. Finally, we describe the datasets used throughout this thesis in Section 2.4.

2.1. Notions and Notation

Sets and Lists

Let $S = \{s_1, s_2, \ldots\}$ be a set. We denote the cardinality of S by $|S|$ and the empty set by $\varnothing$. Given some fixed encoding of the elements of S, the $size(S)$ denotes the sum of the sizes of its elements in that encoding.[1] A subset X of S is indicated as $X \subseteq S$. If $X \subseteq S$ and $X \neq S$, we write $X \subset S$. We denote the set of natural numbers $\{1, 2, \ldots\}$ by $\mathbb{N}$ and the set of real numbers by $\mathbb{R}$. The finite set $\{1, \ldots, n\} \subset \mathbb{N}$ will be denoted by $[n]$ for all $n \in \mathbb{N}$. Given a finite set Σ, called ***alphabet***, a ***sequence***, also called ***list***, or ***string*** of elements of Σ is written as $[s_1, \ldots, s_n]$. In contrast to a set, the order of elements in a sequence is important. The length of a list L, i.e., the number of elements it contains, is denoted by $|L|$ and the ith element of L is denoted by $L[i]$. The set of all finite sequences over Σ is then denoted by Σ^*. Note that in a sequence a certain element s might occur multiple times. A ***permutation*** of Σ is a sequence that contains each element of Σ exactly once. There are exactly $|\Sigma|! = \prod_{i=1}^{|\Sigma|} i$ permutations of Σ.

Graphs

An ***undirected*** (resp. ***directed***) ***graph*** $G = (V, E)$ consists of a finite set V of vertices and a set $E \subseteq \{X \subseteq V : |X| = 2\}$ (resp. $E \subset V \times V$) of edges. When G is clear from the context, n denotes $|V|$ and m denotes $|E|$. Given $G = (V, E)$ we will often use the notation $V(G) := V$ and $E(G) := E$ to refer to the vertex or edge set of G. We consider only simple graphs,

[1] Hence, while in $|S|$ the sizes of the elements of S do not matter, in $size(S)$ they do matter.

i.e., loops and parallel edges are not permitted.[2] Unless otherwise stated, by ***graphs*** we mean undirected graphs. An edge $\{u,v\} \in E(G)$ will be denoted by uv. The set $\mathcal{N}(v) := \{w \in V(G) \,:\, vw \in E(G)\}$ is the set of ***neighbors*** of v. The cardinality of $\mathcal{N}(v)$ is called ***degree*** of v and denoted by $\delta(v)$. A ***leaf*** is a vertex that has exactly one neighbor.

A ***labeled*** graph is a graph G together with a function $l : E(G) \cup V(G) \to \Sigma$ that assigns a label from some finite set Σ to each vertex and each edge. Labeled graphs can model chemical molecules, protein-protein interactions, social networks, the Web graph and other phenomena. To keep the notation and description concise, we will state all results for ***unlabeled*** graphs. All our arguments naturally apply to labeled graphs as well.

A ***subgraph*** of G is a graph G' with $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. G' is a subgraph of G ***induced*** by a subset $V' \subseteq V(G)$ if $V(G') = V'$ and $uv \in E(G')$ if and only if $uv \in E(G)$ for all $u,v \in V'$. Such an induced subgraph is denoted by $G[V']$.

A graph is ***connected*** if for any $v,w \in V(G)$ v can be reached from w by traveling over a sequence of edges $[vv_1, v_1v_2, v_2v_3, \dots, v_iw]$. G is k***-connected*** if G is connected and the removal of any set of $k-1$ vertices does not destroy this property. We call a 2-connected graph ***biconnected***. A ***cycle*** is a minimal biconnected graph with at least three vertices, i.e., the removal of any edge or vertex results in a path. A ***block*** is a maximal subgraph of G with at least three vertices that is biconnected and a ***bridge*** is an edge that does not lie on any cycle. A ***biconnected component*** is a maximal subgraph that is biconnected, i.e., it is either a block, or a bridge, or an isolated vertex. Finally, a ***articulation vertex*** is a vertex whose removal increases the number of connected components of G.

A ***forest*** is a graph that contains no cycle; a tree is a connected forest. A ***path*** is a tree where at most two vertices are leaves[3]. Note that several works in the graph mining literature refer to trees as ***unrooted unordered trees*** or ***free trees***. In this work, a tree is always undirected, unless we explicitly say that it is rooted: A ***rooted tree*** T' can be obtained from a tree T by choosing a root $r \in V(T)$ and by directing every edge $e \in E(T)$ towards r. For each $r \in V(T)$ this definition results in a unique orientation of the edges. A ***spanning tree*** T of a connected graph G is a subgraph of G with $V(T) = V(G)$ that is a tree.

A common generalization of trees are graphs of bounded ***tree-width*** (Robertson and Seymour, 1986b). This property – in some sense – measures how tree-like a graph G is by defining a tree structure on certain induced subgraphs of G. A ***tree decomposition*** of a graph G is a tree T together with a mapping $bag : V(T) \to 2^{V(G)}$ such that

- $\bigcup\limits_{i \in V(T)} bag(i) = V(G)$,
- $e \subseteq bag(i)$ for some $i \in V(T)$ for all $e \in E(G)$, and
- the subgraph $T[\{i \in V(T) \,:\, v \in bag(i)\}]$ of T induced by all vertices whose bags contain v is connected for all $v \in V(G)$.

[2] A *loop* is an edge from a vertex to itself. Note that according to our definitions this can not happen in an undirected graph.

[3] The empty graph and a single vertex are paths, as well

The width of a tree decomposition is the cardinality of its largest bag, $\max_{i \in V(T)} |bag(i)|$. The tree-width of G is the minimum width of all tree decompositions of G. Note that the tree-width of any graph containing at least one edge is at least one. In fact, the tree-width of a graph G is one if and only if G is a forest.

Another generalization of trees are outerplanar graphs. A graph is ***outerplanar***, if (i) it can be drawn in the plane in such a way that edges do not cross each other except maybe in their endpoints and (ii) every vertex lies on the outer face. That is, each vertex can be reached from outside without crossing any edge. A graph G is outerplanar if and only if all its biconnected components are outerplanar (Harary, 1994). A biconnected component B of an outerplanar graph is either bridge, or it is a maximal induced subgraph of G composed of a single Hamiltonian cycle and possibly some non-crossing diagonal edges (Harary, 1994). A ***d-tenuous*** outerplanar graph is an outerplanar graph in which each block has at most d diagonals. Notice that forests are special outerplanar graphs. Furthermore, outerplanar graphs have tree-width at most two.

Isomorphism and Subgraph Isomorphism

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be graphs. They are ***isomorphic***, denoted $G_1 \cong G_2$ if there exists a bijection $\varphi : V_1 \to V_2$ with $uv \in E_1$ if and only if $\varphi(u)\varphi(v) \in E_2$ for all $u, v \in V_1$. In this case φ is called an ***isomorphism*** between G_1 and G_2. 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. The corresponding function $\varphi : V(G_1) \to V(G_2)$ is called ***subgraph isomorphism***. $G_1 \prec G_2$ denotes that $G_1 \preccurlyeq G_2$ and G_1 is not isomorphic to G_2.

As already mentioned in the beginning of this section, we will state all our results for unlabeled graphs. Nonetheless, we will shortly mention how the subgraph isomorphism is defined in the labeled case, as many datasets considered in this work consist of labeled graphs (cf. Section 2.4). Let G_1 and G_2 be two graphs with label functions l_1 and l_2 over the same label set Σ. Then G_1 and G_2 are isomorphic if there exists a subgraph isomorphism φ such that

- $l_2(\varphi(v)) = l_1(v)$ for all $v \in V(G_1)$
- $l_2(\varphi(v)\varphi(w)) = l_1(vw)$ for all $vw \in E(G_1)$.

Checking whether the label of an edge or vertex matches the label of its image can be done in constant time assuming that a suitable encoding is chosen for the elements in the ***finite*** label set Σ. Hence the algorithms considered in this work can be easily extended to the labeled case.

The SubgraphIsomorphism problem, that is, deciding whether $H \preccurlyeq G$ for two graphs H and G, is one of the classical **NP**-complete problems (Garey and Johnson, 1979, Sec. 3.2.1). This negative result holds even for the case that the pattern H is restricted to trees. We will refer to this latter problem as the SubtreeIsomorphism problem. Its **NP**-completeness follows e.g. from the restriction to the problem of deciding whether there exists a Hamiltonian path in G, i.e., a path that contains all vertices of G (Garey and

Johnson, 1979, Sec. 4.2.2). In case that G and H are both trees, the SubtreeIsomorphism problem can be solved in polynomial time (Matula, 1968). Several algorithms were developed for this problem and are discussed in Section 3.1.

The GraphIsomorphism problem, that is, deciding whether two graphs G and H are *isomorphic*, is neither known to be solvable in polynomial time, nor known to be **NP**-complete. Recent work by Babai (2015), claims that there is a quasi-polynomial time algorithm for the GraphIsomorphism problem. In case both G and H are trees, the GraphIsomorphism problem becomes much simpler and can be decided in linear time (this follows, e.g., from Hopcroft and Wong, 1974).[4] One way of solving the GraphIsomorphism problem is to compute and compare canonical strings, which we will describe below. First, however, we need additional notation.

Isomorphism is an equivalence relation. That is, for all graphs G_1, G_2, G_3, it holds

Reflexivity $G_1 \cong G_1$,

Symmetry $G_1 \cong G_2$ if and only if $G_2 \cong G_1$, and

Transitivity if $G_1 \cong G_2$ and $G_2 \cong G_3$ then $G_1 \cong G_3$.

A *graph class* $\mathcal{G}$ is a set of graphs. We will often define graph classes by some common property of the graphs, e.g. the class of all trees, connected graphs, etc. A *representative of* $\mathcal{G}$ is a set that contains exactly one graph from each equivalence class $\bar{G} := \{H \in \mathcal{G} : H \cong G\} \subseteq \mathcal{G}$. That is, the elements in the representative of $\mathcal{G}$ are unique up to isomorphism.

Posets

A *partially ordered set*, or *poset* $(S, <)$ is a set S together with a binary relation $<$ such that for all $x, y, z \in S$ the following conditions hold:

Reflexivity $x < x$,

Antisymmetry if $x < y$ and $y < x$ then $x = y$, and

Transitivity if $x < y$ and $y < z$ then $x < z$.

For any graph class $\mathcal{F}$ the pair $(\mathcal{F}, \preccurlyeq)$ is a poset. If $\mathcal{F}$ is finite, we can represent $(\mathcal{F}, \preccurlyeq)$ by a directed 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 $T_1, T_2 \in \mathcal{F}$. In this way, $x < y$ if and only if there exists a directed path from x to y in the graph $(\mathcal{F}, E)$.

A *total order* is a poset $(S, <)$ where for all $x, y \in S$ either $x < y$ or $y < x$. A total order on S can be represented as a sequence of elements of S. In particular, each permutation of S defines a total order on S. A *topological order* on $(\mathcal{F}, E)$ (resp. $(\mathcal{F}, \preccurlyeq)$) is a sequence (i.e., a total order) $\mathcal{F} = [T_1, T_2, \ldots, T_n]$ satisfying $i < j$ for all $(T_i, T_j) \in E$. A directed graph has a topological order if and only if it is acyclic, i.e., if it corresponds to a partial order (see, e.g. Korte and Vygen, 2012).

[4] This result also holds in case at least one of the graphs is a tree: Whether a graph is a tree can be decided in linear time and a tree can never be isomorphic to a graph that is not a tree.

Let $(\Sigma, <)$ be a total order. The *lexicographical order* induced by $(\Sigma, <)$ is a total order $(\Sigma^*, <')$ of strings over Σ defined as follows: Let S and S' be two strings with $|S| = |S'|$. If $S \neq S'$ let i be the first position where the two strings differ. Then $S <' S' \Leftrightarrow S[i] < S'[i]$. If two strings have different length, we obtain their relation (with respect to $<'$) by padding the shorter string by a novel element not contained in Σ that is larger than all elements in Σ (with respect to $<$). The lexicographical order on canonical strings of trees (described below) is an important part of efficiently removing isomorphic graphs efficiently from a graph class, i.e., computing a representative of the graph class.

Canonical Forms and Canonical Strings of Trees

Given a graph class $\mathcal{G}$, a *canonical string function* is a function $f : \mathcal{G} \to \Sigma^*$ such that

$$f(G) = f(H) \iff G \cong H$$

for all $G, H \in \mathcal{G}$ for a suitable finite alphabet Σ. For some $G \in \mathcal{G}$, we call $f(G)$ its *canonical string*.

Canonical strings allow to decide if two graphs are isomorphic by just testing the equality of their canonical strings. This operation can be done in linear time in the length of the shorter string (assuming that the size of Σ is a constant and that comparing elements in Σ can be done in constant time). Canonical strings also enable testing if a graph G is isomorphic to some graph in a set S of graphs in an efficient way: Using prefix trees (Fredkin, 1960), this can be done in linear time in $|f(G)|$; in particular, the runtime does not depend on $|S|$. The generic graph mining algorithm described in Section 2.2.1 below requires a way to filter out redundant patterns, i.e., patterns such that some isomorphic pattern has been evaluated before. This test can be easily implemented using canonical strings stored in a prefix tree.

A canonical string function that can be computed in polynomial time for all graphs is unlikely to exist as this would imply a polynomial time algorithm for the Subgraph-Isomorphism problem. Polynomial time computable canonical string functions are known, however, for restricted graph classes. If $\mathcal{G}$ is the class of all trees (Asai et al, 2003; Chi et al, 2003; Nijssen and Kok, 2003), the class of all outerplanar graphs (Horváth et al, 2010), or the class of all planar graphs (Hopcroft and Wong, 1974). In fact, there exist several canonical string functions for trees that can be computed in linear time (e.g. Hopcroft and Wong, 1974).

Although our system does not require any specific canonical string function, we now briefly describe the method we have used. It is based on the ideas presented by Chi et al (2003). To obtain a canonical string of a tree G, we first transform G to a rooted tree. To select a root, we repeatedly remove all leaves together with their incident edges from G until one or two vertices remain. These vertices are called the *center* (respectively the *bi-center*) of G and are well defined. The method to obtain the center or bi-center of G can be implemented in $O(|V(G)|)$ (see, e.g., Harary, 1994, Chapter 4). We define the canonical string for G to be the canonical string of the tree rooted at the center of G if G has a center. If G has a bi-center then the canonical string of G is defined to be the lexicographically smaller of the two canonical strings of the trees rooted at the two bi-centers of G.

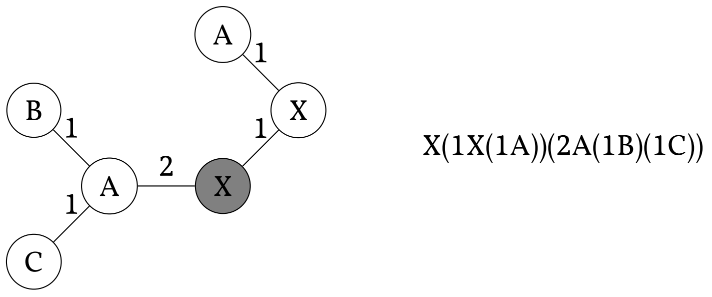


Figure 2.1.: A small tree (left) with its canonical string (right). The vertex shown in gray is the center of the tree.

To obtain a canonical string for a rooted tree labeled by elements from Σ, we start with an arbitrary but fixed total order on the label set $\Sigma \dot\cup \{(,)\}$ and employ a bottom up method. The canonical string of a leaf vertex v is simply its label $l(v)$. For some vertex v that is not a leaf of G we first recursively compute the canonical string c_w for each child w of v. We add the "(" symbol and $l(vw)$ at the beginning of c_w and the ")" symbol at the end of c_w. Then we sort all modified c_ws according to the lexicographical order on the modified Σ^*. This results in a canonical string function for trees that can be computed in polynomial time (Chi et al, 2003). Figure 2.1 shows a small example.

2.2. Frequent Connected Subgraph Mining

Frequent connected subgraph mining has been an area of active research for the last twenty years. It is a natural generalization of the frequent itemset mining problem (Agrawal et al, 1996) to transaction databases consisting of graphs. We will review related work in Chapter 3 and will now formally define the exact computational problem considered in this thesis. In its most general form, it can be formulated as follows:

FrequentConnectedSubgraphMining (FCSM) problem: *Given* a finite list $\mathcal{D} \subseteq \mathcal{G}$ (called *graph database*) for some graph class $\mathcal{G}$ and an integer threshold $t \in [|\mathcal{D}|]$, *list* all graphs $P \in \mathcal{P}$ for some graph class $\mathcal{P}$, called the *pattern class*, that are subgraph isomorphic to at least t graphs in $\mathcal{D}$. The patterns in the output must be pairwise non-isomorphic.

An equivalent definition of this problem uses a *relative* (instead of an absolute) frequency threshold. For a given database $\mathcal{D}$ and a threshold $t \in [|\mathcal{D}|]$ the relative frequency θ is defined as $\theta := \frac{t}{|\mathcal{D}|}$. We will use the relative and absolute thresholds interchangeably when clear from the context. For some pattern $H \in \mathcal{P}$, and database $\mathcal{D}$, we say that H is *t-frequent* (respectively *θ-frequent*) if H is subgraph isomorphic to at least t (respectively $\theta \cdot |\mathcal{D}|$) graphs in $\mathcal{D}$. Note that for some fixed input $\mathcal{D}$ and t, the output of the FCSM problem is always a finite set of pairwise non-isomorphic graphs, i.e. a finite representative of a graph class $\mathcal{F} \subseteq \mathcal{P}$. We will often refer to the output of the FCSM problem as the frequent subgraphs of $\mathcal{D}$, omitting the threshold t (resp. θ) when clear from the context. Note also that our notation explicitly allows multiple instances of the same graph in $\mathcal{D}$.

Other authors considered variants of the FCSM problem that are parametrized by the transaction class only (see, e.g., Horváth and Ramon, 2010). In contrast, we regard a problem that is parameterized by the transaction class *and the pattern class*. The reason is that we are interested in mining frequent trees in graph databases that contain more complex graphs. A formalization that does not distinguish between pattern class and transaction class would always require to list the frequent patterns that are not trees as well, if there are any.

In Section 2.2.1 we give an algorithm for the FCSM problem and formulate sufficient conditions for $\mathcal{G}$ and $\mathcal{P}$ that guarantee the algorithm to generate frequent patterns efficiently. These conditions may be of some independent interest for the study of other special cases of the FCSM problem. Furthermore, in Section 2.2.2 we characterize a relationship between the computational complexity of the FCSM problem and the question whether **P** = **NP**. To this end, we first review efficiency notions for listing problems like the FCSM problem.

Starting from Chapter 3, we focus on the special case of the FCSM problem where $\mathcal{P}$ is the class of trees. This special case will be referred to as FrequentSubtreeMining (FTM) problem. In Chapter 4 and Chapter 5 we will use the generic algorithm to obtain positive complexity results for the FTM problem without restricting the transaction graph class.

Computational Complexity of Enumeration Problems

We will now define notions of efficiency for problems like the FCSM problem. A *listing problem* is a pair $(X, \{O(x) : x \in X\})$, where X is a set of valid input instances and $O(x)$ is the set of all *acceptable solutions* for each $x \in X$. An acceptable solution $o \in O(x)$ is a finite set; an algorithm *solving* a particular listing problem $(X, \{O(x) : x \in X\})$ is an algorithm that outputs for each $x \in X$ all elements in some $o \in O(x)$ exactly once in some arbitrary order. We assume that the listing problems are *polynomially balanced*. That is: $size(G) \leq p(size(x))$ for all $x \in X$, for all $o \in O(x)$, and for all $G \in o$ for some fixed polynomial p (see, e.g. Boley, 2011, Chap. 2.2).

Consider the FCSM problem for transaction class $\mathcal{G}$ and pattern class $\mathcal{P}$: A valid input instance $x \in X$ consists of a graph database $\mathcal{D} \subseteq \mathcal{G}$ and an integer threshold $t \in [|\mathcal{D}|]$. Note that the definition above allows for *multiple correct solutions* $o \in O(x)$: An acceptable solution for x is a maximal set of pairwise nonisomorphic graphs $o \subseteq \mathcal{P}$ such that each graph $G \in o$ is t-frequent in $\mathcal{D}$. There are many such sets, as one can freely choose to replace a pattern with an isomorphic graph that is not identical. However, each acceptable solution is a representative of the same graph class, that is, the elements in o are pairwise nonisomorphic and represent the *class of all graphs that are t-frequent in $\mathcal{D}$*. As a subgraph can have at most as many vertices and edges as its supergraph, the size of each t-frequent subgraph is smaller or equal to the size of the largest graph(s) in the input database; hence this listing problem is polynomially balanced.

In contrast to the well known decision problems, where the output consists of a single bit (interpreted as "Yes" or "No", "Accept" or "Fail", et cetera) the output of a listing problem may vary in size. Consider, for example the task of listing all subsets of a set S. There are $2^{|S|}$ subsets that need to be listed. Hence a correct algorithm can never return the full set in time polynomial in the size of the input as writing the output alone takes exponential time. Therefore, it is common practice to take the size of the output into account when defining efficiency measures for listing problems.

For listing problems, the following *output sensitive complexity measures* are distinguished in the literature (see, e.g., Boley, 2011; Johnson et al, 1988). Suppose an algorithm $\mathfrak{A}$ for some listing problem $(X, \{O(x) : x \in X\})$ gets $x \in X$ as input and outputs some $o \in O(x)$ as a sequence $[p_1, p_2, \cdots, p_n]$ of patterns. Then $\mathfrak{A}$ generates o

- with *polynomial delay*, if the time before the output of p_1, between the output of any two consecutive elements p_i, p_{i+1}, and between the output of p_n and the termination of $\mathfrak{A}$ is bounded by a polynomial of $size(x)$,
- in *incremental polynomial time*, if the algorithm outputs p_1 in time bounded by a polynomial of $size(x)$, the time between outputting p_i and p_{i+1} is bounded by a polynomial of $size(x) + \sum_{j=1}^{i} size(p_j)$, and the time between the output of p_n and termination is bounded by a polynomial of $size(x) + size(o)$.
- in *output polynomial time*, if the algorithm outputs the elements of o in time bounded by a polynomial of $size(x) + size(o)$.

Clearly, polynomial delay implies incremental polynomial time, which, in turn, implies output polynomial time. It is an open problem whether the first two classes are identical, or not. In frequent itemset mining, for example, the FP-Growth algorithm (Han et al, 2004) lists frequent itemsets with polynomial delay, while the Apriori algorithm (Agrawal et al, 1996) does so in incremental polynomial time. We note, however, that the Apriori algorithm can easily be transformed into a polynomial delay algorithm by retaining the output of frequent patterns (Horváth and Ramon, 2010).

The FCSM problem and the FTM problem can *not* be solved in output polynomial time. This follows directly from the negative result in (Horváth et al, 2007) which we will describe below. One way to obtain positive results is to restrict the transaction graph class $\mathcal{G}$ in the FCSM problem. We will review such approaches in Chapter 3. This thesis, however, restricts the pattern class to the class of trees while not restricting the transaction graph class.

2.2.1. A Generic Levelwise Mining Algorithm

We obtain the main results of this thesis by adapting a generic levelwise search mining algorithm to our problem setting. Levelwise search (Mannila and Toivonen, 1997) is one of the most common techniques in pattern mining that can be used to efficiently mine frequent patterns for a broad range of problem settings. Its most popular application is the Apriori algorithm (Agrawal et al, 1996) for frequent itemset mining. In order to find

Algorithm 2.1 A generic levelwise graph mining algorithm.

input: $\mathcal{D} \subseteq \mathcal{G}$ for some graph class $\mathcal{G}$, a pattern class $\mathcal{P}$, and $t > 0$ integer
output: all frequent subgraphs of $\mathcal{D}$ that are in $\mathcal{P}$

```
1: let $\mathcal{S}_0 \subseteq \mathcal{P}$ be the set of frequent pattern graphs consisting of a single vertex
2: for ($l := 0; \mathcal{S}_l \neq \emptyset; l := l + 1$) do
3:     set $\mathcal{S}_{l+1} := \emptyset$ and $\mathcal{C}_{l+1} := \emptyset$
4:     for all $P \in \mathcal{S}_l$ do
5:         print $P$
6:         for all $H \in \rho(P) \cap \mathcal{P}$ satisfying $H \notin \mathcal{C}_{l+1}$ do
7:             add $H$ to $\mathcal{C}_{l+1}$
8:             if SupportCount$(H, \mathcal{D}) \geq t$ then
9:                 add $H$ to $\mathcal{S}_{l+1}$
```

a pattern in level $l + 1$, it completely explores all levels up to l. On the one hand, this strategy is disadvantageous if one is interested in mining *large* frequent patterns, as all (usually exponentially many) subpatterns need to be evaluated first. On the other hand, it allows for very efficient pruning and it allows for an incremental polynomial time pattern generation in frequent subgraph mining even for some **NP**-complete pattern matching operators (Horváth and Ramon, 2010).

Algorithm 2.1 is a generic levelwise search algorithm for the FCSM problem. It is a slight modification of the related algorithm by Horváth and Ramon (2010); the only changes are in Lines 1 and 6. It calculates the set of candidate (resp. frequent) patterns of level l in the set variable $\mathcal{C}_l$ (resp. $\mathcal{S}_l$). In Line 6 it computes the set $\rho(P)$ of refinements of a pattern P obtained from P by extending it with an edge in all possible ways. Due to this fact, Algorithm 2.1 is often referred to as a *generate-and-test* algorithm That is, it either adds a new vertex w to P and connects it to any vertex in $V(P)$ by an edge, or it connects two vertices in $V(P)$ that have not been connected yet.[5] Clearly, $|\rho(P)|$ is bounded by $O\left(|V(P)|^2\right)$. Subroutine SupportCount$(H, \mathcal{D})$ in Line 8 returns the number of graphs $G \in \mathcal{D}$ with $H \preccurlyeq G$.

It is shown by Horváth and Ramon (2010) that the original version of Algorithm 2.1 mines frequent patterns with polynomial delay if patterns and transactions satisfy certain conditions. These conditions have however been formulated for the case that the pattern and transaction graph classes are the same. In the theorem below we generalize these conditions to the case that $\mathcal{P}$ and $\mathcal{G}$ can be different.

Theorem 2.1. *Let $\mathcal{G}$ and $\mathcal{P}$ be the transaction and pattern graph classes satisfying the following conditions:*

1. *All graphs in $\mathcal{P}$ are connected. Furthermore, $\mathcal{P}$ is closed downwards under taking subgraphs, i.e., for all $H \in \mathcal{P}$ and for all connected graphs H' we have $H' \in \mathcal{P}$ whenever $H' \preccurlyeq H$.*

[5] For the case of tree pattern generation, the second type of extension can be omitted, as it always results in cycles. Hence, in this case $|\rho(P)| = |V(P)|$.

2. *The membership problem for $\mathcal{P}$ can be decided efficiently, i.e., for any graph H it can be decided in polynomial time if $H \in \mathcal{P}$.*

3. *Subgraph isomorphism in $\mathcal{P}$ can be decided efficiently, i.e., for all $H_1, H_2 \in \mathcal{P}$, it can be decided in polynomial time if $H_1 \preccurlyeq H_2$.*

4. *Subgraph isomorphism between patterns and transactions can be decided efficiently, i.e., for all $H \in \mathcal{P}$ and $G \in \mathcal{G}$, it can be decided in polynomial time if $H \preccurlyeq G$.*

Then Algorithm 2.1 solves the FCSM *problem with polynomial delay in the size of $\mathcal{D}$ for $\mathcal{P}$ and $\mathcal{G}$.*

The proof of this theorem is very similar to the proof in (Horváth and Ramon, 2010). We nevertheless give it for completeness.

Proof. Let $\mathcal{G}$ and $\mathcal{P}$ be two graph classes such that Conditions 1–4 hold and let $\mathcal{D} \subseteq \mathcal{G}$. We first prove that Algorithm 2.1 is correct (i.e., sound and complete) and irredundant. The soundness is immediate from Lines 6 and 8. To show the completeness, let $H \in \mathcal{P}$ be frequent in $\mathcal{D}$. We prove by induction on $|E(H)|$ that it will be generated by the algorithm. The proof of the base case that H consists of a single vertex is straightforward by Line 1. For the inductive step we have that H has a vertex with degree one or an edge that can be removed without disconnecting H. Let H' be the graph obtained from H by deleting such a vertex (and the edge adjacent to it) or such an edge. By construction, H' is connected and hence $H' \in \mathcal{P}$ follows from Condition 1 as $H' \preccurlyeq H$. Furthermore, H' is frequent in $\mathcal{D}$ as any subgraph of a frequent graph must be frequent. Therefore H' will be generated by Algorithm 2.1 by the induction hypothesis. Furthermore, as $H \in \rho(H') \cap \mathcal{P}$, we have $H \in \mathcal{C}_{|E(H)|}$ by Lines 6 and 7. Therefore, H is added to $\mathcal{S}_{|E(H)|}$ because it is frequent (Line 9), completing the proof of completeness. Finally, the proof of irredundancy is immediate from the condition tested in Line 6.

Regarding the delay, the time before outputting the first pattern (or termination if there is no frequent vertex) is linear in the size of the database. We can count the frequency of a singleton pattern by a single scan over the database. We now show that the time needed for Lines 6–9 is polynomial in the size of $\mathcal{D}$. Conditions 1–3 imply that there is a canonical string representation (i.e., a string unique modulo isomorphism) for all graphs in $\mathcal{P}$ that can be computed in polynomial time. We can store $\mathcal{S}_i$ and $\mathcal{C}_i$ as prefix trees of canonical strings of patterns. In this way, we can add and look up patterns in $\mathcal{S}_i$ or $\mathcal{C}_i$ in time linear in the size of the canonical string of a pattern. $|\rho(H)|$ is polynomial in the size of H and thus polynomial in the size of $\mathcal{D}$. Therefore, by Condition 2, $\rho(H) \cap \mathcal{P}$ can be computed in polynomial time. $H \notin \mathcal{C}_{l+1}$ can be checked in time linear in the size of the canonical string representation of H. SUPPORTCOUNT can be implemented by iterating over $\mathcal{D}$, checking for each graph $G \in \mathcal{D}$ if $H \preccurlyeq G$, and maintaining a counter; by Condition 4 it runs in polynomial time in the size of $\mathcal{D}$. Overall, the time between printing consecutive patterns and the time between printing the last pattern and termination is polynomial in the size of $\mathcal{D}$. Actually, we have shown that the delay of Algorithm 2.1 depends polynomially on the number of graphs in $\mathcal{D}$ and on the runtime of the check for Condition 4. We will use this property in Chapter 5. □

Note that the conditions above allow for mining frequent patterns that do not belong to $\mathcal{G}$. Furthermore, they enable the generation of restricted subsets of *all* frequent subgraphs of some database $\mathcal{D}$. For example, we can mine frequent paths in transaction databases consisting of trees. We will utilize the latter property when restricting $\mathcal{P}$ to trees.

How to efficiently address Condition 4 is usually left out by other graph mining algorithms. In fact, most graph miners focus on efficient candidate enumeration, instead of embedding computation. The literature typically justifies this by showing experimental results on chemical graph databases, where the mining systems are fast. In this thesis, we go in the opposite direction, focusing on the embedding operator, and prove worst case complexity bounds. We refer the reader to the related work (e.g. Chi et al, 2005) for a discussion of efficient candidate generation.

Finally, although it is not required by Theorem 2.1, the complexity of deciding membership in the transaction class $\mathcal{G}$ is a crucial (practical) issue. For some well defined graph classes, e.g., graphs of tree-width at most k, membership is computationally intractable if k is not a constant (Arnborg et al, 1987). Therefore deciding whether a given graph mining algorithm can be applied efficiently (i.e., whether $\mathcal{D} \subseteq \mathcal{G}$) might already be intractable. Even worse, the speed of many existing frequent subgraph mining systems (e.g., Kuramochi and Karypis, 2004; Nijssen and Kok, 2005) often depends on some graph properties that are not formally stated and hence not testable. We aim to avoid this question by choosing a problem relaxation that allows efficient algorithms for arbitrary transaction graph classes. However, we discuss the membership problem for a particular graph class in Section 5.3, as one of our relaxed algorithms allows for an exact solution to the FTM problem for this novel graph class.

2.2.2. The Computational Complexity of Frequent Subtree Mining

After giving sufficient conditions for polynomial delay mining of frequent patterns, we want to investigate the complexity of the FTM problem. We will see that the question whether frequent tree mining is possible in output polynomial time is equivalent to the question whether **P** = **NP** for a broad range of transaction graph classes. For the remaining transaction graph classes, interestingly, frequent subtree mining is connected to the complexity of the HamiltonianPath problem. Informally, we can say that proving results on the efficiency of frequent tree mining will likely be very difficult. This, among other reasons, leads us to investigate a suitable relaxation of the FTM to obtain a practical system that fulfills the requirements mentioned in the introduction. An overview of the results of this section can be found in Figure 2.2.

Horváth et al (2007) have shown that the frequent connected subgraph mining problem cannot be solved in output polynomial time if $\mathcal{P} = \mathcal{G}$ is the class of all graphs. Their proof can be generalized to our more general problem definition that allows the pattern and the transaction graph class to be different. Using their result, we are able to show Theorem 2.2 below. This result implies that the FTM problem cannot be solved in output polynomial time for arbitrary transaction graphs, unless **P** = **NP**; this is even true if

	The HamiltonianPath problem is in **P** for $\mathcal{G}$	The HamiltonianPath problem is **NP**-complete for $\mathcal{G}$
The SubtreeIsomorphism problem is **NP**-complete for $\mathcal{G}$	No polynomial delay frequent tree mining possible ⇒ **P** ≠ **NP** (Theorem 2.1) However, efficient mining might be possible without implying anything on **P** versus **NP**	Polynomial delay frequent tree mining possible ⇔ **P** = **NP** (Theorem 2.2)
The SubtreeIsomorphism problem is in **P** for $\mathcal{G}$	Polynomial delay frequent tree mining is possible (Theorem 2.1)	▨

Figure 2.2.: The relationship between the complexities of the HamiltonianPath and SubtreeIsomorphism problems for transaction graphs from some graph class $\mathcal{G}$ and the complexity of the FTM problem.

we only try to list all frequent paths. The FTM problem stays **NP**-hard for a broad range of transaction graph classes, e.g., the class of planar graphs.[6] We will investigate some further implications below.

Theorem 2.2. *Let $\mathcal{G}$ and $\mathcal{P}$ be graph classes that contain the class of paths and let the* HamiltonianPath *problem be **NP**-complete in $\mathcal{G}$. Then the following things are equivalent:*

1. ***P** = **NP***
2. *The* FTM *problem for $\mathcal{G}$ and $\mathcal{P}$ can be solved with polynomial delay.*
3. *The* FTM *problem for $\mathcal{G}$ and $\mathcal{P}$ can be solved in output polynomial time.*

Proof. Let $\mathcal{G}$ be a graph class in which the HamiltonianPath problem is **NP**-complete.

"1⇒2": If **P** = **NP** then the subgraph isomorphism problem can be decided in polynomial time. Hence, by Theorem 2.1 we can find all frequent subtrees in any finite subset of $\mathcal{G}$ with polynomial delay.

[6] For an overview of graph classes where the HamiltonianPath problem is **NP**-complete, we refer the reader to `http://graphclasses.org/classes/problem_Hamiltonian_path.html`.

"2⇒3": If polynomial delay mining is possible, this immediately implies that mining in output polynomial time is possible.

"3⇒1": Suppose the FTM problem can be solved in output polynomial time for transactions from $\mathcal{G}$. Let $G \in \mathcal{G}$ and P be a path with $|V(P)| = |V(G)|$. As $\{G, P\} \subseteq \mathcal{G}$, all 2-frequent subtrees in this database are listed in output polynomial time (by assumption of 3). The number of output patterns is bound by $|V(G)|$ and hence the mining algorithm terminates in time polynomial in the size of G. The HAMILTONIANPATH problem for G can be decided by checking whether the path P is 2-frequent. Hence **P** = **NP**, as we have just given a polynomial time algorithm for an **NP**-complete problem. □

The proof by Horváth et al (2007) for "3⇒1" uses that the output set has a linear number of elements in the size of the input database; there are exactly $n + 2$ nonisomorphic paths of length at most n.[7] Therefore an output polynomial time algorithm for this particular instance is in fact a polynomial time algorithm in the *input size*. Note that this technique can not necessarily be generalized to different **NP**-complete decision problems (e.g., the CLIQUE problem); a similar database consisting of a clique and some other graph G would have an exponential number of 2-frequent subgraphs.

An important result of this fact is that the proof above holds for all pattern classes that contain the class of all paths. In particular, Theorem 2.2 holds for the class of paths and the class of all connected graphs, as well. A "Proof by Restriction" analogously to decision problems (Garey and Johnson, 1979)[8] for listing problems does, however, not work in general. Without the particular structure of the proof of Theorem 2.2 we could not conclude that the nonexistence of an output polynomial time algorithm for the FTM problem implies the nonexistence of an output polynomial time algorithm for the FCSM problem with some pattern class that contains the class of all trees. For the same input, the output of the two problems might differ; allowing additional elements in the output reduces the output sensitive complexity for the same overall runtime. In the proof of "3⇒1", however, the output is restricted to the polynomially large set of all frequent *paths* by construction of the database. Hence the output sensitive efficiency measures for the different frequent graph mining problems have the same input (i.e., the input *and* the output of the problem).

Theorem 2.2 implies that for many graph classes the existence of an output polynomial time algorithm is equivalent to the existence of a polynomial delay mining algorithm: This is certainly the case if **P** = **NP**. Now suppose **P** ≠ **NP**. Then this is equivalent to the nonexistence of a polynomial delay tree mining algorithm and to the nonexistence of an output polynomial time tree mining algorithm for graph classes where the HAMILTONIANPATH problem cannot be solved in polynomial time. Putting this together, for transaction graph classes, where the HAMILTONIANPATH problem is **NP**-complete, polynomial delay mining, incremental polynomial time mining, and output polynomial time mining are either all possible or all impossible, i.e., here the complexity hierarchy collapses. This also answers a question posed by Horváth and Ramon (2010) for a large num-

[7] Including the empty graph and the singleton graph, containing only one vertex.

[8] That is, **NP**-completeness of the HAMILTONIANPATH problem implies the **NP**-completeness of the SUBGRAPHISOMORPHISM problem.

ber of graph classes: Incremental polynomial time subtree or subgraph mining for computationally intractable embedding operators is not possible for this type of transaction graph classes, unless **P** = **NP**.

An Open Problem

After seeing the negative result for frequent tree mining above, it is natural to ask how far we can go if we are nonetheless interested in mining frequent trees efficiently. That is, is frequent tree mining possible with polynomial delay or at least in output polynomial time in all graph classes where the HAMILTONIANPATH problem is in **P**? We do not know the answer and the question whether efficient frequent subtree mining is possible remains open even for the transaction graph class of *cactus* graphs, one of the simplest extensions of the class of forests. A cactus graph is a graph where every biconnected block is a simple cycle. The HAMILTONIANPATH problem can be decided for cactus graph transactions in polynomial time. This follows from (Matoušek and Thomas, 1992) by noting that paths have vertex degree at most two and cactus graphs have tree-width at most two. In fact, this problem can be solved in linear time as shown in Appendix A. The SUBTREEISOMORPHISM problem, however, is already **NP**-complete for cactus graph transactions (Akutsu, 1993). We can now show the importance and high difficulty of this open problem by discussing the potential two answers separately.

(i) Suppose the problem *can* be solved with polynomial delay. An important immediate consequence of this result would be that polynomial delay frequent pattern enumeration is possible even for **NP**-complete pattern matching operators, solving an open problem (cf. Horváth and Ramon, 2010).

(ii) Suppose it *cannot* be solved with polynomial delay. Then, as the class of trees satisfies Conditions 1–3 of Theorem 2.1, by contraposition we have that Condition 4 of Theorem 2.1 does *not* hold, i.e., the corresponding subgraph isomorphism problem is *not* in **P**. But this would immediately imply that **P** ≠ **NP**, indicating the high difficulty of proving this case, as the subgraph isomorphism problem lies in **NP** for all pattern and text graph classes. Note that this consideration applies also to the particular case of cactus transaction graphs.

As a result, there is a strong connection between the complexity of the HAMILTONIANPATH problem and the SUBTREEISOMORPHISM problem on the one hand and the FTM problem on the other hand for a given transaction graph class. Figure 2.2 collects the results from the above considerations, from Theorem 2.1, and from Theorem 2.2.

We conjecture that case (ii) holds, that is, polynomial delay pattern generation is impossible for computationally intractable pattern matching operators. This is certainly true for graph classes for which the HAMILTONIANPATH problem is **NP**-complete[9]. If our conjecture holds, then it implies that in case of intractable pattern matching operators, the primary question should be whether the pattern mining problem at hand can be

[9] We note that the HAMILTONIANPATH problem is polynomial for the case of cactus graphs, making them an especially interesting candidate graph class. See, also, Appendix A.

solved in incremental polynomial time, and not to show that polynomial delay pattern mining is not possible. Furthermore, even for very simple graph classes, polynomial delay exact frequent subtree mining is most likely very difficult to achieve.

In the remainder of the thesis we hence focus on a relaxation of the frequent subtree mining problem that allows us to easily guarantee polynomial delay by giving up the demand on completeness of the output. As a result of one of our algorithms for this relaxed problem, however, we will discuss an exact algorithm for a novel graph class in Section 5.3. The open problem for cactus graph transactions formulated above shows the significance of this result. We discuss this connection in Section 5.4.

2.3. Embedding Computation

Once we have found the set of frequent subgraphs or subtrees of a given graph database, we are usually interested in doing something with them. First, we could use the patterns directly, for example by manually inspecting them to gain knowledge about the dataset. Another useful application is to use the patterns as a representation language for graphs in the input dataset and, more generally, graphs drawn from the same or a similar distribution. A common way of defining the similarity between two graphs is to compute some similarity measure of their images in the Hamming-cube $\{0,1\}^{|\mathcal{F}|}$ spanned by the elements of the set of frequent patterns $\mathcal{F}$. The binary feature vectors can then be regarded as the incidence vectors of subsets of $\mathcal{F}$. Given a fixed set $\mathcal{F}$ of frequent patterns, we define a *feature space* as the power set $2^{\mathcal{F}}$ of $\mathcal{F}$ and a feature map as $f_{\mathcal{F}} : G \mapsto \{H \in \mathcal{F} : H \preccurlyeq G\}$. The image of G under $f_{\mathcal{F}}$ is called *embedding* of G. For the experimental evaluation of the frequent subtree miners we develop in this thesis, we will often use the corresponding embeddings of graphs, equipped with a suitable metric or kernel function for metric learning. Formally the task of embedding computation in the context of frequent subtree mining is defined as follows:

Tree Embedding Computation (TEC) Problem: *Given* a graph G and a finite set $\mathcal{F}$ of trees, *list* all trees $P \in \mathcal{F}$, that are subgraph isomorphic to G.

Note that the TEC problem is a special case of the FCSM problem for *finite* $\mathcal{P} = \mathcal{F}$, $\mathcal{D} = \{G\}$, and $t = 1$ if $\mathcal{F}$ is closed downwards with respect to subgraph isomorphism. Hence, we can apply a variant of Algorithm 2.1 to solve the problem, as it is possible to decide whether a tree H is contained in a finite set of trees in linear time in the size of H (cf. Section 2.1). However, we discuss more efficient alternatives in Chapter 6.

Jaccard Similarity

One similarity function that we investigate in this thesis on the embedding vectors discussed above is the Jaccard similarity. Given two binary feature vectors $\vec{f_1}$ and $\vec{f_2}$ representing the sets S_1 and S_2, respectively, their *Jaccard-similarity* is defined by

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

with $\textsc{Sim}_{\text{Jaccard}}(\varnothing,\varnothing) := 0$ 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 by the following fast probabilistic technique based on *min-hashing* (Broder, 1997): For a permutation π of $\mathcal{F}$ and feature vector $\vec{f}$, define $h_\pi(\vec{f})$ to be the index of the first entry with value 1 in the permuted order of $\vec{f}$. One can show that the following correspondence holds for the feature vectors $\vec{f}_1$ and $\vec{f}_2$ above (see Broder, 1997, for the details):

$$\textsc{Sim}_{\text{Jaccard}}(S_1,S_2) = \mathbf{P}\left[h_\pi(\vec{f}_1) = h_\pi(\vec{f}_2)\right] \quad ,$$

where the probability $\mathbf{P}$ 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 $\vec{f}_1$ and $\vec{f}_2$: Generate a set $\pi_1,\dots,\pi_K$ of permutations of the feature set uniformly at random and return K'/K, where K' is the number of permutations π_i with $h_{\pi_i}(\vec{f}_1) = h_{\pi_i}(\vec{f}_2)$. 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.

2.4. Datasets

Any general frequent subgraph mining algorithm is expected to process a broad spectrum of graph databases. Most empirical evaluations, however, concentrate on some particular type of graph data, mostly representing small molecules. These graphs share certain properties, e.g. sparsity, small vertex degree, near planarity, and, in particular, a natural set of frequent patterns corresponding to functional groups. While all these properties (especially the last one) motivate frequent subgraph mining in the first place, it is also important to observe the behavior of a mining technique on data that may or may not have such properties. We therefore conducted experiments on molecular, social, and artificial datasets. Table 2.1 gives an overview of key statistics of these datasets. Below we briefly describe their semantics and how we obtained them.

MUTAG (Debnath et al, 1991) is a dataset of 188 connected compounds labeled according to their mutagenic effect on Salmonella typhimurium. On average, each graph has 20 vertices and 22 edges.

PTC contains 344 connected molecular graphs, labeled according to the carcinogenicity in mice and rats. The graphs have 26 vertices and edges on average. The dataset was released as part of the Predictive Toxicology Challenge (see `https://www.predictive-toxicology.org/ptc`) held in 2000 and 2001.

NCI1, NCI109 (Wale et al, 2008) consist of 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. The average number of vertices is 30, the average number of edges is 32 in both datasets.

Dataset	# Graphs	# Vertices	# Edges	max. Degree
MUTAG	188	17.93 ± 4.58	19.79 ± 5.68	4
NCI1	4 110	29.87 ± 13.56	32.30 ± 14.93	4
NCI109	4 127	29.68 ± 13.57	32.13 ± 14.96	5
PTC	344	25.56 ± 16.25	25.96 ± 17.06	4
NCI-HIV	42 688	45.71 ± 23.68	47.71 ± 24.57	12
ZINC	8 946 757	42.39 ± 6.38	43.92 ± 6.55	5
ER-1.0	50	25.70 ± 13.91	24.90 ± 13.77	8
ER-1.4	50	23.52 ± 14.60	31.90 ± 20.08	9
ER-1.8	50	27.06 ± 14.02	48.24 ± 25.39	11
ER-2.0	50	28.32 ± 12.60	56.72 ± 25.93	11
Brownian Motion	200	30.00 ± 0.00	43.47 ± 5.03	9
POKEC neighbors	100	78.72 ± 110.29	182.70 ± 300.40	75
POKEC disks	100	79.72 ± 110.29	261.42 ± 405.43	1 031
HEPPH neighbors	1 000	19.92 ± 47.87	888.31 ± 4 548.81	361
HEPPH disks	1 000	20.92 ± 47.87	908.23 ± 4 593.35	491
ENRON neighbors	1 000	26.61 ± 74.35	199.55 ± 827.49	420
ENRON disks	1 000	27.61 ± 74.35	226.16 ± 898.94	1 244

Table 2.1.: Statistics of our evaluation datasets. We report the number of graph transactions, average number of vertices and edges per graph, and the maximum degree of any vertex in each dataset.

NCI-HIV consists of 42 687 compounds of which 39 337 are connected. The average number of vertices and edges per graph are 41 and 43, respectively. The molecules are annotated with their activity against the human immunodeficiency virus (HIV). In particular, they are labeled by "active" (A), "moderately active" (M), or "inactive" (I). We consider the following three usual binary classification problems: (AMvsI) A and M together versus I, (AvsMI) A versus M and I, and (AvsI) A versus I where instances labeled by M are removed.

ZINC is a subset of 8 946 757 (8 946 755 connected) so called 'Lead-Like' molecules from the zinc database (Irwin et al, 2012) of purchasable chemical compounds. The molecules in this subset have a molar mass between $250g/mol$ and $350g/mol$ and have an average number of 43 vertices and 44 edges.

POKEC is a popular social network in Slovakia with roughly 1.6 million users and more than 30 million directed edges. The crawl by Takac and Zabovsky (2012) includes various public attributes of the users (e.g. gender, age, eye color, etc.). We consider an unlabeled version of this graph as well as a labeled one, where each vertex is labeled by the corresponding user's gender (or "unknown" if the information was not available). In both cases the edges are unlabeled.

HEPPH is a co-authorship network extracted from the arXiv preprint server. Vertices correspond to authors of papers in the area of high energy physics; an undirected edge connects two authors if they coauthored a paper. The graph consists of 12 008 unlabeled vertices and 118 521 unlabeled edges. This snapshot was originally released as a part of 2003 KDD Cup (see Gehrke et al, 2003, for an overview).

ENRON is an email communication network (a first description appeared in Klimt and Yang, 2004). Nodes are email accounts of employees of the Enron company and their acquaintances and an undirected edge exists if there was at least one email sent between two accounts. The graph contains 183 831 edges and 36 692 vertices and is unlabeled.

Erdős-Rényi Datasets All these datasets consist of 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 have different structural complexity, where the structural complexity is defined as the *expected edge factor* $q = \frac{m}{n}$ (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. Then we set the Erdős-Rényi edge probability parameter $p = \frac{2q}{n-1}$, and finally generate G on n vertices in the usual way with this p. This implies that the expected number of edges for a graph on n vertices is $q \cdot n$. If the resulting graph is connected, we add it to the dataset. The vertices and edges may be labeled by choosing a label from fixed sets of vertex and edge labels uniformly at random.

Brownian Motion Datasets To construct such a graph database, we first draw n points from the two-dimensional unit cube independently and uniformly at random and label them with c different labels[10] at random for some $c > 0$ integer. Given a parameter $d \in (0, \sqrt{2}]$, we construct a *threshold graph* by connecting two points if and only if their two-dimensional Euclidean distance is at most d. Subsequent graphs in the database are obtained by (i) moving each point randomly according to a normal distribution with standard deviation μ centered at its former position and (ii) constructing a threshold graph on the resulting set of points with respect to the same threshold d as above. If a point would leave the unit square due to its random move, it is reflected back inside. Hence, a database constructed in this way depends on the parameters n, c, d, μ, and N, where N is the number of time steps (or equivalently, the number of graphs in the database).

We process each social network above by replacing directed with undirected edges (removing duplicate edges and singleton vertices). In order to obtain a graph transaction database from such a social network we consider the graphs induced by the neighborhoods of the vertices. Such small graphs arising from social networks are often

[10] The number of vertex labels has a nontrivial influence on the number of (non-isomorphic) spanning trees of graphs and also on the number of frequent patterns in a graph database.

called *ego nets*. The first dataset variant, which we call *disk*, contains also the vertex itself, while the second variant, called *neighborhood* only considers the neighbors with the central vertex removed. More formally, the disk transaction database of a social network G is the set $\{G[\{v\} \cup \mathcal{N}(v)] : v \in V(G)\}$, while the neighborhood variant is the set $\{G[\mathcal{N}(v)] : v \in V(G)\}$. The disk variant results in connected graphs with a central vertex of high degree, while the neighborhood variant results in mostly disconnected graphs. Table 2.1 shows the number of such ego nets that we extracted, the average number of vertices and edges in these ego nets, and the maximum degree of any vertex in the ego net transaction database.

Data Sources

We obtained the datasets MUTAG, NCI1, NCI109, and PTC from http://www.di.ens.fr/~shervashidze/code.html. The NCI-HIV dataset can be found at http://cactus.nci.nih.gov/. We use a version that was provided by Tamás Horváth. The ZINC dataset is available at http://zinc.docking.org/subsets/lead-like. We downloaded a copy in August 2014. The social datasets POKEC, YOUTUBE, HEPPH, and ENRON are available at http://snap.stanford.edu. For the generation of the artificial datasets we have written a small program.

3. Related Work

We will now review the state-of-the-art of exact and approximate solutions to the FCSM and FTM problem. In particular, we are interested in (i) identifying cases where efficient mining is possible and (ii) reviewing the proposed graph mining algorithms with respect to efficiency (in the sense of Section 2.2.2). As we have seen in the preliminary experiments in Section 1, this question is not only of theoretical, but also of practical interest. Practical algorithms that reliably work on a broad range of transaction graph databases are scarce.

The first (theoretical) task is mostly concerned with reviewing algorithms for the SubtreeIsomorphism and SubgraphIsomorphism problems. These can be used as embedding operators in Algorithm 2.1 and thus yield complexity results for both problems. The second (practical) task is concerned with reviewing various frequent subgraph mining algorithms that were proposed over the last two decades. We will see that all of these algorithms are (i) restricted to certain pattern or transaction graph classes and guarantee some worst-case runtime or (ii) work for arbitrary graph patterns and transactions, without giving worst-case runtime guarantees. There have been several reviews on the topic of frequent subgraph mining (Chi et al, 2005; Jiang et al, 2013; Krishna et al, 2011; Wörlein et al, 2005). However, those surveys only consider papers that propose frequent subgraph mining algorithms, skipping the theoretical results obtainable and treating the computational complexity only as a minor issue.

Frequent subgraph mining has been an active area of research over the last twenty years. Reviewing all related work is hence impossible, forcing us to focus on the most relevant work. We therefore restrict this review to papers that consider a database of small to medium sized graphs as input where a graph pattern is considered to be frequent if it is subgraph isomorphic to at least a certain number of graphs in the database. That is, we restrict our review to articles that consider frequent subgraph mining algorithms in the sense of Definition 2.2 and relevant relaxations of this task. In the literature, this scenario is often called the *transactional setting* of frequent subgraph mining. Furthermore, we do not consider parallelization efforts, as they are orthogonal to the main technical contributions in this thesis.

Another direction of frequent subgraph mining research that can be clearly distinguished considers the *single graph setting*. Here, the task is to find all graphs that are "frequent" in a single (large) graph, for varying definitions of frequency (see, e.g., Bringmann and Nijssen, 2008, for a few such notions). There are also several articles concerned with frequent tree mining that, in fact, solve problems different to the FTM problem. Variations include (i) the type of graph database, (ii) the type of patterns, (iii) the notion of frequency, and (iv) the embedding operator. In particular, various types of "tree" transactions and patterns are considered, e.g., rooted trees (Asai et al, 2003; Chi et al, 2004a;

Nijssen and Kok, 2003; Termier et al, 2002) or rooted ordered trees (Asai et al, 2004; Zaki, 2002). The survey article of Chi et al (2005) gives an excellent overview on the state of the art before 2005 and organizes the work along dimensions (ii) and (iv). The survey article of Jiang et al (2013) extends the temporal scope to the year 2013 and also addresses frequent subgraph mining algorithms.

The transactional setting, i.e., the FTM or FCSM problem, has lead to a lot of research on its own. Between circa 2002 and 2008 various research groups worked on this topic and published articles and software prototypes. It seems, however, that around the year 2008 the general interest in novel algorithms faded and many people moved on to parallelizing existing algorithms (compare Jiang et al, 2013; Petermann et al, 2017) or solving different problem formulations using variations of the existing algorithms. Only a few groups kept working on frequent subgraph mining in the transactional setting after 2008, focusing e.g. on theoretically efficient algorithms (Horváth and Ramon, 2010) or different problem formulations resulting in approximations of the FCSM problem. We are not aware of any truly novel algorithmic approaches for the exact FCSM or FTM problem that resulted in practical algorithms for this setting which were released after 2008.

Jiang et al (2013) conclude that this is due to the maturity of the field. We disagree; our review shows that existing algorithms are either restricted to very simple graph classes or have exponential delay in the worst case. We have mentioned in Chapter 1 that this in fact restricts such graph mining algorithms exactly to chemical graph databases. We will show in Section 4.2 that the state-of-the-art graph mining algorithms (which all have exponential delay in the worst case) are inapplicable on several non-chemical datasets. In fact, there is no clear way to predict whether the graph miners in the literature will be fast or inapplicable on a given dataset, which heavily restricts their usefulness, e.g. in a data exploration setting. Only recently, Schulz et al (2018) presented the first algorithm that can efficiently compute an approximation of the set of frequent trees on arbitrary transaction databases. Hence we conclude that additional work is required to obtain algorithms that are applicable in broader settings; most likely this task involves focusing more on efficient embedding operators.

Outline

We organize the literature review as follows: In Section 3.1 we discuss algorithms for the SubgraphIsomorphism problem that can be applied in Algorithm 2.1 to obtain a frequent subgraph mining algorithm. Section 3.2 reviews existing integrated algorithms. In particular, frequent tree mining algorithms (Section 3.2.1), frequent subgraph mining algorithms (Section 3.2.2), and algorithms that approximate the set of frequent subgraphs in some sense (Section 3.2.3)

3.1. Algorithms for the SubgraphIsomorphism Problem

We have discussed the complexity of the FTM problem in Section 2.2. Theorem 2.1 states that polynomial delay mining of frequent trees is possible if the SubtreeIsomorphism problem (resp. SubgraphIsomorphism problem for more general patterns[1]) can be decided in polynomial time for a given transaction graph class $\mathcal{G}$. Various algorithms have been proposed for restricted versions of the SubgraphIsomorphism problem; possibly even more special cases have been shown to remain **NP**-complete. There is no way to give a complete overview on the work that was done in this area, and in Chapter 5 we will develop yet another algorithm for the SubtreeIsomorphism problem in a novel class of transaction graphs. To this end we discuss the most relevant results that relate to our work. In particular, we focus on the complexity of the SubtreeIsomorphism problem and review the most commonly used practical approach for solving the general SubgraphIsomorphism problem in the context of frequent connected subgraph mining.

Perhaps most prominently, the SubtreeIsomorphism problem can be solved in polynomial time if the pattern graph H is a tree and the transaction graph G is a forest. Various efficient algorithms have been proposed in the last fifty years (Chung, 1987; Lingas, 1983; Matula, 1968, 1978; Shamir and Tsur, 1999; Verma and Reyner, 1989). However, the positive result for the SubtreeIsomorphism problem does not hold if the pattern graph is allowed to be disconnected, i.e., the SubgraphIsomorphism problem is **NP**-complete even for forest transactions (Garey and Johnson, 1979, Theorem 4.6).

The currently fastest known algorithm for the SubtreeIsomorphism problem for a tree pattern H and forest transaction G requires

$$O\left(\frac{|V(G)| \cdot |V(H)|^{1.5}}{\log |V(H)|}\right)$$

time (Shamir and Tsur, 1999). All these algorithms have in common that they are dynamic programming algorithms. Generally, the SubtreeIsomorphism problem is solved by bottom-up evaluation over a rooted version of one of the two trees. This is done by combining partial subgraph isomorphisms of the children of the current vertex by solving bipartite matching instances. The algorithms mainly differ in the details of the evaluation and how efficiently they solve the matching instances. We will describe a generalization of the algorithm presented in (Shamir and Tsur, 1999)[2] in Chapter 5 and hence skip a more detailed discussion here. Any one of the algorithms for tree patterns and forest transactions can be used for the approach developed in Chapter 4.

The positive result on forest transactions has motivated the question whether the SubtreeIsomorphism problem remains feasible for larger transaction graph classes. As already mentioned in Chapter 2, a common generalization of trees are graphs of bounded

[1] Obviously, a polynomial time algorithm for the SubgraphIsomorphism problem for some graph class $\mathcal{G}$ that contains the class of trees implies a polynomial time algorithm for the SubtreeIsomorphism problem for transactions in $\mathcal{G}$.

[2] Their algorithm is a simplification of the algorithm presented in (Chung, 1987) with a more elegant solution for the bipartite matching problems.

tree-width. Matoušek and Thomas (1992) have extended the dynamic programming approach from trees to transaction graphs where the tree-width is bounded by a constant k. Their algorithm runs in polynomial time if the pattern graph is connected and has vertex degree k, or is k-connected, where k is some constant. If such a constraint is not imposed, the SUBGRAPHISOMORPHISM problem remains **NP**-complete for bounded tree-width transactions. Their algorithm first computes a tree decomposition T of G and employs a bottom-up evaluation over some rooted version of T. It builds partial subgraph isomorphisms for the graph induced by the bags of the current vertex and its descendants. In fact, many otherwise hard problems can be solved in polynomial time in a similar way if the tree-width of a graph is bounded by a constant. The work by Matoušek and Thomas (1992) has been generalized to pattern graphs that have *log-bounded fragmentation* by Hajiaghayi and Nishimura (2007). A graph G has log-bounded fragmentation if the removal of k vertices results in at most $O(k \log |V(G)|)$ connected components. Graphs with maximum vertex degree $O(\log |V(G)|)$ are log-bounded fragmentation graphs (for any $0 \leq k \leq |V(G)|$). It is important to note, however, that these results imply only that frequent subtrees can be mined efficiently in bounded tree-width transactions if there is an additional restriction on the vertex degree of the trees.

A different, more restrictive[3], generalization of forests are *almost k-forests*. A graph G is an almost k-forest if each block B of G has at most $|V(B)| + k$ edges. Akutsu (1993) has shown that the SUBGRAPHISOMORPHISM problem can be solved for connected patterns and almost k-forests of bounded vertex degree. Again, if we drop the restriction on the vertex degree, Akutsu has shown that the SUBGRAPHISOMORPHISM problem is **NP**-complete even for tree patterns and almost 0-forest transactions (i.e., cactus graphs). Such graphs are also called cactus graphs and are outerplanar. This implies that the SUBTREEISOMORPHISM problem for outerplanar graphs is **NP**-complete. As outerplanar graphs have tree-width at most two, we have a clear distinction of the complexity of the SUBTREEISOMORPHISM problem based on the tree-width of the transaction graph class: The SUBTREEISOMORPHISM problem is in **P** for the class of graphs with tree-width at most one and is **NP**-complete for the class of all graphs with tree-width k for all $k \geq 2$, unless we restrict the vertex degree of the pattern tree to be a constant.

Marx and Pilipczuk (2014) systematically investigated the complexity of the SUBGRAPHISOMORPHISM problem for several combinations of pattern and transaction graph classes. They consider the tractability of the problem if one imposes (constant) bounds any combination of ten parameters, containing, e.g., number of vertices, number of connected components, maximum vertex degree, and tree-width, for pattern, or transaction. Table 1 in their article shows a large number of restrictions of the SUBTREEISOMORPHISM problem that are **NP**-complete even if some parameters of the transaction graphs (and even the pattern trees) are constant. Most of their (maximal) positive results require either the number of vertices or the maximum vertex degree of the pattern to be constant. The only exception (at least at first glance) relevant to the SUBTREEISOMORPHISM problem is that the SUBGRAPHISOMORPHISM problem between patterns with a constant number of connected components and transactions with bounded genus,

[3] In the sense that an almost k-forest has bounded tree-width.

bounded feedback vertex set and bounded vertex degree (sic) can be decided in polynomial time. This, however, restricts the vertex degree of the pattern tree as well: If there exists a subgraph isomorphism then the maximum vertex degree of the pattern must not be larger than the maximum vertex degree of the transaction. In this sense, the above case is no exception from the rule that efficient SUBTREEISOMORPHISM algorithms for more general graph classes than forests exist only for patterns of bounded vertex degree. In contrast, the SUBTREEISOMORPHISM algorithm discussed in Chapter 5 is efficient when both pattern and transaction graph have unbounded vertex degree. The efficiency of our algorithm depends only on the restriction of a less general branching property of the *transaction* graph.

3.1.1. Embedding Lists and Exponential Algorithms

As positive complexity results even for restrictions of the SUBTREEISOMORPHISM problem are scarce, some researchers have investigated algorithms for the SUBGRAPHISOMORPHISM problem that do not guarantee bounded worst-case runtimes but are fast in practice on many types of graphs that arise in application settings. To this end, Ullmann (1976) proposed to explicitly compute the set of all subgraph isomorphisms from a pattern graph H to a transaction graph.[4] His algorithm fixes an order $[v_1, v_2, \dots, v_{|V(H)|}]$ of the vertices of H and considers the sequence $[H_1, H_2, \dots, H_{|V(H)|} = H]$ of induced subgraphs $H_i = H[\bigcup_{j=1}^{i} v_j]$. It computes the set

$$EL(H_{i+1}, G) := \{\varphi : \varphi : V(H_{i+1}) \to V(G) \text{ is a subgraph isomorphism}\}$$

by extending each subgraph isomorphism $\varphi \in EL(H_i, G)$ to subgraph isomorphisms from H_{i+1} to G as follows: The algorithm checks whether the novel vertex v_{i+1} in H_{i+1} is compatible to φ. That is, whether there exists a vertex $w \in V(G)$ that is not yet part of the image of φ and is connected to all images of the neighbors of v_{i+1} in H_{i+1}. Hence each $\varphi \in EL(H_i, G)$ can be extended to up to $|V(G)| - i$ isomorphisms from H_{i+1} to G. The algorithm either terminates by finding a subgraph isomorphism from H to G or stops after finding a subgraph isomorphism from some H_i to G, but none from H_{i+1} to G.

This method works well for chemical graphs and some other workloads (see, e.g., Nijssen and Kok, 2005; Zhao and Yu, 2008). Due to a moderate number of vertex and edge labels, high sparsity and (almost) planarity of chemical graphs the sizes of the sets $EL(H_i, G)$ tend to be small. The runtime of Ullmann's algorithm is strongly influenced by the total number of subgraph isomorphisms that exist from any H_i in the selected sequence $[H_1, H_2, \dots, H_{|V(H)|} = H]$ to G. This number is bounded by

$$\sum_{i=1}^{|V(H)|} |EL(H_i, G)| \le \sum_{i=1}^{|V(H)|} \binom{|V(G)|}{|V(H_i)|} \cdot |V(H_i)|! \,.$$

This bound is best possible. Consider the case that G is an unlabeled complete graph: For each permutation of each k-sized subset of the vertices of G there exists a unique isomorphism from (any graph) H with $|V(H)| = k$. Hence, the number of subgraph isomor-

[4] Checking whether this set is empty, or not, obviously solves the SUBGRAPHISOMORPHISM problem.

phisms that need to be computed may be exponential in the size of G and facultative in the size of H. There is no known way to compute or estimate the exact number of such embeddings (an exact solution in polynomial time would solve the SubgraphIsomorphism problem). Hence it is required to run the algorithm and wait whether it terminates in feasible time and does not consume all available memory for storing embeddings.

On the other hand, however, this method is easy to implement and particularly well-suited for the workload of frequent subgraph mining systems. For a breadth-first or depth-first mining algorithm all (resp. one) subgraphs of any pattern graph H were already enumerated and the SupportCount method has already been evaluated. Hence, we have (in the notation from above) already computed $EL(H_{|V(H)|-1}, G)$ for some suitable direct predecessor[5] $H_{|V(H)|-1}$ of $H = H_{|V(H)|}$. If we store all embeddings for all patterns from the previous level, we can hence compute the set of embeddings of H into any graph G in the database, by reusing the embeddings of the predecessor pattern.

In practice, it seems to be the case that a low average vertex degree in combination with a moderately-sized set of possible vertex and edge labels dramatically reduces the number of possible subgraph isomorphisms. Empirical evaluations of the existing frequent subgraph mining systems indicate that this approach works well on chemical graphs and some other databases. There is generally no guarantee that it is always the case. In fact, we will see in Section 4.2 that there are many practically relevant graph databases where the runtime and space requirements of Ullmann's algorithm explode for no apparent reason.

An extension of Ullmann's algorithm is due to Cordella et al (1998, 1999). For a given pattern H and a text graph G, the authors incrementally construct embeddings from subgraphs of H into G similar to the algorithm of Ullmann (1976). They propose to add a pruning step during the extension of an embedding. Their algorithm not only checks whether the embedding can be grown by a single vertex, but also whether the image v' of the novel vertex v has enough free neighbors to map v's not yet mapped neighbors to it. If this is not the case, no subgraph isomorphism from H to G can exist that maps v tho v'. This speeds up the algorithm both in theory and in practice. The proposed pruning strategy, however, cannot be used in the incremental fashion described above: The neighbors of v are not known at the time of the extension of the embedding. (The novel vertex is always the last vertex missing to construct a complete embedding of the current pattern graph H.) Hence all neighbors[6] of v in H are already mapped to some vertex of G in the current embedding.

[5] When mining graphs that may contain cycles, the notions are slightly modified to allow the extension to work edge-by-edge, not vertex-by-vertex. In the context of the FTM problem, however, both notions are equivalent.

[6] If H is a tree, there is of course only one such neighbor

3.2. Algorithms for the FCSM Problem

Over the last twenty years, a lot of research has focused on practical implementations of frequent subgraph mining systems. Most of this work has focused on the efficient enumeration of candidate patterns and on canonicalization of the patterns to avoid duplicates (Chi et al, 2005; Jiang et al, 2013). Most graph mining papers address the support counting step in a less detailed manner, either citing some off the shelf subgraph isomorphism algorithm, or roughly sketching ways to keep track of *all* possible embeddings of patterns into the transaction database (compare Section 3.1.1). Jiang et al (2013) suggest that this is due to the fact that the subgraph isomorphism problem is seen as "harder to address". Hence more work is spent to reduce the number of calls to the subgraph isomorphism subroutine as much as possible. While this is an important issue, the bulk of the computational effort for medium to large graph databases is still to evaluate the embedding operator for candidate patterns on database transactions (Wörlein et al, 2005). This is particularly relevant, as it is even the case for chemical graph databases, where the subgraph isomorphism algorithms described in Section 3.1.1 are very fast.

In this thesis, we go into the opposite direction. Contrary to most of the related work we focus on the embedding operator. We will hence review the related work with special regard to the embedding computation techniques it employs. Furthermore, we will also take special interest in the kind of graph databases and the related algorithms used for their evaluation (if any). For literature reviews focusing more on other aspects, we refer the reader to (Borgelt, 2009; Chi et al, 2005; Jiang et al, 2013). As it turns out, only (Chi et al, 2003; Horváth and Ramon, 2010) use efficient embedding operators to solve the FCSM in incremental polynomial time. (Horváth and Ramon, 2010) is mainly a theoretical result as it was not implemented. Thus the only existing implementation with guaranteed worst-case delay (by Chi et al) can only mine trees in forest transactions.

Unless stated otherwise, all practical systems consider labeled graph databases. That is, each vertex and edge is assigned a unique element from a finite set of symbols, called labels. The respective embedding operators are extended to the labeled case, analogously to the definition of subgraph isomorphism (cf. Section 2.1). Table 3.1 gives an overview of the existing exact or approximate FTM and FCSM algorithms for the transactional setting. We will describe them in more detail below.

3.2.1. Frequent Tree Mining Algorithms

Among the practical implementations of frequent subgraph mining algorithms, frequent tree mining algorithms are most closely related to our work here. Several algorithms have been proposed for computing the set of frequent trees in databases of trees, forests, or "arbitrary" graphs. As shown in Section 2.2, frequent subtrees can be enumerated efficiently (i.e., with polynomial delay) in forest transaction databases. However, most systems do not use an efficient embedding operator and hence may result in exponential delay and memory consumption even in this case.

Name	Reference	Transactions	SubgraphIsomorphism	Comment
FreeTreeMiner	Chi et al (2003)	Forests	Chung (1987) (polynomial)	
HybridTreeMiner	Chi et al (2004a)	Forests	Embedding lists (exponential)	
FreeTreeMiner	Rückert and Kramer (2004)	Graphs	support sets (exponential)	
F3TM	Zhao and Yu (2008)	Graphs	Ullmann (1976) (exponential)	
FSG	Kuramochi and Karypis (2004)	Graphs	Embedding lists (exponential)	Mines all frequent subgraphs
MoSS	Borgelt and Berthold (2002) Borgelt et al (2005)	Chemical Graphs	Embedding lists (exponential)	Mines all frequent subgraphs
gSpan	Yan and Han (2002)	Graphs	Cordella et al (1998) (exponential)	Mines all frequent subgraphs
FFSM	Huan et al (2003)	Graphs	Embedding lists (exponential)	Mines all frequent subgraphs
Gaston	Nijssen and Kok (2004) Nijssen and Kok (2005)	Graphs	Embedding lists (exponential)	Can mine paths, trees, and cyclic patterns
–	Horváth and Ramon (2010)	Bounded Tree-Width	specialized incr. pol. time	Mines all frequent subgraphs
Summarize-Mine	Chen et al (2009)	Graphs	Embedding lists (exponential)	Mines a random subset of all frequent subgraphs
MUSE	Zou et al (2010)	Uncertain Graphs	Embedding lists (exponential)	
REAFUM	Li and Wang (2015)	Graphs	Embedding lists (exponential)	β subgraph isomorphism
–	Schulz et al (2018)	Graphs	Dalmau et al (2002) (polynomial)	Partially Injective Homomorphism results in superset of frequent trees

Table 3.1.: An overview on related frequent subtree and subgraph mining systems for forest and graph transaction databases. Unless stated otherwise, these methods enumerate the full set of frequent subtrees and are our direct competitors.

FreeTreeMiner by Chi et al (2003) solves the FTM problem for tree databases. This work introduces tree mining as an area of research and develops the first[7] algorithm that uses canonical representations of trees for efficient pattern generation. The authors propose a canonical string representation for trees and a levelwise algorithm to mine all frequent trees in a tree database. Based on their particular canonical representation, they argue that all frequent trees can be generated by either joining two frequent trees $H+e, H+e'$ with a common parent H that differ in exactly one edge, or by extending the frequent tree H by a single edge f such that the resulting tree has a larger height[8]. Duplicate candidate generation is reduced[9] by identifying nontrivial automorphisms of H and some support counting steps are avoided by first checking whether all possible parent patterns of $H + e$ are frequent. Chi et al use the efficient algorithm of Chung (1987) to compute the support of a candidate tree pattern in the tree database. They evaluate their algorithm on a chemical dataset, an IP multi-cast dataset that represents one-to-many streaming topologies on the Internet, and on synthetic datasets.

HybridTreeMiner by Chi et al (2004a) also solves the FTM problem for tree databases, and, in addition, the problem of mining rooted trees in databases of rooted trees. Hence the name of the algorithm. There are two main differences to their FreeTreeMiner algorithm above: First, they use a DFS approach, instead of a BFS approach and second, they propose a novel way of counting the support. Now, the authors resort to embedding lists but use them in a smart way that requires only one pass over the database. If a candidate pattern $H+e+e'$ is generated by joining two frequent patterns $H+e, H+e'$, its support can be computed by joining the support lists of the parent patterns: Two embeddings are compatible, if they are identical on H, and map the endpoints of e and e' to different vertices. All embeddings for $H + e + e'$ can therefore be constructed by combining such compatible embeddings. The extension operation works in a similar way by combining compatible embeddings of H and the frequent tree corresponding to the single edge f. In this way, an explicit access to the graph database is not necessary after initially computing the embedding lists of all frequent tree patterns consisting of single edges. They evaluate their algorithm on a chemical tree dataset and on a synthetic tree dataset and compare it to FreeTreeMiner (discussed above). They show that this approach is faster by an order of magnitude. Interestingly, the IP multi-cast dataset is not considered in this study. In (Chi et al, 2004b), they extended this system to mine only closed frequent subtrees or maximal frequent subtrees.

FreeTreeMiner by Rückert and Kramer (2004) solves the FTM problem in databases containing cyclic graphs. The authors propose a canonical string representation that allows their candidate generation process to reduce the number of duplicate evaluations of candidate patterns. They define the *height* of a vertex in a tree pattern as the distance to the root of the canonical representation and generate patterns by only extending on leaves

[7] Zaki (2002) introduced "tree mining" before, but considered rooted ordered trees and a different embedding operator.

[8] With respect to its canonical representation which is a rooted tree.

[9] The authors claim to avoid duplicate candidate enumeration by identifying pattern automorphisms. They do not prove, however, that their technique guarantees nonredundant candidate enumeration. FreeTreeMiner additionally compares canonical strings of candidate patterns.

with largest height. When evaluating the frequency of a candidate pattern by computing all of its embeddings explicitly, the algorithm at the same time computes the embedding lists for all extensions by a single edge. All extensions of $height(H) + 1$ are obtained by combining such single edge extensions. These candidate patterns are only recursively extended if they are in canonical form. The authors do not prove the correctness of their algorithm (neither soundness, completeness, nor irredundancy) and evaluate their algorithm on the AIDS database.

F3TM by Zhao and Yu (2008) similarly solves the FTM problem in databases containing cyclic graphs using a depth-first search over the pattern space. They focus on the candidate generation step and employ an iterative version of (Ullmann, 1976) for the support counting step that is intertwined with the candidate generation step. In particular, for a frequent pattern H they explicitly store a subset of all subgraph isomorphisms in an embedding list. The authors focus on the candidate generation step and show that the complete set of frequent patterns of a dataset can be obtained by extending the patterns only on a well defined subset of their vertices, resulting in fewer duplicated candidate patterns. The number of candidate patterns is further reduced by considering automorphisms of the patterns and by considering only pattern extensions that are actually present in some transaction graph. The authors evaluate their algorithm on a variant of the AIDS database considered also in this thesis (cf. Section 2.4) and on artificial data obtained with the generator of Kuramochi and Karypis (2001). In (Zhao and Yu, 2007) they extend F3TM to mine closed frequent trees.

3.2.2. Frequent Subgraph Mining Algorithms

FSG was initially proposed 2001 by Kuramochi and Karypis (2004). It implements Algorithm 2.1 (i.e., it is a levelwise algorithm) for mining all frequent subgraphs in graph transaction databases. To compute the support of a candidate pattern, FSG stores the support set of each frequent pattern and intersects the support sets of parent patterns to reduce the number of explicit subgraph isomorphism tests to be evaluated for any candidate pattern: The downward closure property ensures that a candidate can only be subgraph isomorphic to those graphs where all of its subpatterns are present and hence only such graphs must be explicitly evaluated using the embedding operator. This method requires space that is proportional to support set of each frequent pattern in two consecutive levels of the pattern lattice (the current and the previous level). The authors do not disclose the implementation details or a reference for their embedding operator. They neither mention additional storage requirements for storing embeddings explicitly, which might indicate that they use an algorithm that does not require such knowledge. The authors evaluate FSG on chemical and artificial graph datasets. They do not describe the particular generation of the artificial graphs. Their graph database generator, however, is used by several other authors to evaluate their approaches (e.g. Yan and Han, 2002; Zhao and Yu, 2008). There are graph databases where the performance of FSG drastically decreases (compare Chapter 4). Notably, the algorithm was used by Deshpande et al (2005) to first show the impressive predictive performance of frequent subgraph based learners on chemical graph datasets.

Borgelt et al propose MoSS, a frequent subgraph miner specifically suited for chemical graph databases (Borgelt and Berthold, 2002; Borgelt et al, 2005). Their algorithm implements special domain knowledge (e.g., handling of aromatic bonds) and is a depth-first search over a pattern space that can be "seeded" with a chemically meaningful core pattern that will be contained in all frequent patterns to be found. The authors use embedding lists to compute the support count; their approach, however, suffers from a missing graph canonicalization scheme. Hence patterns are enumerated multiple times (and their support is computed multiple times). Without giving the details, the authors claim that multiple output of equivalent patterns can be suppressed (which would require deciding the isomorphism problem for pairs of patterns). The authors show experiments in which they qualitatively analyze the patterns found using their approach on the NCI-HIV dataset.

gSpan by Yan and Han (2002) mines frequent subgraphs using a depth-first traversal of the pattern space. To avoid multiple enumeration of the same candidate pattern (up to isomorphism), it applies an inclusion-exclusion principle on frequent edges. That is, a pattern is extended with an ever shrinking set of frequent edges. To compute the support of a candidate pattern, the algorithm recursively works on the support sets of the patterns being extended, resulting in a reduced number of calls to the embedding operator. Though the authors do not cite or mention it in the paper, the acknowledgments suggest that gSpan uses the subgraph isomorphism algorithm by Cordella et al (1999). They show experiments on the datasets used by Kuramochi and Karypis (2004) and show that their algorithm outperforms FSG. In (Yan and Han, 2003) the authors extend their algorithm to mine closed frequent subgraphs.

Huan et al (2003) propose FFSM, an algorithm that also mines frequent subgraphs using a depth-first traversal of the pattern space. They use a novel canonical representation of arbitrary graphs that has size $O\left(n^2\right)$ for a graph on n vertices and propose extension and join operators that generate all frequent patterns. However, these operators may generate patterns multiple times, not necessarily in canonical form. Without giving details, the authors claim to be able to decide whether a representation is canonical, and hence that the algorithm is correct (i.e., each pattern is printed exactly once up to isomorphism). They use embedding lists to store all possible embeddings of the frequent patterns in canonical form and show how their extension and join operators can use the embedding lists to only output frequent patterns. The authors later extend their work to maximal frequent subtrees, resulting in the SPIN algorithm (Huan et al, 2004).

Gaston (Nijssen and Kok, 2004, 2005) is the fastest frequent subgraph mining system on chemical graph databases (Wörlein et al, 2005). Their algorithm mines frequent patterns in three stages: First, all frequent paths are generated. In the second stage, tree candidates are grown from the frequent paths. Finally frequent cyclic graphs are grown from the frequent trees and frequent paths by adding edges between existing vertices. Hence Gaston can be seen as both a specialized frequent subtree mining algorithm and as a frequent subgraph mining algorithm: Without overhead, the generation of cyclic graphs can be avoided by stopping after the tree generation step. Candidate generation is based on an efficient canonical representation of graphs that is based on depth-first sequences; only extensions of patterns that are in canonical form are further expanded. This property can be checked in constant time for trees and paths, yielding a very fast enumeration of candidate patterns; for cyclic graphs, however, this property is more difficult to check. Gaston traverses the pattern space in a nonstandard postorder: The support of all extensions of a frequent pattern in canonical form is evaluated before calling the search function recursively for the first (frequent) extension. In this way, the number of allowed extension operations can be restricted efficiently, yielding a smaller number of candidate extensions in subsequent steps. There are two variants of Gaston that differ in their support counting subroutine. The first variant uses embedding lists, the second computes the subgraph isomorphisms "from scratch" for each candidate pattern. The authors are not very specific on the details of the latter. They describe it as a backtracking algorithm that has exponential worst-case running time in the size of the pattern and the transaction graphs involved. They evaluate their algorithm on an artificial tree dataset and on three large molecular datasets.

Horváth and Ramon (2010) propose an algorithm that mines all frequent connected subgraphs in transaction databases consisting of graphs of bounded tree-width. Impressively, their algorithm runs in incremental polynomial time, while the embedding operator by itself is **NP**-complete (compare Section 3.1). That is, the SUBGRAPHISOMORPHISM problem is **NP**-complete for transaction graphs with tree-width at most some constant k if the vertex degree of the pattern is not bounded by a constant, as well. This result is up to our knowledge the only existing result that describes an efficient algorithm for a problem in the upper left quadrant of Figure 2.2: The HAMILTONIANPATH problem can be solved in polynomial time due to the result of Matoušek and Thomas (1992), as paths have vertex degree at most two. Their algorithm identifies a polynomially sized subset of *non-redundant iso-quadruples* that are stored for each frequent subgraph and each transaction. Such iso-quadruples represent partial subgraph isomorphisms but – in comparison to explicitly storing all possible embeddings from the patterns to the transaction graphs – may represent multiple embeddings of the pattern that are in some sense equivalent. Their embedding operator extends ideas from (Hajiaghayi and Nishimura, 2007) to the case that the vertex degree of the pattern is unbounded. Interestingly, the approach of Horváth and Ramon requires a breadth-first traversal of the pattern space to result in an efficient algorithm. They show that almost all (>99.9%) of the graphs in a large chemical graph database have tree-width at most 3, and hence that their result is practically relevant but don't give any empirical evaluation of their algorithm. Horváth et al (2013) extend these techniques to mine all frequent *induced* subgraphs in transaction databases consisting of bounded tree-width graphs with unbounded vertex degree in incremental polynomial time.

3.2.3. Algorithms for Relaxed Problems

As we are interested in a relaxation of the FTM problem in this thesis, our work is related to other relaxations that were proposed for the transactional setting. There has also been some interest in dealing with graph databases that contain noisy data. While this setting is different from ours, some of the resulting algorithms can be applied to exact transactional graph databases and yield approximations of the set of frequent subgraphs.

Chen et al (2009) try to address the drawbacks of the practical frequent subgraph mining systems described in Section 3.2.2 on larger graph transactions, i.e., the large number of embeddings of a pattern that need to be explicitly considered by the embedding operators described in Section 3.1.1. To this end, they propose to replace each graph in the database by a *summarized graph* and to mine frequent patterns in this novel graph database using the gSpan algorithm (Yan and Han, 2002). A summarized graph G' is created from a labeled transaction graph G by choosing a random partition $V(G) = V_1 \dot\cup V_2 \dot\cup \ldots \dot\cup V_k$ of the vertex set of G such that for all $i \in [k]$, all vertices in V_i have the same label. Now, the vertices of G' are the partitions V_i and there exists an edge (V_i, V_j) with label l if and only if there exists an edge $(v_i, v_j) \in E(G)$ with $v_i \in V_i$, $v_j \in V_j$, and label l. Hence, the summarized graph G' is not simple, i.e., it may contain self-loops and multiple edges (with different labels) between any two vertices. This construction results in a two-sided error, i.e. for two graphs H and G and a summarized graph G' of G there may be

false negatives: $H \preccurlyeq G$ but $H \not\preccurlyeq G'$, or

false positives: $H \not\preccurlyeq G$ but $H \preccurlyeq G'$.

These effects obviously translate to the set of frequent patterns found by the algorithm of Chen et al. To deal with false negatives, the authors propose to lower the frequency threshold in the mining phase and give a probabilistic guarantee on its effectiveness. To further increase the recall of frequent patterns their algorithm repeats the summarization independently several times. To address the false positives, they propose to simply retest the patterns found to be frequent in the summarized graph database on the original graph database. Together, this yields the Summarize-Mine algorithm that guarantees to find a subset of all frequent subgraphs in a given database. As this is very close to our problem formulation presented in Chapter 4, we explicitly stress the differences: (i) Chen et al (2009) find frequent subgraphs instead of frequent subtrees and use an exponential worst-case time embedding algorithm, thus they are not able to guarantee any delay bounds. (ii) Their algorithm requires to retest all patterns found in the summarized graph database on the original database to ensure that they are indeed frequent. Hence, (iii) no real structural simplification of the FCSM or SubgraphIsomorphism problems takes place. The methods proposed in this thesis, on the other hand, (i) guarantee polynomial delay, by using an efficient embedding operator, (ii) do not require to re-evaluate patterns on the original database to guarantee that each output pattern is indeed a frequent tree, and (iii) accomplish this by transforming an infeasible FCSM problem to a FTM problem which can be solved efficiently (i.e., with polynomial delay).

Zou et al (2010) propose MUSE to mine patterns in databases of *uncertain graphs*. An uncertain graph is a labeled graph G together with a probability function $p : E(G) \to [0,1]$ on its edges and represents the probability distribution P over all graphs $(V(G), E')$ for $E' \subseteq E(G)$, with $P((V(G), E')) := \prod_{e \in E'} p(e)$. Now, for a pattern graph H and an uncertain graph G, the probability of H matching G is defined as

$$P_{\preccurlyeq}(H,G) = \sum_{E' \subseteq E(G)} P((V(G),E'))I(H,(V(G),E'))\,,$$

where $I(H, (V(G), E')) = 1$ if $H \preccurlyeq (V(G), E')$, otherwise $I(H, (V(G), E')) = 0$. Given a graph database $\mathcal{D}$ and a frequency threshold $\theta \in (0, 1]$, MUSE approximates the set of all pattern graphs H with $\frac{1}{|\mathcal{D}|} \sum_{G \in \mathcal{D}} P_{\preccurlyeq}(H, G) \geq \theta$, i.e., where the average probability of H matching the graphs is at least θ. They show that counting the number of such patterns for a given database is #P-complete (Valiant, 1979) and their algorithm approximates the set of such patterns. MUSE works by generating patterns on the database with probabilities removed from the edges by explicitly storing and extending the embeddings as described in Section 3.1.1. Based on these explicit embeddings they propose an exponential time algorithm to compute the matching probabilities and an approximate algorithm that computes an interval of matching probabilities. They show how to obtain an algorithm that guarantees with high probability for some $\epsilon \in (0, 1]$ and a frequency threshold θ that all patterns with support at least θ are output, all patterns with support less than $(1 - \epsilon)\theta$ will not be output, and decisions for remaining patterns are arbitrary. However, due to the necessity of evaluating a function over all embeddings of a pattern, the method does not run in output polynomial time. In a way, our work in this thesis can be seen as the opposite approach: We consider some probability distributions on the set of spanning trees given by the database graphs and obtain frequent subtrees from certain samples directly, instead of mining patterns on the underlying graphs. Our goal, however, is to approximate the set of exact frequent subtrees in the original database, instead of the set of patterns whose average probability is above some threshold.

Li and Wang (2015) are interested in transactional graph databases that contain graphs where some vertices, edges, or labels may be "wrong". They propose to relax the notion of isomorphism and subgraph isomorphism. To this end they introduce β (subgraph) isomorphism, where a graph H is β subgraph isomorphic to a graph G if there exists a sequence of vertex and edge additions or deletions and relabeling operations of length at most β that transforms H into a graph that is (subgraph) isomorphic to G. If applied in Algorithm 2.1 such an embedding operator would result in finding a *superset* of the frequent patterns with respect to subgraph isomorphism: β subgraph isomorphism is equivalent to subgraph isomorphism for $\beta = 0$ and for any $\beta > 0$ the existence of a subgraph isomorphism from H to G implies the existence of a β subgraph isomorphism from H to G. Their proposed tool REAFUM, however, first selects a small subset of "representative" graphs for a given graph database and considers only those patterns as candidates that can be found in the set of representative graphs. Frequency counting takes place on the full dataset and is based on storing all embeddings of all approximate matches of the patterns (i.e., an extension of the ideas described in Section 3.1.1). In their experimental evaluation they show that they are able to find more patterns than the exact Gaston algorithm on a small molecular dataset. Due to the candidate selection process, the resulting pattern set is not guaranteed to be a superset of all frequent patterns with respect to normal subgraph isomorphism.

Recently, Schulz et al (2018) proposed a frequent tree mining algorithm that employs *partially injective* embedding operators between graph homomorphism and subgraph isomorphism. Subgraph isomorphisms are injective graph homomorphisms; Schulz et al propose to add *some* injectivity constraints to graph homomorphisms, while maintaining computational efficiency of the embedding operators. In particular, graph homomorphism can be decided in polynomial time for patterns of bounded tree-width and *arbitrary transaction graphs* (Dalmau et al, 2002). Hence, for tree patterns, which have tree-width one, one can add a number of binary injectivity constraints between vertices (implemented by new edges with a new label) to the pattern as long as the resulting graph remains of bounded tree-width. Deciding homomorphism between such an extended pattern and a transaction graph that is extended with all possible edges with that new label ensures that the injectivity constraints are fulfilled. As a result, tree patterns that are frequent with respect to subgraph isomorphism are frequent with respect to partially injective homomorphism, as well. Schulz et al propose a mining strategy for this embedding operator that finds a superset of all frequent tree patterns (with respect to subgraph isomorphism). This is done by mining "maximally" constrained tree patterns that are defined by *k-trees*. A k-tree is a maximal graph that has tree-width k, that is, the addition of a novel edge between two existing vertices results in a graph with tree-width $k + 1$. k-trees have an algorithmic definition that allows to efficiently enumerate these patterns, using Algorithm 2.1. The output of the algorithm then is a set of k-trees consisting of a tree "core" and some binary injectivity constraints. Omitting the injectivity constraints, we obtain a set of trees, some of which may be isomorphic, that contains the set of frequent subtrees (with respect to subgraph isomorphism). Hence this method can be seen as approximating the set of frequent subtrees "from the other side" than the algorithms proposed in this thesis.

4. Probabilistic Frequent Subtrees

We will now investigate how to efficiently obtain frequent subtrees from databases of arbitrary graphs. The main result for this chapter is a *polynomial delay* frequent tree mining algorithm for *arbitrary* transaction graphs that is *sound*, but *incomplete*. We empirically demonstrate that (i) the predictive performance of the incomplete output of our mining algorithm is competitive to that of *all* frequent subtrees and that (ii) our algorithm is capable to mine frequent trees efficiently in a broad range of such graph databases where even the most popular frequent subgraph mining algorithms are unable to produce any result in reasonable time.

Our preliminary experiments in Chapter 1 showed that most existing frequent subgraph or frequent subtree mining systems are practically restricted to very simple graph databases; usually chemical graphs. On these datasets, however, frequent subtrees are powerful features, i.e., the predictive performance of learning algorithms based on frequent subtree features is high (e.g. Deshpande et al, 2005). We therefore expect that frequent tree patterns might be good predictors in other settings as well. Furthermore, there is a general interest in tree-based features for graphs beyond chemical molecules (e.g. Chi et al, 2004a; Kibriya and Ramon, 2013). However, to empirically evaluate this assumption, e.g. for the social graphs considered in the preliminary experiments, novel techniques are required to generate such patterns in practically feasible time and space constraints.

So how can we find frequent subtrees in (more) general graph databases? Recall from Chapter 2 that we cannot expect an output polynomial time algorithm for the FTM problem unless **P** = **NP** (see Theorem 2.2). Hence we cannot expect to find an efficient exact algorithm for the FTM problem without either (i) restricting the problem to some feasible graph class or (ii) giving up the correctness of our algorithm. The state-of-the-art frequent subtree mining systems follow approach (i) by either using specialized efficient matching operators for restricted transaction classes (cf. Section 3.1) or by using heuristics that practically restrict them to certain graph classes (cf. Section 3.1.1). As a result there exists no mining system that is applicable to arbitrary graph databases in practice. Thus there is a need to build a system that can find frequent subtrees in the graph databases not yet covered by any algorithm. We hence follow approach (ii) to close this gap and abandon the requirement of the correctness of the mining algorithm to keep it applicable to arbitrary databases of small to medium sized graphs. In particular, our algorithm will find a subset of frequent subtrees that can be computed efficiently on all graph databases.

We present a frequent subtree mining algorithm with one-sided error that is *not* restricted to any particular graph class. That is, it always terminates in time *polynomial* in the size of the database and the number of frequent patterns generated. The algorithm calculates a subset of the frequent subtrees of a given graph database as follows: (i) We represent each input graph by a forest formed by vertex disjoint copies of k random spanning trees for some *small* k and (ii) compute the set of subtrees frequent in the forest database generated in step (i). See Figure 4.1 for an example. Combining this representation, the fact that it can be generated in polynomial time, and the positive result that frequent subtree mining in forests can be solved with polynomial delay (cf. Section 3.1), we arrive at an algorithm computing a subset of the frequent tree patterns in time *polynomial* in the combined size of the input database and the set of generated tree patterns. In particular, it will output the patterns with polynomial delay. We call our method probabilistic subtree mining and the resulting pattern set *probabilistic frequent subtrees*.

Our approach is sound, but incomplete: Each generated probabilistic tree pattern is guaranteed to be a frequent subtree of the database. Some frequent subtrees, however, may be missed by the algorithm, as they are not necessarily frequent with respect to the random forest database generated in step (i). Hence the set of probabilistic frequent subtrees is always a subset of the set of frequent subtrees for a fixed database and frequency threshold. Our somewhat unusual idea is motivated by the fact that any tree found by our mining algorithm is not only frequent with respect to the database, but with high probability it has a relatively high frequency also in the set of spanning trees for each transaction graph containing it. Thus, there must be a high chance that such a tree pattern will be detected with this method in a query graph as well, if it is part of it.[1]

We empirically evaluate the proposed method on the real-world and artificial datasets described in Section 2.4. In particular, we investigate the recall of the probabilistic frequent subtrees with respect to all frequent subtrees for various numbers of random spanning trees per graph.[2] Our technique is faster by at least two orders of magnitude on Erdős-Rényi random graphs of low density. On social graphs and Erdős-Rényi random graphs of moderate density, probabilistic frequent subtrees can be found quickly, while the exact frequent subgraph (subtree) mining algorithms fail. On chemical graphs, we observed only a marginal loss in the predictive performance of our probabilistic subtrees with respect to the exact frequent subtree. We show that with increasing size of the dataset we needed decreasing numbers of sampled spanning trees per graph to obtain a close approximation of the predictive performance of frequent subgraphs. In particular, for the NCI-HIV dataset consisting of more than 40 000 molecular graphs, 5 sampled spanning trees per graph resulted in almost identical predictive performance.

Outline

The rest of this chapter is organized as follows: Section 4.1 gives a detailed description of our algorithm and formally defines the relaxation of the FTM problem considered in this thesis (4.1.1). It also discusses a reason for the practical success of our idea (4.1.2) and discusses implementation issues (4.1.3). Section 4.2 shows our experimental results

[1] We assume that the query graph has been selected from the same (unknown) probability distribution as the graphs in the input database.

[2] Notice that precision is always 100% due to the soundness of the algorithm.

regarding the runtime (4.2.1), recall (4.2.2), stability (4.2.3), and predictive performance (4.2.4) of our probabilistic frequent subtrees. Finally, Section 4.3 concludes with results and open questions that will be discussed in the following chapters.

4.1. Mining Probabilistic Frequent Subtrees

We now formally define a relaxation of the FTM problem and present our first algorithm to tackle this problem. The main result of this chapter is similar to Theorem 2.1 for the relaxed FTM problem. To arrive at its definition, recall first that the task of finding the set of *all* frequent subtrees of a given database $\mathcal{D}$ raises the following two related computational problems (cf. Sections 2.1 and 2.2):

(P1) The FTM Problem: *Given* a finite set $\mathcal{D}$ of graphs and a frequency threshold $t \in [|\mathcal{D}|]$, *generate* the set $\mathcal{F}$ of frequent *trees*, i.e., *all* trees H with $|\{G \in \mathcal{D} : H \preccurlyeq G\}| \geq t$.

(P2) The SubtreeIsomorphism Problem: *Given* a tree H and a graph G, *decide* whether or not $H \preccurlyeq G$.

The second problem appears in the support counting step of all algorithms solving (P1) with the generate-and-test paradigm. In particular, Algorithm 2.1 calls another algorithm for (P2) as a subroutine. 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 the demand on the correctness of the subtree isomorphism test for (P2). As we will show, this results in practically effective algorithms. The goal of this thesis is to obtain a system that is applicable to arbitrary graph databases. In particular, we want to give some output in reasonable time on any kind of small to medium sized graph data. We approach this problem by requiring polynomial delay during the generation of frequent trees and are willing to trade in the correctness of the algorithm. That is, we will require each pattern listed by our algorithm to be a frequent tree in the database, but will allow to miss some of the frequent trees.

4.1.1. The Relaxed Frequent Subtree Mining Problem

Regarding the relaxation of (P1), we consider for each graph $G \in \mathcal{D}$ a forest $\mathfrak{S}_k(G)$ formed by the *vertex disjoint* union of k *random* spanning trees of G. We then solve (P1) for this random forest database. More precisely, for a connected[3] graph G we sample k spanning trees and $\mathfrak{S}_k(G)$ consists of k connected components. Each component is isomorphic to at least[4] one of the k sampled spanning trees of G. With this problem relaxation we arrive at a frequent subgraph mining algorithm that is easy to implement and practically

[3] For ease of exposition, we restrict our description to connected graphs. The techniques described in this thesis can be extended to disconnected graphs by applying them for each connected component separately.

[4] It might occur that some of the sampled spanning trees are isomorphic. We address this in Section 4.1.3.

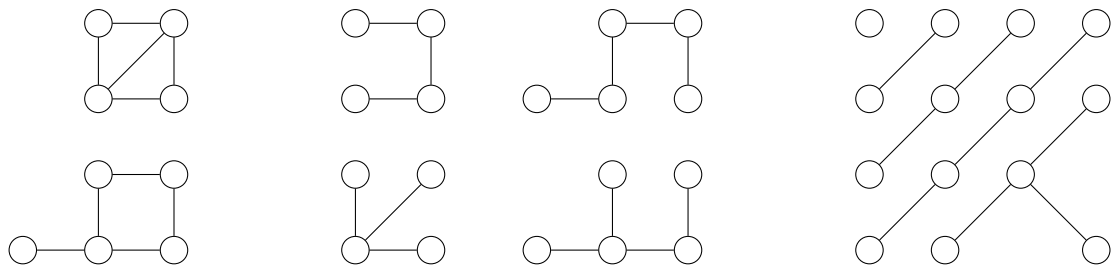


Figure 4.1.: A database $\mathcal{D}$ consisting of two graphs G_1, G_2 (left), the forests $\mathfrak{S}_2(G_1)$ and $\mathfrak{S}_2(G_2)$ of two sampled spanning trees of G_1 and G_2 (middle), and the set of the five 2-frequent tree patterns found in the forest database $\mathcal{D}' = \{\mathfrak{S}_2(G_1), \mathfrak{S}_2(G_2)\}$ (right). Note that all 2-frequent subtrees are found. Using only one spanning tree for each graph in this example, however, would always result in an incomplete output.

effective, as shown in Section 4.2. We call the resulting tree patterns *probabilistic frequent subtrees* of $\mathcal{D}$ to distinguish them from the set of all frequent trees. Figure 4.1 shows an example of the idea on a database consisting of two graphs.

Our approach effectively relaxes the problem of mining frequent subtrees in *arbitrary* graphs to that of mining trees in a forest database. In contrast to the computational intractability of (P1), this relaxed problem can be solved with polynomial delay if k is bounded by a polynomial in the input size. This can be done using Algorithm 2.1 with a suitable embedding operator (compare Section 3.1). This means in practice that our algorithm guarantees to find a certain amount of patterns in an acceptable time if k is chosen appropriately.

Algorithm 4.1 shows the high level pseudo-code of this approach. In addition to $\mathcal{D}$ and t in problem (P1), the input contains an additional parameter $k \in \mathbb{N}$ as well. k specifies an upper bound on the number of spanning trees to be generated for the transaction graphs. It is easy to see that for any $\mathcal{D}$, t, and k, Algorithm 4.1 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 some frequent patterns.

Another advantage of our technique is that it assumes neither explicitly nor implicitly any structural restriction on the input graphs. Random spanning trees can be drawn efficiently from any graph (Wilson, 1996); after this step, we are in a world that consists of forests and trees.

Algorithm 4.1 PROBABILISTIC FREQUENT SUBTREE MINING

```
input: graph database $\mathcal{D} \subseteq \mathcal{G}$, frequency threshold $t > 0$ integer, and $k > 0$ integer
output: a random subset of the set of frequent subtrees of $\mathcal{D}$
1: $\mathcal{D}' := \varnothing$
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 the vertex disjoint union of those trees to $\mathcal{D}'$
5: list all subtrees that are $t$-frequent in $\mathcal{D}'$
```

The above mentioned properties of Algorithm 4.1 can be generalized. We will show that the mining technique described above can be seen as the generic levelwise algorithm with a special embedding operator that is used in the SUPPORTCOUNT subroutine (compare Algorithm 2.1). This embedding operator solves a relaxed SUBTREEISOMORPHISM problem. We will show that efficiency and completeness results analogous to Theorem 2.1 hold in this case, as well.

Let $\mathcal{P}$ be the class of trees and $\mathcal{G}$ a graph class. Then a function

$$f : \mathcal{P} \times \mathcal{G} \to \{0,1\}$$

is a *relaxed subtree isomorphism decision function*, if for all $H \in \mathcal{P}$ and $G \in \mathcal{G}$ it fulfills

one-sided error $f(H,G) = 1 \quad \Rightarrow \quad H \preccurlyeq G$

monotonicity $f(H,G) = 1 \quad \Rightarrow \quad f(H',G) = 1$ for all $H' \preccurlyeq H$

Given such a function f, the SUBTREEISOMORPHISM $_{Relaxed}$ (f) problem is to decide for a given tree H and a graph G whether $f(H,G) = 1$. The relaxed frequent subtree mining problem with respect to relaxed subtree isomorphism decision function f is defined as follows:

RELAXED FREQUENT SUBTREE MINING (FTM $_{Relaxed}$ (f)) PROBLEM: *Given* a finite set $\mathcal{D} \subseteq \mathcal{G}$ for some graph class $\mathcal{G}$, and an integer threshold $t > 0$, *list* the set of all trees H with $f(H,G) = 1$ for at least t graphs G in $\mathcal{D}$.

The FTM and FTM $_{Relaxed}$ (f) problems are connected in the following way:

Lemma 4.1. *Given a finite set $\mathcal{D} \subseteq \mathcal{G}$ for some graph class $\mathcal{G}$, and an integer threshold $t > 0$, and a relaxed subtree isomorphism decision function f. Then the output of the* FTM $_{Relaxed}$ (f) *problem is always a subset of the output of the* FTM *problem.*

Proof. We have to show that each element in the output of the FTM $_{Relaxed}$ (f) problem is frequent with respect to subgraph isomorphism in $\mathcal{D}$. Let H be a tree in the output of FTM $_{Relaxed}$ (f). Then the support count of H, i.e., the number of graphs $G \in \mathcal{D}$ such that $f(H,G) = 1$ is at least t. Hence, by assumption about the one sided error, the support with respect to subgraph isomorphism is at least t. □

Theorem 4.2 below shows how to efficiently compute a solution for a FTM $_{Relaxed}$ (f) problem. It practically reduces finding a solution for the FTM $_{Relaxed}$ (f) problem to finding an efficient embedding algorithm $\mathfrak{A}(f)$ for the SubtreeIsomorphism $_{Relaxed}$ (f) problem.[5]

Theorem 4.2. *Let $\mathfrak{A}(f)$ be an algorithm that solves the* SubtreeIsomorphism $_{Relaxed}$ (f) *problem in polynomial time. Then Algorithm 2.1 using $\mathfrak{A}(f)$ in the* SupportCount *subroutine solves the* FTM $_{Relaxed}$ (f) *problem with polynomial delay.*

Proof Sketch. The proof of Theorem 4.2 is analogous to the proof of Theorem 2.1. Note that this proof only requires the pattern matching operator to be efficiently computable and to be monotone. □

Our probabilistic frequent subtree approach proposed in the beginning of this section is in fact the solution to a FTM $_{Relaxed}$ problem: Let $\mathcal{D}$ be a database of arbitrary graphs and $\mathcal{D}'$ be a corresponding database of sampled spanning trees $\mathfrak{S}_k(G)$ for all $G \in \mathcal{D}$. Then

$$f(H,G) := \begin{cases} 1 & \text{if } H \preccurlyeq \mathfrak{S}_k(G) \\ 0 & \text{otherwise} \end{cases}$$

for all trees H is a relaxed subtree isomorphism decision function for the (finite) transaction graph class $\mathcal{G} = \mathcal{D}$. As a result, each choice of a random database $\mathcal{D}'$ for $\mathcal{D}$ results in a SubtreeIsomorphism $_{Relaxed}$ (f) problem and corresponding FTM $_{Relaxed}$ (f) problem. Both corresponding problems can be solved efficiently, as this particular SubtreeIsomorphism $_{Relaxed}$ (f) can be decided in polynomial time: Spanning trees can be sampled in polynomial time for arbitrary graphs (see Section 4.1.3) and subgraph isomorphism between trees and forests can be decided in polynomial time (see Section 3.1). Note also that the function f can easily be extended to a novel graph G by sampling a forest $\mathfrak{S}_k(G)$ for G. To simplify our notation, we omit the decision function f from now on, when it is clear from the context. The relation $H \preccurlyeq \mathfrak{S}_k(G)$ will be referred to as H *probabilistically matches* G.

As a result of these considerations, probabilistic frequent subtrees are a way to define a class of FTM $_{Relaxed}$ and SubtreeIsomorphism $_{Relaxed}$ problems. Lemma 4.1 implies that probabilistic frequent subtrees are always a subset of the full set of frequent subtrees. Theorem 4.2 implies that the full set of probabilistic frequent subtrees for a particular choice of sampled spanning trees can be mined with polynomial delay. Note that Algorithm 4.1 implements exactly this approach. We will use the general framework provided by Lemma 4.1 and Theorem 4.2 in Chapter 5 and the algorithm for the SubtreeIsomorphism $_{Relaxed}$ problem in Chapter 6.

Note that the relaxation of (P2) could be implemented in two different ways: (i) Each time we evaluate a probabilistic match of some pattern H, we sample $\mathfrak{S}_k(G)$ anew for all $G \in \mathcal{D}$ or (ii) $\mathfrak{S}_k(G)$ is sampled once and reused for multiple invocations of the algorithm in the support count step. Our theoretical results above require variant (ii) of the

[5] Lemma 4.1 and Theorem 4.2 can also be formulated for a relaxed version of the FCSM problem. Note further, that formulating the problems with opposite one-sided error (i.e., a "No" is a "No", but a "Yes" might be a "No", as well) results in an algorithm that mines a superset of all frequent patterns. These issues might be of interest for future work; we don't investigate them in this thesis.

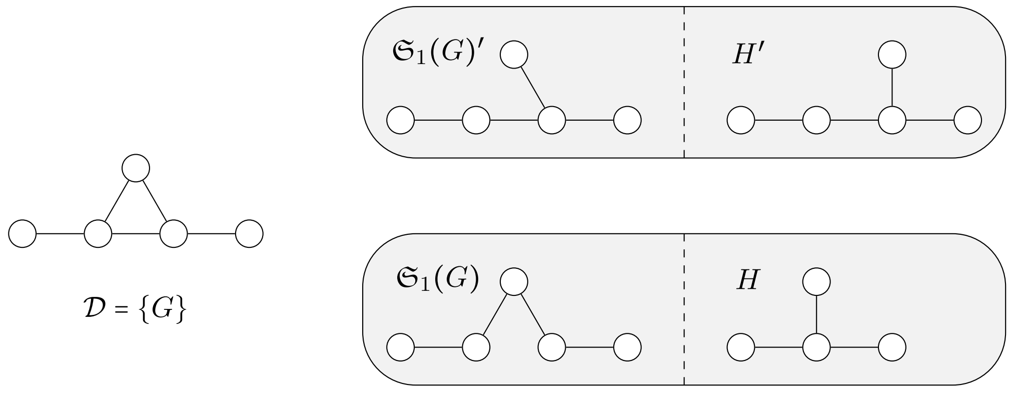


Figure 4.2.: A database $\mathcal{D}$ consisting of a single graph G (left) and two evaluations of the probabilistic matching operator with independently sampled sets of spanning trees $\mathfrak{S}_1(G), \mathfrak{S}_1(G)'$ for two pattern trees H, H' (right). In this case, H' is a probabilistic match to G, while H is not, although $H \preccurlyeq H'$.

probabilistic match computation. In this way, we reduced the FTM problem on arbitrary graphs to that of frequent tree mining in a *fixed* forest database. If we used variant (i) of the embedding operator in the generic mining algorithm, we would lose the monotonicity of the embedding operator: As a result of the resampling of spanning trees it is not guaranteed that all subgraphs of a pattern were identified as probabilistic matches. Figure 4.2 illustrates this situation. Due to this situation and due to the fact that sampling spanning trees induces a nontrivial cost (see Section 4.1.3), we stick with the variant of the embedding operator that receives a tree H and a forest $\mathfrak{S}_k(G)$ as input, which was computed in a preprocessing step.

The incompleteness of our proposed probabilistic frequent subtree pattern sets with respect to the set of frequent subtrees raises two important questions:

1. How *stable* is the output of Algorithm 4.1 and what is its *recall* with respect to *all* frequent subtrees? (Note that precision is always one 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 4.1.2 that certain *important* tree patterns are very likely to be among the probabilistic frequent subtrees even for small values of k. To complement this analysis, we show in Section 4.2 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.

Regarding the second question above, we experimentally show in Section 4.2 on different real-world benchmark graph datasets that the predictive performance of our probabilistic approach is comparable to the predictive performance of the full set of frequent

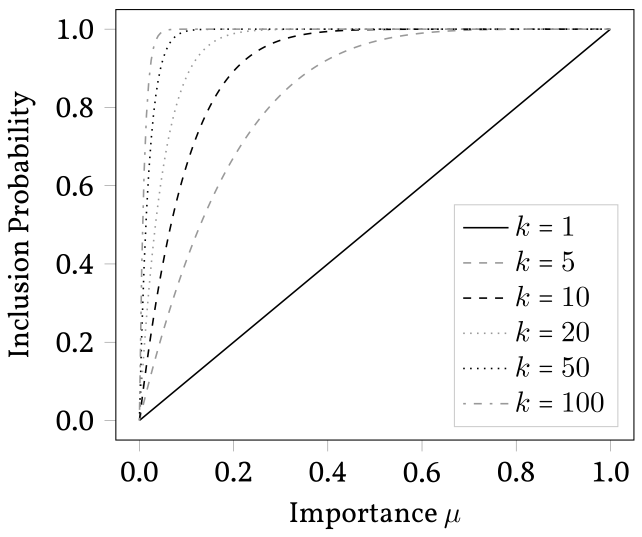


Figure 4.3.: The function $1 - (1 - \mu)^k$ for different values of k.

subtrees. In fact, this holds not only for the set of all frequent trees, but also for the full set of frequent *subgraphs*. Before presenting the empirical results in Section 4.2, we first analyze the recall of our approach theoretically in Section 4.1.2 and discuss some implementation issues and the time complexity of Algorithm 4.1 in Section 4.1.3.

4.1.2. Probabilistic Bounds and the Importance of Subtrees

The rationale behind our probabilistic technique is as follows. For a connected graph G, let $\mathfrak{S}(G)$ be the forest of *all* spanning trees of G. That is, $\mathfrak{S}(G)$ is the graph formed by the vertex disjoint union of all spanning trees of G. For the remainder of this section, we will regard $\mathfrak{S}_k(G)$ (resp. $\mathfrak{S}(G)$) as a *set* of k (rep. all) spanning trees of G. Note that this is equivalent to considering it as a forest in the following sense: There exists a spanning tree $S \in \mathfrak{S}_k(G)$ (resp. $\mathfrak{S}(G)$) (as a set) such that $H \preccurlyeq S$ if and only if $H \preccurlyeq \mathfrak{S}_k(G)$ (resp. $\mathfrak{S}(G)$) (as a forest). Using these notions, a tree T is ***μ-important*** in G if

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

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

$$\mathbf{P}\left[\exists S \in \mathfrak{S}_k(G) \ : \ T \preccurlyeq S\right] \geq 1 - (1 - \mu)^k \ . \tag{4.1}$$

which follows directly from

$$\mathbf{P}\left[\forall S \in \mathfrak{S}_k(G) \,:\, T \not\preccurlyeq S\right] \leq (1-\mu)^k \ .$$

The bound in (4.1) implies that for any graph G and μ-important tree pattern T in G for some $\mu \in (0,1]$, and for $\delta \in (0,1)$,

$$\mathbf{P}\left[\exists S \in \mathfrak{S}_k(G) \,:\, T \preccurlyeq S\right] \geq 1-\delta \tag{4.2}$$

holds whenever

$$k \geq \frac{1}{\mu} \ln \frac{1}{\delta} \ . \tag{4.3}$$

Here (4.3) is obtained from (4.1) and (4.2) by the inequality $1 - x \leq e^{-x}$. (See, also, Figure 4.3 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 4.3. *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 $k \geq \frac{1}{\mu} \ln \frac{1}{\delta}$,*

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

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.

Mining μ-Important Patterns

Now let $\mathcal{D}$ be a graph database, $\mu > 0$ some importance value, and $t \in \mathbb{N}$ a frequency threshold for a FTM $_{Relaxed}$ problem. Let H be a tree that is μ-important in at least $t+x$ graphs in $\mathcal{D}$ (this implies that H is a frequent tree with respect to the exact FTM problem with threshold t). Using Proposition 4.3 and a simple application of a standard combinatorics result (also used by Chen et al, 2009, in a similar context), we have

Theorem 4.4. *Let $\mathcal{D}^k$ be a forest database obtained from $\mathcal{D}$ by independently sampling k spanning trees for each $G \in \mathcal{D}$ uniformly at random. Let H be a tree that is μ-important in at least $t+x$ graphs in $\mathcal{D}$ and let s_H^k be the support of H in $\mathcal{D}^k$, i.e., the number of forests $G' \in \mathcal{D}^k$ with $H \preccurlyeq G'$. Then*

$$\boldsymbol{P}\left[s_H^k < t\right] \leq \sum_{i=0}^{t-1} \binom{t+x}{i} (1-(1-\mu)^k)^i (1-\mu)^{k(t+x-i)} \, .$$

Proof. Let $\mathcal{D}(H) := \{G \in \mathcal{D} \,:\, H \text{ is } \mu\text{-important in } G\}$, let $\mathcal{D}^k(H) := \left\{G' \in \mathcal{D}^k \,:\, G \in \mathcal{D}(H)\right\}$, and let σ_H^k be the support of H in $\mathcal{D}^k(H)$. The probability that H is t-frequent in $\mathcal{D}^k$ is larger or equal to the probability that H is t-frequent in $\mathcal{D}^k(H)$. Adding additional forests to $\mathcal{D}^k(H)$ that are drawn from graphs in $\mathcal{D}$, where H is subgraph isomorphic to a $\mu' < \mu$-fraction of the spanning trees can only increase the probability of H being t-frequent in the larger database (recall that t is an ***absolute*** threshold). Hence, we have $\mathbf{P}\left[s_H^k < t\right] \leq \mathbf{P}\left[\sigma_H^k < t\right]$. Furthermore, the probability of H being t-frequent in $\mathcal{D}^k(H)$ can only decrease if we (pessimistically) assume that the probability of H being subgraph of any spanning tree of any $G \in \mathcal{D}(H)$ is exactly μ. Therefore we can bound $\mathbf{P}\left[\sigma_H^k < t\right]$ by the cumulative density function of a binomial distribution evaluated at t with parameters $n = \left|\mathcal{D}^k(H)\right|$ and $p = 1 - (1-\mu)^k$. See Lemma B.1 in the appendix. Hence,

$$\mathbf{P}\left[s_H^k < t\right] \leq \mathbf{P}\left[\sigma_H^k < t\right] \leq \sum_{i=0}^{t-1} \binom{\left|\mathcal{D}^k(H)\right|}{i} (1-(1-\mu)^k)^i (1-\mu)^{k(\left|\mathcal{D}^k(H)\right|-i)}$$

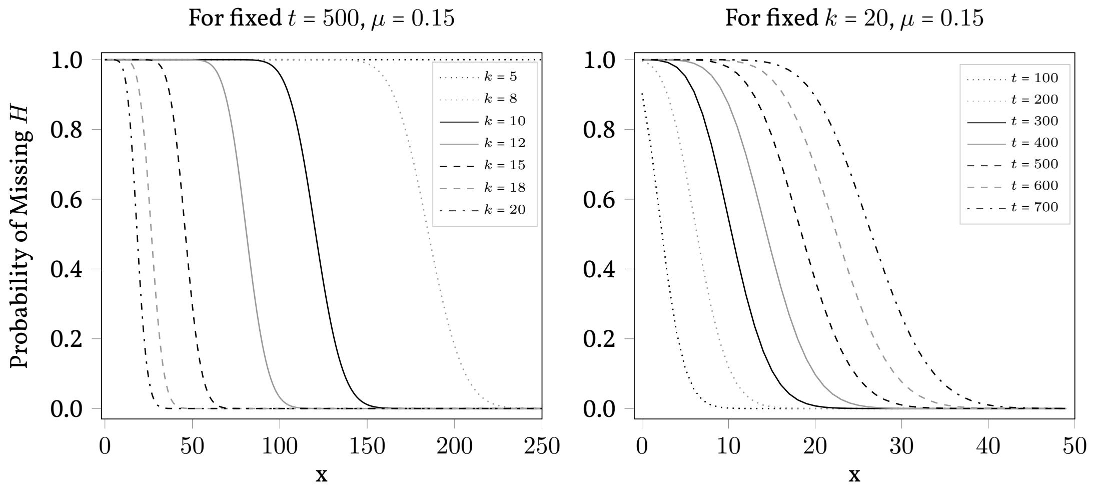
and setting $t + x := |\mathcal{D}(H)| = \left|\mathcal{D}^k(H)\right|$ yields the claim. $\square$

Using Theorem 4.4, we can bound the probability of missing a pattern that is μ-important in a large number of graphs in the database. Even for relatively small values of k, most frequent subtrees can be expected to be found, if they are μ-important in only marginally more graphs than required by the threshold. For example, if we sample $k = 20$ spanning trees per graph and set the frequency threshold to $t = 500$, we find any H that is $\mu = 0.15$-important in at least 527 graphs in the original database with probability greater than 95%. Figure 4.4 shows the probability of missing a tree pattern H that is μ-important on at least $t + x$ patterns for different k (Figure 4.4 (a)) and different values of t (Figure 4.4 (b)). Note that the bound given by Theorem 4.4 does not depend on the size of the overall database $\mathcal{D}$; we expect, however, that with increasing size of the database the actual probability of missing μ-important patterns (in the sense of the theorem) will decrease as we can expect to draw more spanning trees containing H from graphs in $\mathcal{D}$ where H was not μ-important.

As we are interested in solving the FTM $_{Relaxed}$ problem, we can bound the probability of missing patterns if we keep the frequency thresholds of the exact FTM problem and the FTM $_{Relaxed}$ problem we are considering identical as described in Section 4.1.1. We note that Theorem 4.4 can be used to bound the number of false negatives also in the case that we lower the frequency threshold of the FTM $_{Relaxed}$ problem we are solving. This, however, would result in false positives with high probability, i.e., patterns found to be frequent in the forest database $\mathcal{D}^k$ for threshold $t' < t$ that are not t-frequent in $\mathcal{D}$. We are not aware of any efficient way to remove such false positive patterns from the output. One obvious way would be to solve the (intractable) SubgraphIsomorphism problem (P2) exactly for each pattern tree and transaction graph (in fact, Chen et al, 2009, propose to do exactly that). But this would immediately destroy the advantageous polynomial delay guarantee of our approach.

4.1.3. Implementation Issues and Runtime Analysis

So far, we have not discussed the complexity of Algorithm 4.1 and have left out the implementation details. Line 3 of Algorithm 4.1 can be implemented using the algorithm of Wilson (1996), which has an expected runtime of $O\left(|V(G)|^3\right)$ in the worst case. In fact,



(a) Influence of different numbers of sampled spanning trees k for fixed threshold t and importance μ

(b) Influence of different frequency thresholds t for fixed k and μ

Figure 4.4.: Probability of missing a tree pattern H that is μ-important on $t + x$ graphs in a graph database when sampling k spanning trees per graph and using frequency threshold t.

it is conjectured to be much smaller for most graphs (Wilson, 1996). Thus, the sampling step of our algorithm runs in expected $O\left(k\,|V(G)|^3\right)$ time. If we do not require the spanning trees to be drawn uniformly, we can improve this time and achieve a deterministic $O\left(k\,|E(G)| \log |V(G)|\right)$ runtime. This is achieved by choosing a *random* permutation of the edge set of a graph and then applying Kruskal's minimum spanning tree algorithm (Kruskal, 1956) 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.

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 as detailed in Section 2.1. For each tree in $\mathfrak{S}_k(G)$, this can be done in $O\left(|V(G)| \log |V(G)|\right)$ time as detailed in Section 2.1. 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 of Algorithm 4.1.

Finally we note that for Line 5 of Algorithm 4.1, we can use any one of the existing algorithms generating frequent connected subgraphs (i.e., subtrees) from forest databases (cf. Section 3.2). However, many of these algorithms do not guarantee polynomial delay even for forest transaction databases. We therefore implemented a polynomial delay FTM algorithm based on the generic algorithm from Section 2.2 that uses the subtree isomorphism algorithm of Shamir and Tsur (1999) as embedding operator.

4.2. Experimental Evaluation

We now empirically evaluate our probabilistic frequent subtree mining algorithm described in Section 4.1 on the datasets described in Section 2.4. First we present results indicating that it is up to an order of magnitude faster than any comparable frequent subgraph mining algorithm on several graph databases beyond chemical graphs and that it is able to mine frequent patterns also in situations where state-of-the-art algorithms do not. Next, we demonstrate that the recall of its output is high with respect to the set of all frequent subtrees. Then we give empirical evidence that probabilistic frequent subtrees are stable under resampling of the random spanning trees. Finally, we show that the predictive performance of probabilistic frequent subtree based learners is comparable to that of exact frequent subtree and frequent subgraph based learners.

We compare our probabilistic frequent subtree mining algorithm, which we call PS, with Gaston and FSG (see Section 3.2). Both programs are used in the versions provided by the authors.[6] FSG is a levelwise algorithm similar to Algorithm 2.1. The FSG implementation only computes the set of all frequent subgraphs. To obtain the set of frequent subtrees, one hence needs to compute the set of frequent subgraphs and remove those graphs that contain cycles; this can be done in linear time. Gaston implements a depth-first search in the pattern space. It allows to mine the set of frequent subgraphs, as well as the set of frequent subtrees via a command line argument of the program. Gaston is available in two variants: One that stores embedding information in memory (Gaston), and one that re-evaluates its embedding operator (Gaston-re) which has a smaller memory footprint but is slower. We compare to both variants. We also tried to include gSpan.[7] However, the 64-bit version used quite a lot of memory and was repeatedly killed by our operating system on almost all datasets and parameter settings. We hence refrain from comparing to any results of gSpan in this study.

Based on the preliminary experiments from Chapter 1, we focus on mining frequent patterns up to size 10. In this section we use relative frequency thresholds for ease of comparison among datasets with different numbers of transaction graphs. All our experiments are conducted on a Linux desktop machine with Intel i7-4770 CPU at 3.40GHz and 16GB of RAM. We use only a single core at a time, as all three implementations are single-threaded.

[6] Gaston: `http://liacs.leidenuniv.nl/~nijssensgr/gaston/download.html`
FSG: `http://glaros.dtc.umn.edu/gkhome/pafi/download`

[7] gSpan: `https://www.cs.ucsb.edu/~xyan/software/gSpan.htm`

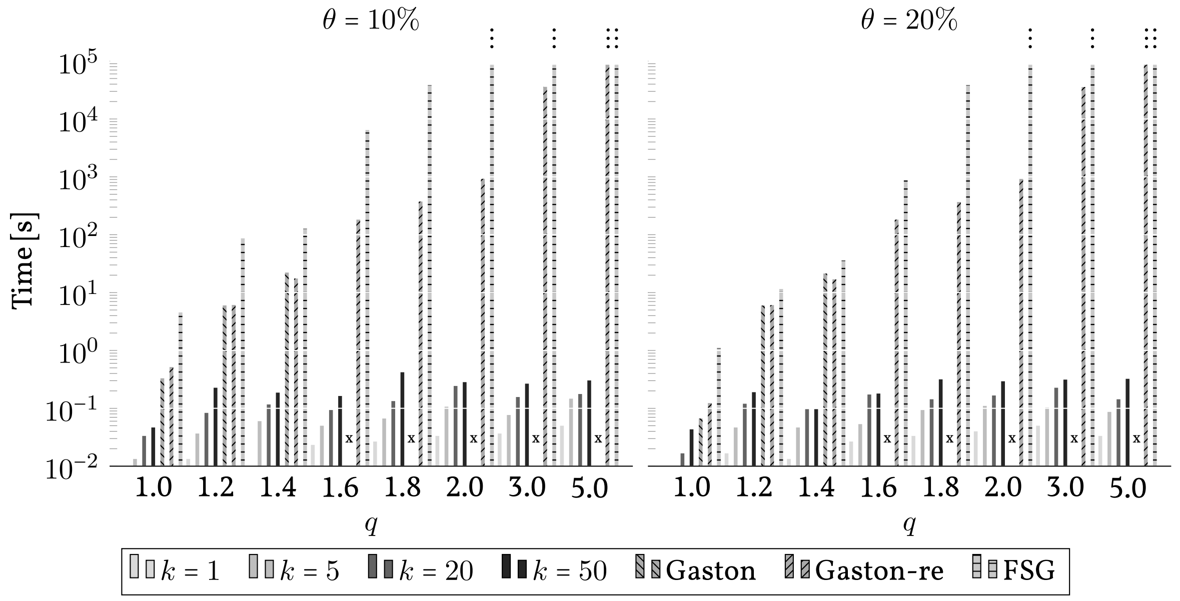


Figure 4.5.: Runtime of our method, compared to FSG and Gaston on Erdős-Rényi datasets of varying expected edge factor q. Dots over bars signal that the run was terminated after 24 hours, a small x indicates that the algorithm terminated with an error.

4.2.1. Runtime

We compare the runtime of PS, FSG, and both Gaston variants on artificial, social, and chemical graph datasets. For our algorithm we report the combined time for sampling and frequent pattern generation. Our implementation of PS generates frequent trees with levelwise search. It uses the algorithm of Shamir and Tsur (1999) as the subroutine for the support counting step and the algorithm of Wilson (1996) for sampling spanning trees. 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. Our algorithms were implemented in C and compiled using gcc.

Random Graphs

Figure 4.5 shows the runtime on unlabeled Erdős-Rényi datasets for expected edge factors q varying between 1.0 and 5.0. (Note the $\log$ scale for the y-axis.) That is, by increasing q we decrease the sparsity of the generated graphs, compare Section 2.4. 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, Gaston, and Gaston-re are 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). This was consistently the case once q exceeded 1.8. Gaston, on the other hand, terminated with memory allocation errors for $q > 1.4$. Gaston-re was able to compute the frequent patterns up to $q = 3.0$ in less than a day (roughly ten hours for $q = 3$) but failed to terminate in a day for $q = 5.0$. Up to 50 sampled spanning trees, our probabilistic approach is always faster than all competitors, outperforming them by at least one order of magnitude. For large q our method still terminates in less than a second for all k, while FSG was aborted after a day without finishing and Gaston failed.

To show the scalability with database size, we generated random graph databases with 1 000 graphs (of at most 50 vertices, each) and ran probabilistic frequent subtree mining on them for varying expected edge factors. Our method was able to mine frequent patterns on all such graph databases, while FSG and the Gaston variants were again not able to finish in one day. It is worthwhile noting that for the large random databases the runtime of our mining technique did not depend on the expected edge factor q of the graphs, but only on the sampling parameter k. For $k = 1$, mining took at most 6 seconds and for $k = 10$ at most 26 seconds, independently of q. Scaling of the mining time for the parameters in between was roughly linear.

Social Graphs

Neighborhood graphs extracted from social networks pose a huge challenge for existing exact frequent subtree and frequent subgraph miners. In particular, in the disk variant any neighborhood graph (ego net) has a central vertex of high degree, whereas chemical graphs usually have a small constant vertex degree (compare Table 2.1). Furthermore, for chemical graphs the difference between the number of vertices and the number of edges is a small constant, which is not the case for ego nets. The vertex degree seems to be an important parameter of the complexity of many SubgraphIsomorphism algorithms (compare Section 3.1). We will also see below that the number of different embeddings of tree patterns into ego nets seems to be very high, resulting in exploding runtime and memory requirements of algorithms that solve the SubgraphIsomorphism problem by explicitly computing and storing all possible embeddings of a pattern.

We therefore for a start only considered the first 100 ego nets (according to the vertex numbering in the original data) of the unlabeled POKEC social network. Still, neither FSG nor Gaston were able to generate any frequent patterns. In contrast, our probabilistic frequent subtree mining algorithm was able to mine patterns up to size 10 for both neighborhood and disk variants of POKEC. Table 4.1 shows the average runtimes over three independent runs of the algorithms for the unlabeled dataset variants. It can be seen that the mining takes 2-3 times as long on the disk variant, compared to the neighborhood variant for any fixed k and θ in the labeled as well as the unlabeled case.

Next, we consider ego net databases of 1 000 ego-nets each from HEPPH and ENRON (see Table 4.1). Impressively, FSG is able to mine frequent patterns on both variants of these datasets, but takes quite some time to find the 201 frequent subtree patterns. Both Gaston variants, however, fail on all these datasets. Our method, on the contrary, works on all datasets. We notice that the runtime of our method is more sensitive to the average

		Disks			Neighbors		
θ	Method	POKEC	HEPPH	ENRON	POKEC	HEPPH	ENRON
5%	FSG	>1d	10 899.83	50 419.01	>1d	10 845.28	14 936.00
	GASTON	err	err	err	err	err	err
	GASTON-re	>1d	>1d	>1d	>1d	>1d	>1d
	PSF $l = 1$	1.09	2.76	3.19	0.72	2.42	2.67
	PSF $l = 2$	1.40	3.82	4.33	0.85	3.05	3.29
	PSF $l = 5$	1.65	5.45	5.70	1.05	4.58	4.33
	PSF $l = 10$	1.94	7.56	7.50	1.49	6.28	5.40
10%	FSG	>1d	10 886.86	48 479.19	>1d	10 838.66	15 796.61
	GASTON	err	err	err	err	err	err
	GASTON-re	>1d	>1d	>1d	>1d	>1d	>1d
	PSF $l = 1$	1.20	2.52	3.40	0.70	2.28	2.44
	PSF $l = 2$	1.45	3.60	4.24	0.87	2.98	3.10
	PSF $l = 5$	1.64	5.61	5.88	1.15	4.37	4.79
	PSF $l = 10$	2.04	7.83	7.42	1.46	5.73	5.79
20%	FSG	>1d	11 126.28	46 542.58	>1d	10 832.73	15 924.55
	GASTON	err	err	err	err	err	err
	GASTON-re	>1d	>1d	>1d	>1d	>1d	>1d
	PSF $l = 1$	1.10	2.03	2.56	0.71	1.57	1.97
	PSF $l = 2$	1.32	2.95	3.53	0.87	2.26	2.69
	PSF $l = 5$	1.63	4.99	5.96	1.13	3.62	4.25
	PSF $l = 10$	1.93	6.49	7.42	1.49	4.72	5.56

Table 4.1.: Runtime (in seconds) of PS, FSG, and Gaston on ego nets extracted from social networks. "err" denotes that the algorithm terminated with an error, while ">1d" indicates that we terminated the algorithm after running for a day (86 400s).

		unlabeled				labeled			
		disks		neighbors		disks		neighbors	
θ	k	time	#	time	#	time	#	time	#
5%	1	1.09	201	0.72	201	68.76	50 432	54.84	50 339
	2	1.40	201	0.85	201	145.10	61 374	119.49	61 683
	5	1.65	201	1.05	201	450.42	64 542	318.13	64 405
	10	1.94	201	1.49	201	893.91	64 605	564.48	64 573
10%	1	1.20	201	0.70	201	32.51	19 665	28.00	20 963
	2	1.45	201	0.87	201	133.45	44 889	102.30	47 274
	5	1.64	201	1.15	201	437.27	63 358	327.45	62 338
	10	2.04	201	1.46	201	895.19	64 512	655.00	64 098
20%	1	1.10	201	0.71	200	10.81	4 229	4.52	2 677
	2	1.32	201	0.87	200	45.95	12 509	30.38	9 982
	5	1.63	201	1.13	201	322.86	40 790	258.09	42 261
	10	1.93	201	1.49	201	854.72	60 707	623.10	58 766

Table 4.2.: Runtime (in seconds) and number [#] of probabilistic frequent subtrees found by our method on ego nets extracted from the POKEC social network. The labeled variants (right) encode the gender of the users and have much larger sets of probabilistic frequent subtrees. Note that neither FSG nor Gaston were able to produce any output on these datasets.

number of edges in the transactions, than to the average number of vertices: The runtimes for the disk variants are between factors of 1.5 and 4 larger than for the respective neighbor variants. The average runtimes (over both parameters of the algorithm) of the datasets also grow almost monotonically with the average number of edges in the graphs.

For the POKEC dataset, there were also vertex labels available indicating whether the user represented by the vertex is male, female, or did not provide gender information. Table 4.2 shows the runtimes and number of probabilistic frequent subtree patterns found by our algorithm. We note that there are a lot more frequent patterns that are discovered by the probabilistic frequent subtree mining algorithm, compared to the unlabeled variants of the datasets. In fact, in this setting, increasing the sampling parameter k of our mining algorithm results in a drastic increase in the number of probabilistic frequent tree patterns that are returned by our algorithm.

As the labels do not change the topology of the graphs we suspect that one reason for the bad performance of FSG and Gaston on the unlabeled datasets must be the huge number of possible embeddings of a given unlabeled tree pattern in the text graphs. FSG and Gaston essentially store (or recompute) *all* embeddings, while our probabilistic frequent subtree mining algorithm only stores v-characteristics, whose number is bounded by a polynomial in the size of the graphs. This gives our algorithm a practical advantage over the other algorithms that are unable to return results in reasonable time or even at all.

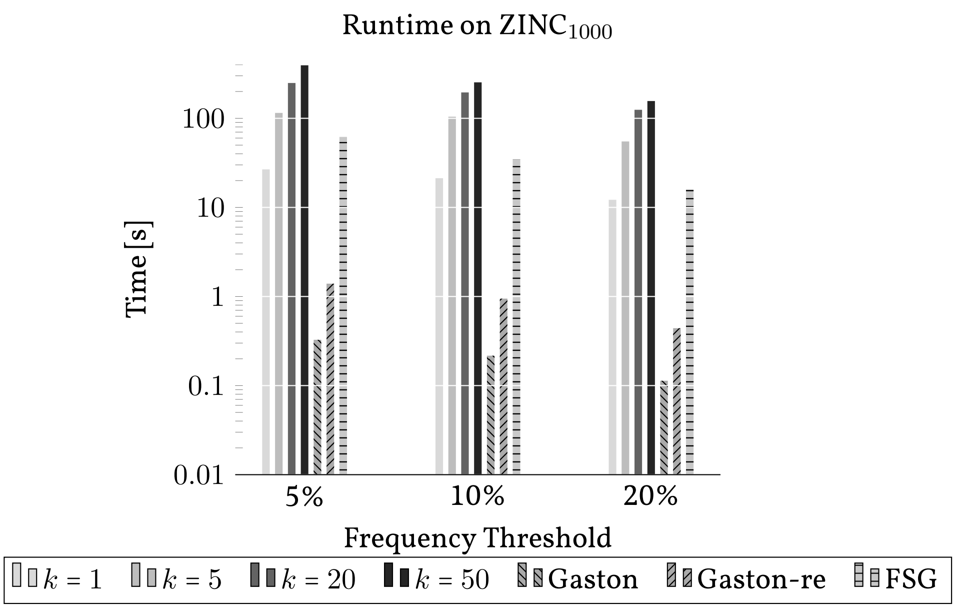


Figure 4.6.: Runtime results on ZINC$_{1000}$ (in seconds) for PS, FSG, and Gaston for different frequency thresholds $\theta \in \{5\%, 10\%, 20\%\}$.

Chemical Graphs

Figure 4.6 reports the runtime results (in seconds) on a subset of 1 000 molecules of the ZINC dataset for FSG, both Gaston variants, and for PS with $k \in \{1, 5, 20, 50\}$. In contrast to the runtime on artificial and social datasets, our method is faster than FSG only for $k = 1$, while being slower than FSG even for the case of $k = 5$. Gaston's speed on this dataset is impressive; it outperforms FSG and PS by at least an order of magnitude. Both variants process the dataset in less than a second for all frequency thresholds.

We therefore assume that FSG and Gaston are highly optimized for structurally very simple *labeled* graphs, where they have a competitive advantage over our method. 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. Recall that for Erdős-Rényi datasets with expected edge factor $q = 1.0$ were able to compute the set of frequent patterns.

4.2.2. Recall

As discussed in Section 4.1, for any graph database $\mathcal{D}$ the pattern set $\mathcal{F}$ found by our probabilistic mining algorithm is a subset of all frequent subtrees F_T, which in turn is a subset of all frequent subgraphs F. We now analyze the recall of our method, i.e. the amount of frequent subtree patterns that are found when applying Algorithm 4.1 for various values of k and θ. To this end, let the ***recall*** $R(k, \theta) := \frac{|\mathcal{F}|}{|F_T|}$ be the fraction of θ-frequent tree

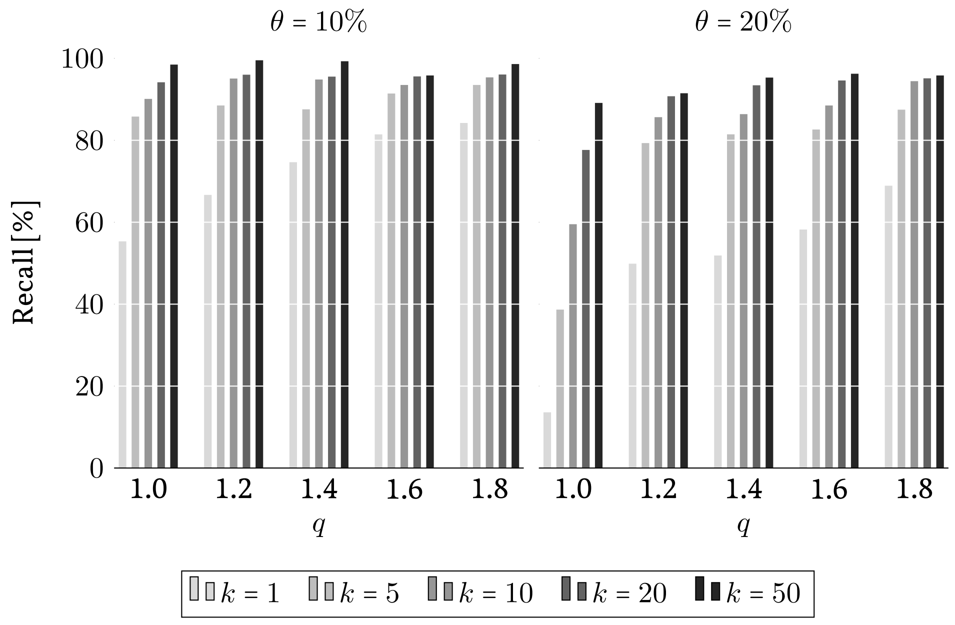


Figure 4.7.: Recall of our method on Erdős-Rényi graph databases with varying expected edge factor q, for frequency thresholds 10% and 20%.

patterns found by Algorithm 4.1 for k random spanning trees. Using one of the exact frequent subgraph miners, we compute the set of frequent trees. As mentioned above, this can be done with either FSG or Gaston, as long as they are able to produce results.

Figure 4.7 shows the recall $R(k, \theta)$ of our method for one run on Erdős-Rényi datasets and for frequency thresholds 10% and 20%. It is restricted to expected edge factors $q \leq 1.8$, as neither FSG nor Gaston are able to compute the full set of frequent patterns within a day beyond this value for the remaining datasets with $q > 1.8$. We will discuss this issue in Section 4.2.1 below. 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 the chemical graph datasets 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 4.3 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 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.

Dataset	θ	$k = 1$	$k = 2$	$k = 3$	$k = 10$	$k = 20$
	5%	20.13 ± 1.20	35.53 ± 1.34	46.48 ± 0.51	78.32 ± 0.85	91.11 ± 1.29
NCI-HIV	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
	5%	36.80 ± 0.87	56.70 ± 1.65	68.42 ± 0.94	92.50 ± 0.45	97.92 ± 0.55
ZINC	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 4.3.: Recall with standard deviation of the probabilistic tree patterns on the NCI-HIV and ZINC datasets for frequency thresholds 5%, 10%, and 20%

We were not able to compute the exact set of frequent subtrees or frequent subgraphs on social graphs using FSG or Gaston. Nonetheless we know that the recall of our method is above 90% on these graphs for $k = 10$, in most cases even for $k = 1$. This is due to the fact that there are exactly 201 pairwise non-isomorphic spanning trees of size up to 10 vertices if there are no vertex and edge labels (Sloane, 2016). Hence the number of frequent tree patterns cannot be larger than that. For 10 sampled spanning trees per block our method has above 90% recall (except on the neighbor variant of HEPPH for $\theta = 20\%$); often it finds almost all frequent subtrees (for $\theta = 5$ even for a single sampled spanning tree per block). The exact numbers of frequent patterns found by PS are presented in Tables 4.1 and 4.2. The recall of PS on the labeled POKEC variants remains an open question. Up to our knowledge there are no formulas for the number of nonisomorphic trees in the general labeled case to provide a lower bound on the recall.

4.2.3. Stability of Probabilistic Subtree Patterns

The results of Section 4.2.2 above indicate that a relatively high recall of the frequent tree patterns can be achieved on molecular, social, and random graph databases, even for a very small number of random spanning trees. We now report empirical results showing that the output pattern set of Algorithm 4.1 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 PS 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 chemical 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 PS five 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 4.4 reports $|\mathcal{F}_i \setminus U_{i-1}|$, i.e., the number of *new* probabilistic subtree patterns found in iteration i for $i \in \{1, \dots, 5\}$ 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 reduces to at most 22 in the upcoming iterations.

		Iteration					
Dataset	k	1	2	3	4	5	$\overline{\mathfrak{S}(G)}$
NCI-HIV	1	3 920	20	5	10	14	169
ZINC$_{40k}$	1	9 898	18	17	11	10	36
ER-1.0	10	692	2	5	8	3	7
ER-1.2	10	750	2	0	0	11	55
ER-1.4	10	806	18	0	0	0	2267
ER-1.6	10	824	1	0	0	0	$3.4 \cdot 10^5$
ER-1.8	10	824	2	0	0	0	$8.7 \cdot 10^7$
ER-2.0	10	850	0	0	1	0	$1.9 \cdot 10^9$
ER-3.0	10	814	26	1	4	0	$9.9 \cdot 10^{15}$
ER-5.0	10	822	4	0	0	20	$1.1 \cdot 10^{22}$
POKEC disks	1	18 852	4 606	5 780	5 545	4 275	$1.3 \cdot 10^{26}$
	5	62 772	910	1 578	690	799	
	10	64 505	60	61	26	60	
	20	64 599	6	5	4	10	
POKEC neighbors	1	19 671	6 442	6 911	8 181	6 017	–
	5	62 637	669	1 222	947	1 354	
	10	64 113	211	242	220	186	
	20	64 499	52	28	51	13	

Table 4.4.: 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. The number in iteration 1 is the number of probabilistic subtrees found. $\overline{\mathfrak{S}(G)}$ denotes the median number of spanning trees per graph in the dataset for comparison.

We observed this behavior consistently on the artificial Erdős-Rényi graphs (over all observed edge factors, all numbers of sampled spanning trees, and all frequency thresholds). Table 4.4 shows the results for $\theta = 10\%$, $k = 10$, and 5 iterations. Each artificial dataset consists of 50 graphs. In contrast to the experiments in the previous sections, however, we label the graphs using ten vertex labels and two edge labels, respectively. The number of newly discovered probabilistic patterns cannot be large for unlabeled Erdős-Rényi graphs, as the recall in this case is very close to one (cf. Section 4.2.2 above). To put our recall and stability experiments into context, note that the median[8] number of spanning trees per graph is depicted in Table 4.4, as well.

Finally, we repeat this experiment for the labeled variants of the POKEC dataset. With the same argument as above, we know that the pattern sets found for the unlabeled variant must be very stable. The two labeled datasets show a relatively large number of newly discovered patterns for each of the five iterations and $k = 1$ sampled spanning trees. Each

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

newly discovered set of probabilistic frequent tree patterns in the POKEC variants newly discovers between 4 200 and 8 200 patterns not contained in the union of the patterns from previous iterations. However, this number drops dramatically once we increase k. For $k = 5$ the fraction of newly found patterns (with respect to the union of previously found patterns) is below 3% in all iterations. For $k = 20$, this number drops to at most 52 newly discovered patterns.

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.

4.2.4. Predictive Performance

In this section we show that the predictive performance of probabilistic subtree patterns compares favorably with that of the frequent subgraph patterns. We deliberately consider the more expressive complete output of FSG, including also frequent subgraphs containing cycles, because we compare the runtime of our method to that of FSG needed to compute *all* frequent subgraphs. Recall from the preliminary experiments in Chapter 1 that the predictive performance of frequent subtrees is very similar.

We choose, as does most related work, a wrapper method and report the achieved area under the ROC-curve (AUC) of a well trained support vector machine (SVM) (Cortes and Vapnik, 1995). To this end, we consider the seven binary classification problems described in Section 2.4. We compare the predictive performance of (i) the frequent subgraph patterns computed by FSG (Deshpande et al, 2005) with that of (ii) the probabilistic frequent subtree patterns for different k and for different frequency thresholds. We restrict our evaluation of the predictive performance to chemical graphs only. These graph databases are the only ones where our competitors could reliably produce results. For (ii), we use only the results with Wilson's random spanning tree sampling algorithm (Wilson, 1996); we obtained nearly identical accuracy and runtime results with the greedy sampling algorithm based on Kruskal's method (cf. Section 4.1.3). For our evaluation, we use the SVM provided by the libSVM package (Chang and Lin, 2011) with a radial basis function kernel. We repeat Algorithm 4.1 four times using different sets of sampled trees, resulting in different sets of probabilistic subtrees and different embedding vectors of the database graphs. We report the average and standard deviation of AUC values from a 3-fold cross validation for each resulting database representation. The same procedure is applied to the frequent subgraph pattern set, here we use a different splitting for the cross validation in each run.

Note that this evaluation requires us to compute feature vectors for each graph in the databases. Here, we compute the feature vectors and sets of (probabilistic) frequent patterns simultaneously for the full data sets using the frequent subgraph mining al-

θ	k	MUTAG	PTC	NCI1	NCI109
1%	1	81.72 ± 1.22	56.20 ± 1.54	79.73 ± 0.26	78.64 ± 0.20
1%	2	82.98 ± 0.46	57.03 ± 0.88	81.74 ± 0.22	80.89 ± 0.15
1%	5	85.47 ± 0.80	59.18 ± 0.54	83.45 ± 0.12	83.07 ± 0.14
1%	10	88.33 ± 0.30	59.67 ± 0.26	84.09 ± 0.10	83.79 ± 0.15
1%	20	89.32 ± 0.14	60.10 ± 0.09	84.43 ± 0.06	84.23 ± 0.05
1%	FSG	91.18 ± 0.46	63.62 ± 1.01	86.87 ± 0.10	86.84 ± 0.09
5%	1	80.79 ± 1.26	54.92 ± 1.69	76.90 ± 0.40	75.67 ± 0.23
5%	2	82.30 ± 0.41	55.05 ± 1.25	78.87 ± 0.17	77.73 ± 0.17
5%	5	84.20 ± 0.90	56.12 ± 0.67	80.75 ± 0.17	80.31 ± 0.16
5%	10	86.35 ± 0.15	56.14 ± 0.29	81.60 ± 0.10	81.12 ± 0.13
5%	20	87.66 ± 0.26	56.34 ± 0.19	82.15 ± 0.05	81.73 ± 0.05
5%	FSG	89.01 ± 0.64	58.00 ± 1.86	83.76 ± 0.13	83.86 ± 0.06
10%	1	80.99 ± 1.23	54.05 ± 1.84	75.41 ± 0.43	74.10 ± 0.28
10%	2	82.60 ± 0.44	54.35 ± 1.48	77.28 ± 0.22	76.08 ± 0.17
10%	5	84.22 ± 0.86	54.17 ± 0.87	79.09 ± 0.16	78.05 ± 0.14
10%	10	86.23 ± 0.16	53.94 ± 0.28	79.95 ± 0.09	79.01 ± 0.10
10%	20	86.95 ± 0.11	53.99 ± 0.19	80.44 ± 0.05	79.61 ± 0.07
10%	FSG	87.34 ± 0.46	56.76 ± 1.96	81.66 ± 0.10	81.55 ± 0.24
20%	1	81.02 ± 1.43	53.36 ± 2.16	72.78 ± 0.35	70.84 ± 0.32
20%	2	83.12 ± 0.53	53.05 ± 0.79	74.94 ± 0.22	73.77 ± 0.17
20%	5	84.68 ± 0.82	52.34 ± 0.89	77.05 ± 0.15	76.13 ± 0.11
20%	10	86.92 ± 0.16	51.86 ± 0.52	77.79 ± 0.06	76.90 ± 0.10
20%	20	88.10 ± 0.06	51.97 ± 0.22	78.15 ± 0.06	77.33 ± 0.08
20%	FSG	88.36 ± 0.00	55.82 ± 2.59	77.41 ± 0.09	77.92 ± 0.02

Table 4.5.: AUC values [%] of an SVM classifier on MUTAG, NCI1, NCI109, and PTC for frequency thresholds t between 1% and 20% when using features generated by FSG and our method for $k \in \{1, 2, 5, 10, 20\}$.

gorithms. We defer the detailed discussion of this topic to Chapter 6. In particular, we do not discuss here the case that finding a feature representation should be part of the learning process.

Table 4.5 shows the results for the classification problems on MUTAG, NCI1, NCI109, and PTC. We can see that the frequent subgraph patterns outperform our probabilistic subtree patterns for all frequency thresholds and all choices of k. However, if we select $k = 20$ spanning trees, the accuracy is fairly close to that of the exact frequent subtree patterns for all datasets and for all frequency thresholds. Furthermore, the results suggest that we can achieve or perhaps even increase the predictive accuracy of the exact frequent subgraph patterns at a certain frequency threshold θ by using the probabilistic frequent subtree patterns with parameters $k = 20$ and frequency threshold $\theta/2$. It is also worth noting that the increase of accuracy slows down as a function of k; the gain of increasing

θ	k	AvsI	AMvsI	AvsMI
5%	FSG	o.o.m	o.o.m	o.o.m
5%	1	89.27 ± 0.20	72.35 ± 0.23	88.23 ± 0.24
5%	2	89.94 ± 0.12	74.09 ± 0.69	89.09 ± 0.74
5%	5	91.17 ± 0.13	75.65 ± 0.27	90.63 ± 0.17
10%	FSG	91.31 ± 0.38	75.29 ± 0.24	90.82 ± 0.31
10%	1	88.53 ± 0.81	71.32 ± 0.54	87.45 ± 1.18
10%	2	88.28 ± 1.51	71.09 ± 0.21	87.29 ± 0.62
10%	5	91.11 ± 0.23	74.30 ± 0.18	90.27 ± 0.08
20%	FSG	91.35 ± 0.39	74.24 ± 0.26	90.57 ± 0.17
20%	1	86.75 ± 0.76	68.55 ± 0.73	86.00 ± 0.74
20%	2	86.40 ± 1.00	68.79 ± 0.61	85.79 ± 0.74
20%	5	90.29 ± 0.28	73.17 ± 0.56	90.27 ± 0.53

Table 4.6.: Average AUC values for the three learning problems on the NCI-HIV benchmark dataset for the frequent subgraph patterns and the probabilistic frequent subtree patterns for $k = 1, 2$ and for different frequency thresholds.

k from one to five spanning trees is much larger than that of increasing k from five to ten on all datasets except MUTAG, where the second increase is comparable to the first. We assume that this behavior on MUTAG is due to the small number of molecules in the dataset.

The results on NCI-HIV are presented in Table 4.6. On the one hand, one can see that from a frequency threshold of 10%, the results using frequent subgraph patterns are more stable than those with the probabilistic frequent subtree patterns on all three problems. Though the frequent subgraph patterns outperform the probabilistic frequent subtree patterns on the same frequency threshold, the difference seems marginal once we compare the best results on each problem, especially in light of the runtime benefits presented above. On the other hand, however, for the frequent subgraph patterns, the SVM could be trained only for $\theta = 10\%$, while for the probabilistic frequent subtree patterns we obtained the result in half of the time for $\theta = 5\%$. For this frequency threshold, FSG was unable to produce any result because it ran out of memory. For larger frequency thresholds, we had difficulties with training the SVM using all frequent patterns because of its excessive memory usage. These observations clearly show the limitation of the frequent subgraph patterns over the probabilistic frequent subtree patterns when the predictive performance required can be achieved only for low frequency thresholds. Finally we note that there is no improvement when sampling two instead of one spanning tree per graph, but a drastic increase when increasing k to five. This result fits well with the evaluation of our method on the artificial datasets.

4.3. Summary

We have presented a method to mine probabilistic subtree patterns, i.e., subtrees in a forest database obtained by randomly selecting k spanning trees for each transaction graph in the input database and for some small value of k. Our empirical results on Erdős-Rényi random graphs, ego nets from social networks, and chemical graphs show that even for small values of k ($k \leq 20$), the output of the probabilistic frequent subtree mining algorithm is of high recall and stability. Runtime comparisons with the FSG and Gaston frequent subgraph mining algorithms clearly demonstrate the superiority of our probabilistic approach on graph datasets beyond chemical graphs: The speed of the probabilistic frequent subtree mining algorithm is faster by at least one order of magnitude in the few cases where FSG and Gaston were able to terminate in reasonable time. Furthermore, our method allowed to mine probabilistic frequent subtrees where the traditional exact methods failed. Our empirical results on various real-world benchmark graph datasets show that the probabilistic feature space considered is expressive enough in terms of predictive performance compared to that of ordinary frequent subgraphs.

One of the strengths of our method is that it is not restricted to any particular graph class. This advantageous property allows us to empirically investigate frequent subtree sets on more complicated graph classes beyond molecular graphs, such as random graphs of medium to high edge density, ego nets and possibly many other types of graph transaction databases, such as knowledge graph databases.

However, so far we did not address two important questions that remain to be answered in the subsequent chapters of this thesis: First, the runtime of Algorithm 4.1 directly depends on the number k of spanning trees sampled for each graph. Generally, the number of spanning trees is large for all but the most simple graphs (e.g. cycles). In fact, Cayley's formula tells us that there can be up to n^{n-2} spanning trees in a graph with n vertices (Cayley, 1889). Thus there is an exponentially large gap between the number of spanning trees our algorithm considers and the total number of spanning trees. We thus will investigate in Chapter 5 how to increase the number of spanning trees that can be efficiently considered. In this way, we try to boost the recall of the probabilistic subtrees and hence likely their predictive performance.

Another question yet to be addressed is how to efficiently compute the embedding vectors of graphs, given a set of (probabilistic) frequent subtrees. Although this problem appeared in Section 4.2.4 as a subtask, the techniques described in this chapter are not sufficient to solve it completely. Hence, a crucial step is missing for a complete subtree mining system that can be used in an inductive setting. Algorithm 2.1 and Algorithm 4.1 can both be modified to output the embedding vectors for the graphs in their input database $\mathcal{D}$. However, novel graphs[9] cannot be embedded right away in the feature space spanned by the (probabilistic) subtrees. We address the efficient computation of such embeddings in Chapter 6.

[9] That is, graphs that were not part of the database $\mathcal{D}$ used for probabilistic frequent subtree mining.

5. Boosted Probabilistic Frequent Subtrees

Utilizing that a tree is subgraph isomorphic to a graph if and only if it is subtree isomorphic to one of the graph's spanning trees, Algorithm 4.1 from the previous Chapter generates probabilistic frequent subtrees in the following simple way: It replaces each transaction graph in the input database by a forest formed by the vertex disjoint union of a *random* subset of its spanning trees and generates the set of frequent connected subgraphs (i.e., subtrees) for the forest database obtained. These probabilistic frequent subtrees can be enumerated with polynomial delay if the number of spanning trees in the sample is bounded by a polynomial of the graph's size for each transaction graph in the database. For all but the most simple graphs, however, the number of spanning trees is exponential in the size of G. In fact, the number of spanning trees of G is exponential in the *cyclomatic number* $|E(G)| - |V(G)| + 1$ of G.[1]

Hence, there is an exponential gap in the number of spanning trees that Algorithm 4.1 can efficiently consider and the total number of spanning trees of the transaction graphs. This fact may negatively influence the recall of our probabilistic subtree mining algorithm, as it may lead to a large number of subtrees of the database graphs that are μ-important for very small values of μ only. Such unimportant patterns, however, can only be reliably found if a large number of spanning trees is considered (cf. Section 4.1.2).

In this chapter we go beyond the limitation of processing polynomially many spanning trees only. We present an algorithm which can generate probabilistic frequent subtrees from arbitrary graphs with polynomial delay by considering a potentially *exponentially* large subset of the spanning trees for each graph in the database. The core of our mining algorithm is a pattern matching algorithm that, for a tree pattern H and transaction graph G, (i) partitions G into a certain set of induced subgraphs, (ii) considers a (random) subset of *local* spanning trees for each induced graph, and (iii) decides whether H is subtree isomorphic to one of the *global* spanning trees of G obtained by combining its local spanning trees. It is inspired by the paradigms developed by Matoušek and Thomas (1992) and by Shamir and Tsur (1999) for solving subgraph isomorphism for other graph classes.

In a nutshell, our algorithm decides the pattern matching problem by a dynamic programming algorithm traversing a rooted tree generated for G in a bottom-up manner and computing the final solution from previously calculated partial ones. In our case, the nodes of the rooted tree controlling the evaluation are constructed from the articulation vertices of G. Each node v of such a tree is associated with the set of spanning trees of cer-

[1] In general, the cyclmatic number, or circuit rank of a graph G is $|E(G)| - |V(G)| + c$, where c is the number of connected components of G.

tain biconnected components of G containing v. For all such local spanning trees τ, we solve the partial subtree isomorphism problem corresponding to v by carefully extending the partial subtree isomorphisms already computed for τ. Iterating over all spanning trees and over all nodes, we can correctly decide subtree isomorphism for the part of G which is "below" v in the rooted tree associated with G. We prove that our algorithm decides subgraph isomorphism from H to $\mathfrak{S}$ correctly, where $\mathfrak{S}$ is the set of spanning trees of G that can be obtained from the combinations of the local spanning trees. Furthermore, our algorithm runs in time polynomial in the combined size of H, G, and f, where f is an upper bound on the number of selected local spanning trees. The significance of this result is that the number of global spanning trees in $\mathfrak{S}$ can be *exponential* in f. This property has immediate consequences to probabilistic and exact frequent subtree mining.

Regarding *probabilistic* frequent subtree mining, by (implicitly) considering exponentially many global spanning trees instead of polynomially many ones, our technique has an improved performance in terms of *recall* over the simple algorithm from Chapter 4. On the one hand, this improvement is only marginal on real-world molecular graph datasets, due to the relatively simple graph structure of pharmacological compounds (cf. Horváth and Ramon, 2010; Horváth et al, 2010). On the other hand, however, on *threshold graphs*, which have applications among others in *spectral clustering* (see, e.g., von Luxburg, 2007), the algorithm presented here results in a much higher recall compared to the simple one. It is important to note that all threshold graphs used in our experiments had a structural complexity *beyond* that of the molecular graphs of pharmacological compounds. None of the state-of-the-art frequent subgraph mining algorithms were able to produce any output for threshold graphs in practically feasible time.[2]

Our pattern matching operator implies a novel result regarding the complexity of the (exact) FTM problem, as well. We extend the known positive complexity results on frequent tree mining by a new one formulated for a graph class that is of theoretical as well as practical interest. Recall from Chapter 3 that despite more than two decades of research there are only a few non-trivial theoretical results concerning the complexity of frequent subgraph mining are known. Beyond forests, frequent connected subgraphs can be listed in incremental polynomial time for graphs of bounded tree-width (Horváth and Ramon, 2010). The subgraph isomorphism algorithm proposed in this chapter, however, is always correct if all local spanning trees are considered.[3] Accordingly, a sufficient condition for our frequent pattern mining algorithm to be correct and efficient (i.e., polynomial delay) is that the input graphs are *locally easy*: A graph G of size n is locally easy if for all vertices v of G, the union of the biconnected components containing v has at most $\mathrm{poly}(n)$ spanning trees.

The class of locally easy graphs is *orthogonal* to all graph classes that are defined by a constant upper bound on some *monotone* graph property (e.g., graphs of bounded tree-width); a graph property is called *monotone* if it is closed under taking subgraphs. By *or-*

[2] Recall that we have observed a similar behavior on Erdős-Rényi graphs and social graphs in the previous chapter.

[3] We recall that the problem of deciding whether a tree is subgraph isomorphic to a graph G is NP-complete in general (see, e.g., Garey and Johnson, 1979) and remains computationally intractable even for very simple graphs, e.g., when G is a cactus graph (Akutsu, 1993).

thogonality we mean that the graph class always contains an infinite number of graphs that are not contained in the other graph class. It turns out that the class of locally easy graphs includes a number of interesting and practically relevant graph classes. The most natural example is the class of *forests*. *Pseudoforests* (i.e., graphs in which every connected component has at most one cycle) and their generalizations, *cactus* graphs (i.e., in which all edges belong to at most one simple cycle) of bounded block degree (i.e., the maximum number of blocks sharing a vertex is bounded by a constant) are some further straightforward subclasses of locally easy graphs. Other examples include the class of *d-tenuous outerplanar* graphs (Horváth et al, 2010) of bounded block degree and that of *k-easy* graphs of bounded block degree, where a graph is k-easy for some constant $k \geq 0$ integer if all biconnected components have $O\left(n^k\right)$ spanning trees. These and other graph classes show that our positive result on mining locally easy graphs is an important step towards exploring the border between tractable and intractable fragments of the frequent pattern mining problem. We conjecture that generalizing our positive result to the first "natural" graph class beyond locally easy graphs is at least as difficult as solving the **P** vs. **NP** problem.

Outline

The rest of this chapter is organized as follows. We present our subtree isomorphism algorithm in Section 5.1 and prove its correctness and runtime guarantees. Using this pattern matching algorithm, in Section 5.2 we describe our mining algorithm enumerating probabilistic frequent subtrees in arbitrary graph databases with polynomial delay and empirically compare its runtime and recall on threshold graphs to our algorithm from Chapter 4. We discuss exact frequent subgraph mining for locally easy graphs in Section 5.3, together with some important theoretical and practical properties of this graph class. Finally we conclude in Section 5.4 and mention some interesting open problems for further research.

5.1. An Efficient Embedding Operator for Trees

This section is devoted to the support counting step of our boosted mining algorithm (cf. SupportCount$(H, \mathcal{D})$ in Line 8 of Algorithm 2.1). In Theorem 5.1 below we first claim that SubtreeIsomorphism can be decided in time polynomial in the number of local spanning trees of certain induced subgraphs of G. In Section 5.2 we show that the algorithm used in the proof of this result can be modified in a natural way to decide SubtreeIsomorphism with one-sided error in polynomial time. This is achieved by considering a potentially exponentially large subset of the spanning trees of G, for any arbitrary graph G. This modified algorithm will allow for efficient probabilistic frequent subtree mining. To state Theorem 5.1, our main result for this section, we first introduce the following notation: For a graph G and $v \in V(G)$, let $f_v(G)$ be the number of spanning trees in the union of the biconnected components containing v and define $f_{\max}(G) = \max_{v \in V(G)} f_v(G)$.

Theorem 5.1. *The* SubtreeIsomorphism *problem can be solved in time*

$$O\left(f_{\max}^2(G) \cdot |E(G)| \cdot |V(H)|^{1.5}\right) .$$

To put Theorem 5.1 into context, we note that SubtreeIsomorphism is a well-known **NP**-complete problem (it generalizes e.g. the HamiltonianPath problem). If, however, the transaction graph is a tree, as well, the restricted problem belongs to **P** (see, e.g., Shamir and Tsur, 1999). This positive result, together with that on generating the spanning trees of a graph with polynomial delay (Read and Tarjan, 1975), implies that SubtreeIsomorphism is in **P** if G has only polynomially many spanning trees; just list all spanning trees τ of G and check if H is subgraph isomorphic to τ. Theorem 5.1 generalizes this straightforward positive result to graphs that can have *exponentially* many spanning trees. To prove Theorem 5.1, we present Algorithm 5.1 and show that it decides the SubtreeIsomorphism problem correctly (Lemma 5.6) and in time stated in the theorem (Lemma 5.7).

Algorithm 5.1 is inspired by the ideas in (Matoušek and Thomas, 1992) and (Shamir and Tsur, 1999). Analogously to tree decompositions of bounded tree-width graphs (see, e.g., Diestel, 2012), our dynamic programming algorithm splits G into certain induced subgraphs and evaluates partial (non-induced) subgraph isomorphisms from subtrees of H to such subgraphs. The evaluation order of our algorithm is, however, controlled by a rooted tree skeleton defined on the articulation vertices of G. For all nodes v of the tree skeleton, the biconnected components that are "below" v in G are replaced by a (local) spanning tree τ in all possible ways. The subproblem corresponding to v is then solved by carefully combining τ with the spanning trees of the previous level. Iterating over all (local) spanning trees of the biconnected components, we can correctly decide Subtree-Isomorphism for the part of G which is "below" v. We will now describe the algorithm and necessary notation.

In what follows, H and G denote a tree and an arbitrary graph, respectively. We assume w.l.o.g. that G is connected and that $2 \leq |V(H)| \leq |V(G)|$, implying that all biconnected components of G contain at least two vertices. We fix an arbitrary vertex $r \in V(G)$ and will implicitly also consider r, when talking about G. For a block B of G we define its *root* v to be the vertex of B with the smallest distance to r and will refer to B as a v-*rooted* block. For any $v \in V(G)$, the subgraph formed by the set of v-rooted blocks of G is denoted by $\mathcal{B}(v)$. Clearly, $\mathcal{B}(v)$ can be empty. On the set of roots of the blocks in G we define a directed graph $\mathcal{T}$ as follows (since G and r have been fixed, we omit them in the notation): For any $u, v \in V(\mathcal{T})$ with $u \neq v$, $(u, v) \in E(\mathcal{T})$ if and only if there exists a block B with root v such that $u \in V(B)$. We call $\mathcal{T}$ the *tree skeleton* of G (see, also, Figures 5.1 a) and b)). In the proposition below we show that $\mathcal{T}$ is indeed a rooted tree. This tree will be used to direct our dynamic subgraph isomorphism algorithm.

Proposition 5.2. *$\mathcal{T}$ is a tree rooted at r.*

Proof. It suffices to show that for all $u \in V(\mathcal{T})$ with $u \neq r$, u has outdegree at most one; the claim then follows by noting that the outdegree of r is zero and that $\mathcal{T}$ is connected, as G is connected. Suppose for contradiction that there exists $u \in V(\mathcal{T})$, $u \neq r$, with two

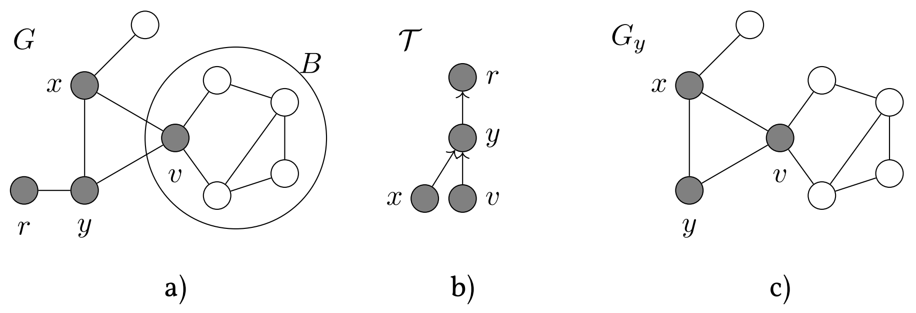


Figure 5.1.: G, tree skeleton $\mathcal{T}$ and G_y for a small graph G (with respect to r). v is the root of the block B. Roots are shown in gray, while vertices that are not roots are shown in white.

different parents $v_1, v_2 \in V(\mathcal{T})$. Let $B_i \in \mathcal{B}(v_i)$ $(i = 1, 2)$. Then $B_1 \neq B_2$ and there is a path P_1 (resp. P_2) in G connecting r and v_1 (resp. v_2) that is edge disjoint with B_1 (resp. B_2). The union of P_1 and P_2 together with the paths connecting u with v_1 and u with v_2 contains a cycle intersecting both B_1 and B_2. But then u, v_1, and v_2 all belong to the same block of G, contradicting the maximality of B_1 and B_2. □

We need some further concepts. Let $v, w \in V(G)$. Then w is ***below*** v if all paths connecting r and w in G contain v. A ***rooted subgraph*** G_v of G for v is the subgraph of G induced by the set of vertices below v (see Figure 5.1 c) for an example). The same notation will be used consistently for the pair consisting of the tree pattern H and some vertex $y \in V(H)$, i.e., for any $u, y \in V(H)$, H_u^y is the tree obtained from the tree H rooted at y by keeping the subtree rooted at u. The definitions and the connectivity of G imply that G_v is connected, $G_r = G$, and G_w is a single vertex if and only if $w \notin V(\mathcal{T})$. A vertex $w' \in V(G)$ is called a ***child*** of v, if $vw' \in E(G)$ and $w' \in V(\mathcal{B}(v))$.

A ***guidance tree*** of G is a pair $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ such that $\mathcal{T}$ is a tree skeleton of G and $\mathcal{S}$ is a family of sets $\mathcal{S}_v$ for all $v \in V(\mathcal{T})$. That is, all nodes v of $\mathcal{T}$ are associated with a set S_v, called the ***bag*** of v. Each $\mathcal{S}_v$ is a subset of the set of spanning trees of $\mathcal{B}(v)$, called ***local*** spanning trees, all rooted at v. If $\mathcal{S}_v$ contains ***all*** spanning trees of $\mathcal{B}(v)$ for every $v \in V(\mathcal{T})$, then $\mathbb{T}$ is referred to as a *complete* guidance tree of G. For the remainder of this section, by guidance trees we always mean complete guidance trees. (Incomplete guidance trees will be considered in Section 5.2.)

Let $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be a guidance tree of G and let $v \in V(\mathcal{T})$. An ***iso-triple***[4] ξ of H relative to v is a triple (H_u^y, τ, w) such that $u \in V(H)$, $y \in \mathcal{N}(u) \cup \{u\}$, $\tau \in \mathcal{S}_v$, and $w \in V(\tau)$. Let G' be an induced subgraph of G and τ' be a spanning tree of G'. Then $G\{G'/\tau\}$ denotes the graph obtained from G by removing all edges of G' that are not in τ (i.e., by substituting G' with τ). Now we are able to define the partial subgraph isomorphisms we are inter-

[4] Though our terminology is similar to that in (Hajiaghayi and Nishimura, 2007), which in turn is based on the concepts in (Matoušek and Thomas, 1992), the definitions of iso-triples and characteristics in this thesis are semantically different from their definitions.

Algorithm 5.1 Subgraph Isomorphism from a Tree into a Connected Graph

```
Input  : tree $H$ with $|V(H)| > 1$ and an arbitrary connected graph $G$ with $|V(G)| \geq |V(H)|$
Output: TRUE if $H \preccurlyeq G$; o/w FALSE

MAIN:
 1: set $\mathcal{C} := \emptyset$
 2: pick a vertex $r \in V(G)$ and compute the complete guidance tree $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ of $G$ for
    the tree skeleton $\mathcal{T}$ rooted at $r$
 3: for all $v \in V(\mathcal{T})$ in a postorder do
 4:     for all $\tau \in \mathcal{S}_v$ do                          // $\mathcal{S}_v \in \mathcal{S}$ is the bag of $v$ in $\mathbb{T}$
 5:         for all $w \in V(\tau)$ in a postorder do
 6:             $\mathcal{C} := \mathcal{C} \cup$ CHARACTERISTICS$(v, u, \tau, w)$
 7:             if $(H_u^u, \tau, w) \in \mathcal{C}$ then return TRUE
 8: return FALSE

FUNCTION CHARACTERISTICS$(v, u, \tau, w)$:
 1: $\mathcal{C}_\tau := \emptyset$
 2: for all $\theta \in \Theta_{vw}(\tau)$ do
 3:     for all $u \in V(H)$ do
 4:         let $\tau'$ be the tree satisfying $\theta = \tau \cup \tau'$
 5:         let $C_\tau$ (resp. $C_{\tau'}$) be the set of children of $w$ in $\tau$ (resp. $\tau'$) and
            $C_\theta := C_\tau \cup C_{\tau'}$
 6:         let $B = (C_\theta \dot{\cup} \mathcal{N}(u), E)$ be the bipartite graph with
            $cu' \in E$ if and only if $(c \in C_\tau \wedge (H_{u'}^u, \tau, c) \in \mathcal{C}) \vee (c \in C_{\tau'} \wedge (H_{u'}^u, \tau', c) \in \mathcal{C})$
            for all $cu' \in C_\theta \times \mathcal{N}(u)$
 7:         if $B$ has a matching that covers $\mathcal{N}(u)$ then
 8:             add $(H_u^u, \tau, w)$ to $\mathcal{C}_\tau$
 9:         for all $y \in \mathcal{N}(u)$ do
10:             if $B$ has a matching covering $\mathcal{N}(u) \setminus \{y\}$ then
11:                 add $(H_u^y, \tau, w)$ to $\mathcal{C}_\tau$
12: return $\mathcal{C}_\tau$
```

ested in. A *v-characteristic* is an iso-triple $\xi = (H_u^y, \tau, w)$ relative to v such that there exists a subgraph isomorphism φ from H_u^y to $(G\{\mathcal{B}(v)/\tau\})_w$ with $\varphi(u) = w$. In the lemma below we provide a characterization of subgraph isomorphisms from H to G in terms of v-characteristics. Its proof follows directly from the definitions.

Lemma 5.3. *Let H be a tree, G be a graph with root r, and $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be a guidance tree of G such that $\mathcal{T}$ is rooted at r. Then $H \preccurlyeq G$ if and only if there exists a v-characteristic (H_u^u, τ, w) for some $v \in V(\mathcal{T})$, $u \in V(H)$, $\tau \in \mathcal{S}_v$, and $w \in V(\tau)$.*

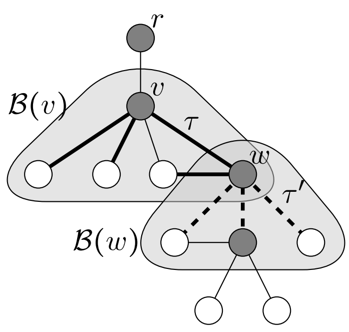


Figure 5.2.: This figure shows a small graph G with its subgraphs $\mathcal{B}(v)$ and $\mathcal{B}(w)$ (depicted by the rounded triangles). One spanning tree τ of $\mathcal{B}(v)$ and τ' of $\mathcal{B}(w)$ are shown in solid bold and dashed bold, respectively.

Notice that the number of v-characteristics (H_u^y, τ, w) is bounded by a polynomial in the number of local spanning trees τ. To be more exact, there are at most $|V(H)|\cdot|V(\mathcal{B}(v))|$ v-characteristics for each local spanning tree $\tau \in \mathcal{S}_v$. We will show how these characteristics can be computed recursively by a post-order traversal of the tree skeleton $\mathcal{T}$. In order to recover all v-characteristics, the spanning trees of the w-rooted blocks must carefully be combined with τ when w itself is also a root (i.e., $w \in V(\mathcal{T})$). To formalize these considerations, we introduce the following notation. For any $v \in V(\mathcal{T})$, $\tau \in \mathcal{S}_v$, and $w \in V(\tau)$ we define $\Theta_{vw}(\tau)$ by

$$\Theta_{vw}(\tau) := \begin{cases} \{\tau \cup \tau' \,:\, \tau' \in \mathcal{S}_w\} & \text{if } w \in V(\mathcal{T}) \setminus \{v\} \\ \{\tau\} & \text{o/w (i.e., if } w \notin V(\mathcal{T}) \text{ or } v = w\text{),} \end{cases}$$

where $\tau \cup \tau'$ is the graph with vertex set $V(\tau) \cup V(\tau')$ and edge set $E(\tau) \cup E(\tau')$. That is, for the case that $w \in V(\mathcal{T}) \setminus \{v\}$, $\Theta_{vw}(\tau)$ is the set of trees obtained by "gluing" the local spanning tree τ and τ' at vertex w, for all local spanning trees $\tau' \in \mathcal{S}_w$. The definition is correct, as $V(\tau) \cap V(\tau') = \{w\}$ for this case. Note that if w is a root vertex different from v then w always has at least one child in $\mathcal{B}(w)$, i.e., τ' is always a tree with at least one edge. As an example, the combination of the dashed and the bold tree in Figure 5.2 denotes an element of $\Theta_{vw}(\tau)$. In Lemma 5.4 below we first provide a characterization of v-characteristics for subtrees H_u^y with $y \in \mathcal{N}(u)$.

Lemma 5.4. *Let H be a tree, G be a graph, and $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be a guidance tree of G. An iso-triple (H_u^y, τ, w) of H is a v-characteristic for some $v \in V(\mathcal{T})$ and $y \in \mathcal{N}(u)$ if and only if there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\mathcal{N}(u) \setminus \{y\}$ to the children of w in θ such that for all $u' \in \mathcal{N}(u) \setminus \{y\}$ there is a subgraph isomorphism $\varphi_{u'}$ from $H_{u'}^u$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\psi(u')}$ mapping u' to $\psi(u')$.*

Proof. "$\Rightarrow$" Suppose (H_u^y, τ, w) is a v-characteristic for some $v \in V(\mathcal{T})$ and $y \in \mathcal{N}(u)$. Then, by definition, there is a subgraph isomorphism φ from H_u^y to $(G\{\mathcal{B}(v)/\tau\})_w$ with $\varphi(u) = w$. Let R be an arbitrary spanning tree of $(G\{\mathcal{B}(v)/\tau\})_v$ containing the image $\varphi(H_u^y)$ as a subtree. Then $R[V(\mathcal{B}(w))] \in \mathcal{S}_w$ and $R[V(\mathcal{B}(v))] = \tau$ and hence $\theta = R[V(\mathcal{B}(v))] \cup R[V(\mathcal{B}(w))] \in \Theta_{vw}(\tau)$ implying that for all $u' \in \mathcal{N}(u) \setminus \{y\}$, φ maps $H_{u'}^u$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\varphi(u')}$. As φ is injective we can set ψ to be the restriction of φ to $\mathcal{N}(u) \setminus \{y\}$. As φ is a subgraph isomorphism, we can set $\varphi_{u'}$ to be the restriction of φ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\varphi(u')}$ for all $u' \in \mathcal{N}(u) \setminus \{y\}$.

"$\Leftarrow$" Let $\varphi : V(H_u^y) \to V(G(\{\mathcal{B}(v)/\tau\})_w)$ with $\varphi : u \mapsto w$ and $x' \mapsto \varphi_{u'}(x')$ for all $u' \in \mathcal{N}(u) \setminus \{y\}$ and $x' \in V(H_{u'}^u)$. Since for all u', $\varphi_{u'}$ is at the same time a subgraph isomorphism from $H_{u'}^u$ to $(G\{\mathcal{B}(v)/\tau\})_w$, it holds that $\varphi_{u'}(u') = \psi(u')$. But then, as ψ is injective, φ is a subgraph isomorphism, implying the claim. □

In Lemma 5.5 we formulate an analogous characterization for the entire pattern H (i.e., for $y = u$). The proof of this lemma is similar to that of Lemma 5.4.

Lemma 5.5. *Let H, G, and $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be as in Lemma 5.4. An iso-triple (H_u^u, τ, w) of H is a v-characteristic for some $v \in V(\mathcal{T})$ if and only if there exists a $\theta \in \Theta_{vw}(\tau)$ and an injective function ψ from $\mathcal{N}(u)$ to the children of w in θ such that for all $u' \in \mathcal{N}(u)$ there is a subgraph isomorphism $\varphi_{u'}$ from $H_{u'}^u$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\theta\})_{\psi(u')}$ mapping u' to $\psi(u')$.*

Lemma 5.6 below is concerned with the correctness of Algorithm 5.1 deciding subtree isomorphism from a tree into an arbitrary text graph G. We assume without loss of generality that G is connected.

Lemma 5.6 (Correctness). *Algorithm 5.1 is correct, i.e., for all trees H and connected graphs G with $2 \leq |V(H)| \leq |V(G)|$, it returns* TRUE *if and only if $H \preccurlyeq G$.*

Proof. Algorithm 5.1 first fixes a root r of G (Line 2) and computes the complete guidance tree $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ of G, where $\mathcal{T}$ is rooted at r. By traversing the skeleton tree $\mathcal{T}$ in a postorder manner (Line 3), it calculates the set $\mathcal{C}$ of v-characteristics for all $v \in V(\mathcal{T})$ (Lines 4–6). We only need to show that $\mathcal{C}$ is correct (i.e., complete and sound); the correctness of the algorithm then follows directly from Line 7 using Lemma 5.3.

The completeness of $\mathcal{C}$ holds by the fact that all possible iso-triples $\xi = (H_u^y, \tau, w)$ relative to v are tested for being v-characteristics (Lines 3, 4, and 5 of MAIN together with Lines 3, 7, and 9 of CHARACTERISTICS). Thus, it remains to show that it is decided correctly whether or not $\xi = (H_u^y, \tau, w)$ is a v-characteristic. We prove this by double induction on the height $h_{\mathcal{T}}(v)$ of v in $\mathcal{T}$ and on the height $h_\tau(w)$ of w in τ. Depending on whether or not $h_\tau(w) = 0$ and $h_{\mathcal{T}}(v) = 0$, four cases can be distinguished. We only show the base case (i.e., $h_{\mathcal{T}}(v) = h_\tau(w) = 0$) and the most general case (i.e., $h_{\mathcal{T}}(v) > 0$ and $h_\tau(w) > 0$) by noting that the proofs of the other two cases can be shown by an argumentation similar to the one used for the most general case.

For the base case $h_{\mathcal{T}}(v) = h_\tau(w) = 0$ we have $C_\theta = \emptyset$ and hence $B = (\mathcal{N}(u), \emptyset)$ (Lines 5 and 6 of CHARACTERISTICS). Applying Lemma 5.4 to this case, ξ is a v-characteristic if and only if $\mathcal{N}(u) = \{y\}$, which, in turn, holds if and only if there is a matching covering $\mathcal{N}(u) \setminus \{y\}$ in B (Lines 10–11 of CHARACTERISTICS), as there are no edges in B.

If $h_{\mathcal{T}}(v) > 0$ and $h_\tau(w) > 0$ then $C_\tau \neq \emptyset$. Two cases can be distinguished: (i) If $w \notin V(\mathcal{T})$ then $C_{\tau'} = \emptyset$ and thus $C_\theta = C_\tau$. Applying Lemma 5.4 to this case, ξ is a v-characteristic if and only if there exists an injective function $\psi : \mathcal{N}(u) \setminus \{y\} \to C_\tau$ such that for all

$u' \in \mathcal{N}(u) \setminus \{y\}$, there exist a child c of w in τ (i.e., $c \in C_\tau$) and a subgraph isomorphism $\varphi_{u'}$ from $H^u_{u'}$ to $(G\{\mathcal{B}(v)/\tau\})_c$ with $\varphi_{u'}(u') = \psi(u') = c$ (i.e., a v-characteristic $(H^u_{u'}, \tau, c)$). By the induction hypothesis, the bipartite graph B is constructed correctly in Line 6 of Characteristics, and hence ψ exists if and only if there exists a matching in B covering $\mathcal{N}(u) \setminus \{y\}$. (ii) If $w \in V(\mathcal{T})$ then $C_\theta = C_\tau \cup C_{\tau'}$ with $C_\tau, C_{\tau'} \neq \varnothing$. Then, by Lemma 5.4, ξ is a v-characteristic if and only if for all $u' \in \mathcal{N}(u) \setminus \{y\}$ there exist a child c of w in θ and an injective function $\psi : \mathcal{N}(u) \setminus \{y\} \to C_\tau \cup C_{\tau'}$ such that there is a subgraph isomorphism $\varphi_{u'}$ from $H^u_{u'}$ to $(G\{\mathcal{B}(v) \cup \mathcal{B}(w)/\tau \cup \tau'\})_c$ with $\varphi_{u'}(u') = \psi(u') = c$. Such a subgraph isomorphism either corresponds to a v-characteristic $(H^u_{u'}, \tau, c)$ for $c \in C_\tau$, which has already been computed by the induction hypothesis on $h_\tau(w)$, or to a w-characteristic $(H^u_{u'}, \tau', c)$ for $c \in C_{\tau'}$, which has already been computed by the induction hypothesis on $h_{\mathcal{T}}(v)$. Hence ψ exists if and only if a matching in B (constructed in Line 6 of Characteristics) covering $\mathcal{N}(u) \setminus \{y\}$ exists (Lines 10–11 of Characteristics). The proof for the v-characteristics (H^u_u, τ, w) using Lemma 5.5 is analog for the check in Lines 7–8 of Characteristics. □

In Lemma 5.7 below we show that the runtime of Algorithm 5.1 is polynomial in $f(G)$ and the combined size of H and G, where $f(G) = \max_{\mathcal{S}_v \in \mathcal{S}} |\mathcal{S}_v|$ for some complete guidance tree $\mathbb{T}$. Together with Lemma 5.6 this implies Theorem 5.1 by noting that $f(G)$ is bounded by $f_{\max}(G)$.

Lemma 5.7 (Runtime). *Algorithm 5.1 runs in $O\left(f^2(G) \cdot |E(G)| \cdot |V(H)|^{1.5}\right)$ time.*

Proof. Note that the edge sets of the v-rooted components of G form a partition of $E(G)$, i.e.,

$$E(G) = \dot{\bigcup_{v \in V(\mathcal{T})}} E(\mathcal{B}(v)) \,. \tag{5.1}$$

This partition and the tree skeleton $\mathcal{T}$ can be computed in linear time (Tarjan, 1972). By definition, $|\mathcal{S}_v| \leq f(G)$ for all $v \in V(\mathcal{T})$. Thus, as the spanning trees of a graph can be generated with linear delay (Read and Tarjan, 1975), $\mathcal{S}_v$ can be computed in $O(|E(\mathcal{B}(v))| \cdot f(G))$ time for each $v \in V(\mathcal{T})$. Hence, by (5.1), Main spends altogether

$$O(|E(G)| \cdot f(G)) \tag{5.2}$$

time for computing the guidance tree $\mathbb{T}$. Furthermore, Main calls subroutine Characteristics $O(|V(G)| \cdot f(G))$ times. This is due to the fact that the number of pairs (v, w) (cf. Lines 3 and 5) is $O(|V(G)|)$, as each vertex w can occur in at most two sets of v-rooted components: In $\mathcal{B}(v)$ for its unique parent v in $\mathcal{T}$ (unless $w = r$) and in $\mathcal{B}(w)$ if w is a root itself. Regarding the complexity of Characteristics, note that $|\Theta_{vw}(\tau)|$ is bounded by $f(G)$ (see Line 2 of Characteristics) and that the bipartite graph B constructed in Line 6 has at most $|\mathcal{N}(u)| + |\mathcal{N}(w)|$ vertices for any $\theta \in \Theta_{vw}(\tau)$.

The edges of B can be constructed by membership queries to $\mathcal{C}$. We can implement the set $\mathcal{C}$ of characteristics found by the algorithm as a multidimensional array of polynomial size (in $f(G)$ and $|V(G)|$) such that each look-up and storage operation can be performed in constant time. A maximum matching of B can be found in $O\left(|\mathcal{N}(u)|^{1.5} \cdot |\mathcal{N}(w)|\right)$ time

(Hopcroft and Karp, 1973, Thm. 3). Applying the same trick as in (Chung, 1987; Shamir and Tsur, 1999) for ordinary subtree isomorphism, we can answer the matching queries for u and all of its neighbors in Line 7 and 10 of CHARACTERISTICS using a single bipartite matching computation and an additional operation that is linear in the size of B. Hence one iteration of CHARACTERISTICS runs in time $O\left(|\mathcal{N}(w)| \cdot |V(H)|^{1.5} \cdot f(G)\right)$, using the handshaking lemma for the tree H, i.e. $\sum_{u \in V(H)} |\mathcal{N}(u)| = 2 \cdot |E(H)| \in O\left(|V(H)|\right)$ and that $O\left(\sum_{u \in V(H)} \mathcal{N}(u)^{1.5}\right) \subseteq O\left(|V(H)|^{1.5}\right)$.

Thus, applying the handshaking lemma a second time for G, we obtain an overall time complexity $O\left(f(G)\left(|E(G)| + f(G) \cdot |E(G)| \cdot |V(H)|^{1.5}\right)\right)$ which, in turn, is equal to

$$O\left(f^2(G) \cdot |E(G)| \cdot |V(H)|^{1.5}\right) \,, \tag{5.3}$$

as claimed. □

Note that in the case that H and G are both trees, $f(G) = 1$ and hence (5.3) corresponds to the time complexity of the ordinary subtree isomorphism algorithms given in (Chung, 1987; Matula, 1968). We will address the implications of this algorithm for probabilistic and exact frequent tree mining in the next two sections.

5.2. Mining Boosted Probabilistic Frequent Subtrees

In Chapter 4 we introduced the FTM $_{Relaxed}$ problem and presented Algorithm 4.1 enumerating probabilistic frequent subtrees with polynomial delay. It is based on replacing each graph in the input with a forest formed by the vertex disjoint union of a random subset of its spanning trees. On the one hand, the more spanning trees are considered by the algorithm, the higher the recall of its output is. On the other hand, however, its delay depends linearly on the number of spanning trees, implying that in order to guarantee polynomial delay it can consider at most *polynomially* many spanning trees per graph. In this section we show that the results from Section 5.1 allow us to go beyond this limitation. In particular we now propose a *boosted* probabilistic frequent subtree mining algorithm for the FTM $_{Relaxed}$ problem that, using a variant of Algorithm 5.1, implicitly considers *exponentially* many spanning trees for the transaction graphs and still guarantees polynomial delay. In Section 5.2.2 we empirically compare its performance to that of the simple algorithm (Algorithm 4.1) from Chapter 4.

Recall that Algorithm 5.1 decides the SUBTREEISOMORPHISM problem by splitting the input transaction graph G into certain induced subgraphs and by considering the set of *all* local spanning trees for all such induced subgraphs. In case it takes *not* all, but only some subsets of the local spanning trees, its output becomes correct only with respect to the subset of global spanning trees of G that can be constructed by “gluing” together the local spanning trees considered in all possible ways. In Theorem 5.8 below we formulate a straightforward extension of Theorem 5.1 to this more general setting of the SUBTREE-ISOMORPHISM problem.

Algorithm 5.2 Subgraph Isomorphism from a Tree with One-Sided Error

```
Input  : tree $H$ with $|V(H)| > 1$ and guidance tree $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ for some connected graph $G$
         with $|V(G)| \geq |V(H)|$
Output: TRUE if $H \preccurlyeq \mathfrak{S}(\mathbb{T})$; o/w FALSE

MAIN:
1: set $\mathcal{C} := \emptyset$
2: for all $v \in V(\mathcal{T})$ in a postorder do
3:     for all $\tau \in \mathcal{S}_v$ do                                  // $\mathcal{S}_v \in \mathcal{S}$ is the bag of $v$ in $\mathbb{T}$
4:         for all $w \in V(\tau)$ in a postorder do
5:             $\mathcal{C} := \mathcal{C} \cup$ CHARACTERISTICS$(v, u, \tau, w)$
6:             if $(H_u^u, \tau, w) \in \mathcal{C}$ then return TRUE
7: return FALSE
```

To state this result, we need the following notion. Let $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be an arbitrary (i.e, not necessarily complete) guidance tree of G with bag $S_v \in \mathcal{S}$ for all $v \in V(\mathcal{T})$ and consider the graph T with $V(T) = V(G)$ and $E(T) = \bigcup_{v \in V(\mathcal{T})} E(\tau_v)$, where $\tau_v \in \mathcal{S}_v$ for all $v \in V(\mathcal{T})$. The definitions imply that T is a spanning tree of G. Hence, the disjoint union of all such spanning trees of G, i.e, which can be obtained by taking all possible combinations of the local spanning trees in the bags, forms a forest. We denote this forest by $\mathfrak{S}(\mathbb{T})$ (recall that the forest of spanning trees considered by the algorithms in Chapter 4 was denoted by $\mathfrak{S}_k(G)$). We are ready to formulate the following claim:

Theorem 5.8. *Let H be a tree, G be a graph, and $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ be a guidance tree of G. Then one can decide whether $H \preccurlyeq \mathfrak{S}(\mathbb{T})$ in time*

$$O\left(f'^2(G) \cdot |V(G)| \cdot |V(H)|^{1.5}\right) \quad ,$$

where $f'(G) = \max_{v \in V(\mathcal{T})} |\mathcal{S}_v|$.

Proof. Consider Algorithm 5.2 for the modified pseudo code of MAIN given in Algorithm 5.1, using the same subroutine CHARACTERISTICS. Its input includes $\mathbb{T} = (\mathcal{T}, \mathcal{S})$, instead of G. (Line 2 of MAIN in Algorithm 5.1 is accordingly removed.) The proofs of Lemma 5.6 and Lemma 5.7 immediately apply to the partial sets of local spanning trees as well, implying the correctness with respect to $\mathfrak{S}(\mathbb{T})$. Regarding the runtime, note that we can replace $|E(G)|$ in (5.3) by $|V(G)|$. Indeed, as $\mathbb{T}$ is given as input, Algorithm 5.2 does not have to consider G directly, as it has a direct access to the local spanning trees via $\mathbb{T}$, each having at most $|V(G)|$ edges. □

Note that Theorem 5.1 in the previous section is the restriction of Theorem 5.8 above to the special case that $\mathbb{T}$ is a *complete* guidance tree. Furthermore, Theorem 5.8 shows that Algorithm 5.2 solves the SUBTREEISOMORPHISM $_{Relaxed}$ problem of deciding subgraph isomorphism from trees into arbitrary graphs with one-sided error. That is, if Algorithm 5.2 returns "YES", then the answer is always correct; o/w it may happen that there exists a spanning tree T of G such that $H \preccurlyeq T$, but $H \not\preccurlyeq \mathfrak{S}(\mathbb{T})$. This property holds also for the algorithm in Chapter 4, which guarantees efficiency by explicitly considering a *polynomial* number of (random) *global* spanning trees of G. The importance of the result in Theorem 5.8 above is that it guarantees polynomial time even for the case that the number of *local* spanning trees in the bags of $\mathbb{T}$ is bounded by a *polynomial* of G which, in turn, may however implicitly represent *exponentially* many *global* spanning trees in $\mathfrak{S}(\mathbb{T})$ (cf. Section 5.3 for a straightforward example of this case). This result may be of some independent interest. Theorem 5.8 gives rise to the following positive result on efficient mining of frequent subtrees (without loss of generality, we formulate it for connected transaction graphs):

Theorem 5.9. *Let $\mathcal{D}$ be a finite set of connected graphs, $\mathbb{T}_G = (\mathcal{T}_G, \mathcal{S}_G)$ be a guidance tree of G for all $G \in \mathcal{D}$, and let $\mathcal{D}'$ be the set of forests defined by $\mathcal{D}' = \{\mathfrak{S}(\mathbb{T}_G) : G \in \mathcal{D}\}$. Then for any positive frequency threshold, the set of frequent subtrees of $\mathcal{D}'$ can be generated with delay polynomial in the combined size of the original dataset $\mathcal{D}$ and f', where f' is the maximum cardinality of the bags in $\mathbb{T}_G$ over all $G \in \mathcal{D}$.*

Proof. The proof follows directly from Theorem 4.2 together with Theorem 5.8. □

Clearly, for all positive frequency thresholds, any frequent subtree of $\mathcal{D}'$ is at the same time a frequent subtree of $\mathcal{D}$ as well. (The reverse direction does not hold for potential incompleteness.) For the particular case, which is in the focus of this section, that the bags in $\mathbb{T}_G$ are some *random* subsets of the corresponding sets of all local spanning trees, frequent subtrees of $\mathcal{D}'$ will be referred to as *probabilistic* frequent subtrees. We note that this definition is different from the one introduced in Chapter 4. Applying Theorem 5.9 to this case we have that probabilistic frequent subtrees can be listed with polynomial delay in the size of $\mathcal{D}$, whenever f' is bounded by a polynomial in the size of $\mathcal{D}$. We now discuss some algorithmic and implementation issues concerning the generation of such random bags.

The notions of boosted probabilistic frequent subtrees and probabilistic frequent subtrees are connected. Consider a guidance tree $\mathbb{T}$ where each bag contains k local spanning trees sampled independently and uniformly at random and fix some order on the spanning trees in each bag. Then, by "gluing" together all "first" local spanning tree in each bag, we obtain a first global spanning tree. Continuing in this way with all "second", "third", etc., we obtain k global spanning trees which are drawn independently and uniformly at random (from the set of global spanning trees of G). Let this set be called $\mathfrak{S}_k(G)$. Recall from Section 4.1.2 that this set can be interpreted as a forest and is identical to the concept with the same name considered in Chapter 4.

In the other direction, given a forest $\mathfrak{S}_k(G)$ of global spanning trees, we can construct a guidance tree $\mathbb{T} = (\mathcal{T}, \mathcal{S})$ by choosing a root and splitting the k global spanning trees into bags of k local spanning trees for each $v \in V(\mathcal{T})$. Hence, fixing either $\mathfrak{S}_k(G)$ or $\mathbb{T}$ and computing the other as above, we have $\mathfrak{S}_k(G) \preccurlyeq \mathfrak{S}(G)$ and the boosted algorithm considers at least the k global spanning trees in $\mathfrak{S}_k(G)$ (and possibly more, by considering other combinations of local spanning trees). As a result, for a given graph database and

Algorithm 5.3 Boosted Probabilistic Frequent Subtree Mining

```
input: a graph database $\mathcal{D} \subseteq \mathcal{G}$, frequency threshold $t > 0$ integer, and $k > 0$ integer
output: the set of frequent subtrees of $\mathcal{D}' = \{\mathfrak{S}(\mathbb{T}_G) \ : \ G \in \mathcal{D}\}$
1: $\mathbb{T}_\mathcal{D} := \emptyset$
2: for all $G \in \mathcal{D}$ do
3:     pick a root $r \in V(G)$ and compute a guidance tree $\mathbb{T}_G = (\mathcal{T}, \mathcal{S})$ s.t.
        $\mathcal{S}_v$ is a set of $k$ random spanning trees of $\mathcal{B}(v)$ for all $v \in V(\mathcal{T})$
4:     add $\mathbb{T}_G$ to $\mathbb{T}_\mathcal{D}$
5: list all subtrees with frequency at least $t$ in $\mathcal{D}' = \{\mathfrak{S}(\mathbb{T}_G) \ : \ \mathbb{T}_G \in \mathbb{T}_\mathcal{D}\}$
```

each set F of boosted probabilistic frequent subtrees, there exists a set $F' \subseteq F$ of probabilistic frequent subtrees. Further note that for this reason Theorem 4.4 holds for boosted probabilistic frequent subtrees as well: Considering a superset of global spanning trees can only increase the success probability of the embedding operator.

5.2.1. Implementation Issues

We now discuss some algorithmic and implementation issues concerning the generation of such random bags. Recall that Algorithm 4.1 simply samples k global spanning trees for each graph in the database. We apply the same idea *locally*, that is, we sample k local spanning trees for each bag of $\mathbb{T}_G$. Local spanning trees can be sampled in the same way as global spanning trees, given the v-rooted components: A spanning tree of such an induced subgraph B of G can be generated uniformly at random in *expected* time $O\left(|V(B)|^3\right)$ using the algorithm of Wilson (1996). We can again improve on this time and achieve a deterministic algorithm with $O\left(|E(B)| \cdot \log(|V(B)|)\right)$ runtime if the spanning trees are not required to be drawn uniformly at random (cf. Section 4.1.3).

Regarding the practical implementation of this algorithm, we note that sampling the spanning trees is actually never the dominating term. Following the idea of Theorem 5.9, instead of sampling local spanning trees anew for each invocation of the embedding operator, we select a root for all $G \in \mathcal{D}$ in a preprocessing step and consider the corresponding tree skeleton $\mathcal{T}$ of G. Algorithm 5.3 shows the pseudo code of this idea. For each $v \in V(\mathcal{T})$ we sample, with replacement, l spanning trees of the v-rooted components, where $l \in \mathbb{N}$ is some user specified parameter. In case of sampling identical local spanning trees for a block, we keep only one copy to speed up the algorithm. In particular, if all v-rooted components are bridges for some $v \in \mathcal{T}$, then the graph induced by the v-rooted components is a tree. In this case, we can safely just use this tree once, instead of sampling l identical spanning trees without changing the set of computed v-characteristics. We call such a root *trivial*.

The *global* spanning trees in $\mathfrak{S}(\mathbb{T})$ above, considered implicitly by our algorithm, are *random*. They are generated neither uniformly nor independently from the set of all spanning trees of G, even if we sample the local spanning trees uniformly and independently at random. This is due to the fact that any random local spanning tree picked for a nontrivial root contributes to at least two spanning trees in $\mathfrak{S}_G$, whenever G (with respect to the fixed root r) has at least two non-trivial roots. Our experimental results in Section 5.2.2 below however show that despite this kind of dependency, the recall increases by increasing values of l.

5.2.2. Experimental Evaluation

In this section we experimentally demonstrate the advantage of our algorithm mining probabilistic frequent subtrees by sampling local spanning trees over Algorithm 4.1 sampling global spanning trees. In what follows we will refer to the former technique as *boosted probabilistic subtree* (BPS) and to the latter one as *probabilistic subtree* (PS) mining. In particular, we show for different values of t that within time t, BPS considers a dramatically larger number of spanning trees per graph on average compared to PS, resulting in an improvement in terms of recall of frequent subtrees.

Our experiments clearly indicate that the amount of improvement strongly depends on the structural properties of the transaction graphs at hand. The improvement obtained for molecular graphs of small pharmacological compounds is negligible; we observed this consistently on several such benchmark graph datasets. As already mentioned, most *exact* frequent pattern mining algorithms have an excellent performance on this kind of graphs, with Gaston (Nijssen and Kok, 2005) being notably the fastest. However, all these exact methods seem to be limited to this particular graph class, as they were unable to produce any frequent patterns in feasible time, even for slightly more complicated structures beyond molecular graphs (cf. Section 4.2). In particular, for small neighborhood graphs extracted from social networks, none of the existing implementations were able to return any frequent patterns. In contrast, already PS could consistently produce an output having such a high recall (cf. Section 4.2.2) of frequent patterns that makes BPS unnecessary for this kind of graphs. This is due to the fact that such neighborhood graphs typically contain only one biconnected component and hence, BPS and PS behave similarly on them.

If, however, the transaction graphs have exponentially many spanning trees *and* several blocks at the same time, then PS is able to consider only a small fraction of all spanning trees, implying a negative impact on the recall. Such situations occur, for example, in case of *threshold graphs*, which are defined by local neighborhood relationships between objects in a metric space. Two vertices representing two objects are connected by an edge if and only if the distance of the corresponding objects is smaller than some given threshold (see Figure 5.3 for a threshold graph on 30 two-dimensional points). This kind of graphs have different practical applications, for example in spectral clustering (von Luxburg, 2007). While in that application field there are only rules of thumb on how to choose a suitable threshold for a particular metric and clustering task, one is interested in threshold graphs having a high edge density within each cluster and a low one among the clusters. This requirement typically results in threshold graphs having multiple biconnected blocks that are connected by a few bridges only and hence, in a large number of spanning trees. To demonstrate the advantage of BPS over PS, we have therefore considered threshold graphs in our experiments.

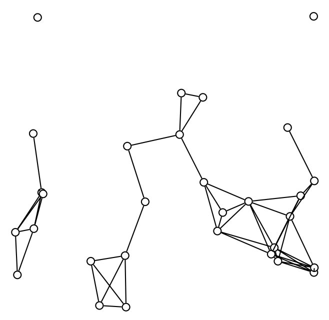


Figure 5.3.: A threshold graph on 30 points in the 2D Euclidean unit square for $d = 0.2$.

In particular, we evaluate our methods on artificial graph data sets that simulate the two-dimensional Brownian motion over time (see Section 2.4). To obtain the dataset, we generated $N = 200$ graphs with $n = 30$ vertices with $c \in \{2, 5, 10, 30\}$ random colors. We set $d = 0.2$ and $\mu = 0.02$, as the threshold graphs induced by these numbers fulfilled the desirable structural properties discussed above.

We first compare the average number of spanning trees and non-isomorphic spanning trees considered by the PS and BPS methods. As the resulting graphs may be disconnected (see, e.g., Figure 5.3), we extend our algorithms to this case as follows: We compute the number of *different*[5] spanning trees considered for each connected component separately, sum them up, and normalize the result by the number of connected components. To obtain the number of non-isomorphic spanning trees for each graph, we compute a canonical string for each tree in the above two sets, count the number of different strings, and again normalize by the number of connected components of the graph. Notice that the average number of non-isomorphic spanning trees calculated in this way can be smaller than one (e.g. when G has many singleton vertices with the same label).

Table 5.1 shows the average number of sampled spanning trees and that of non-isomorphic spanning trees for the threshold graph dataset defined above for BPS and PS. One can see that for all parameters $k \in \{2, 5, 10, 30\}$, both the average number of sampled spanning trees and the resulting non-isomorphic spanning trees is much larger for BPS. For example, for 30 labels and $k = 10$ we get on average only 4.51 different spanning trees and 4.06 non-isomorphic spanning trees for PS. On the other hand, BPS considers on average 2 606.08 different and 349.90 non-isomorphic spanning trees, when sampling $k = 10$ local spanning trees for each biconnected block. In order to obtain a similar number of non-isomorphic spanning trees on average with PS, one would need to sample at least 350 global spanning trees per graph.

An interesting observation is that the fraction of non-isomorphic spanning trees to different trees considered is rather different for PS and BPS. While for PS almost all sampled trees are non-isomorphic, this fraction drops to below 20% for BPS and larger values of k. We do not know whether this is because the overall number of non-isomorphic span-

[5] Here, two spanning trees T, T' are identical if and only if $E(T) = E(T')$. This is different from our usual notion that graphs are equal if and only if they are isomorphic.

	2 Labels		5 Labels		10 Labels		30 Labels	
k	PS	BPS	PS	BPS	PS	BPS	PS	BPS
2	1.44	4.89	1.44	4.98	1.43	4.81	1.44	4.74
	0.94	2.57	0.98	2.59	0.98	2.65	0.99	2.62
3	1.87	29.91	1.87	32.44	1.86	28.13	1.86	31.66
	1.33	7.89	1.40	9.15	1.41	8.63	1.41	9.68
4	2.27	118.78	2.27	102.48	2.27	112.12	2.26	100.03
	1.70	21.44	1.78	23.82	1.80	29.69	1.81	24.81
5	2.66	243.88	2.66	243.44	2.66	230.08	2.65	265.21
	2.04	38.58	2.16	44.67	2.19	53.97	2.20	53.69
6	3.04	510.26	3.03	465.82	3.04	490.62	3.05	498.13
	2.40	63.13	2.53	77.55	2.56	91.49	2.60	99.82
7	3.42	865.82	3.43	880.51	3.42	789.28	3.41	883.88
	2.73	99.16	2.91	117.07	2.93	129.55	2.96	141.80
8	3.79	1 364.81	3.77	1 382.15	3.80	1 306.57	3.77	1 231.39
	3.06	147.61	3.24	161.30	3.31	183.91	3.32	187.95
9	4.16	1 996.29	4.17	1 979.02	4.14	1 888.02	4.15	1 816.81
	3.38	200.06	3.62	251.41	3.65	257.12	3.70	277.92
10	4.51	2 717.93	4.51	2 744.65	4.51	2 868.81	4.51	2 606.08
	3.79	260.56	3.97	326.28	4.02	364.06	4.06	349.90

Table 5.1.: Average number of spanning trees considered by PS and BPS. For each number k of sampled global (resp. local) spanning trees for PS (resp. BPS) we report the average number of sampled spanning trees per connected component in the first row and the resulting average number of *non-isomorphic* spanning trees per connected component in the second row.

ning trees is rather small or because of the fact that the combination of local spanning trees results in many "similar" global spanning trees due to the dependency. We assume the latter by stressing that the average number of non-isomorphic spanning trees is still much larger than what can be achieved with a reasonable parameter k for PS.

Finally, we investigate the recall of frequent subtree patterns that can be obtained in a given time budget. That is, we fix a (low) frequency threshold $\theta = 2\%$ (corresponding to the absolute frequency threshold of 4) and mine (probabilistic) frequent subtrees on the threshold graph database for increasing values of k until the algorithm exceeds a runtime budget of 200 seconds. For a given value of k and for both methods PS and BPS, we repeat the mining algorithm ten times and average runtime and recall to mitigate for the effects of the random samples. Figure 5.4 shows the number of frequent patterns found (y-axis) per time (x-axis) for increasing values of the sampling parameter k. BPS obtains a significantly larger number of frequent patterns per time than PS for all time budgets up to 200 seconds[6]. For example, for 10 colors, we obtain on average 73 396 patterns in 195 seconds

[6] Note that according to this definition, the plots in Figure 5.4 can end before $x = 200$.

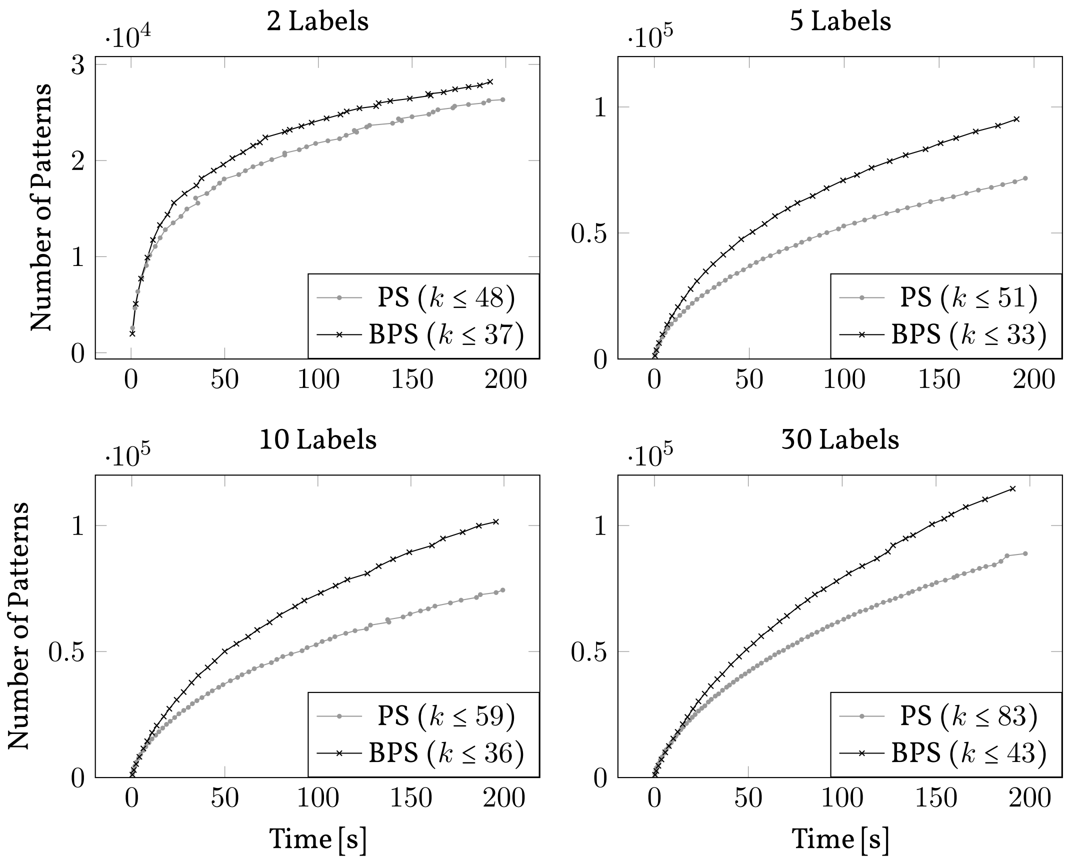


Figure 5.4.: Recall curves for 2%-frequent subtrees on the threshold graph database for PS and BPS for different numbers of vertex labels ranging from 2 to 30. Each dot corresponds to the average of 10 runs of the respective algorithms for some given value of k, ranging from 1 to the number indicated in the legends.

for $k = 59$ using PS and 101 503 patterns for $k = 36$ for BPS, a 38.3% increase. Comparing the runtimes necessary to obtain a given amount of frequent patterns, this difference gets even more concrete. To obtain at least the same number of frequent patterns returned by PS in at most 200 seconds, BPS needs only 148.77s, 106.74s, 109.52s, and 124.38s, for 2, 5, 10, and 30 vertex labels, respectively. Thus, on transaction graphs consisting of several dense biconnected components, such as, for example, threshold graphs, BPS clearly has a superior performance over PS.

5.3. Exact Frequent Subtree Mining on Locally Easy Graphs

Recall that frequent subtrees can be mined efficiently in forest databases or if the number of spanning trees in each transaction graph G is polynomial in the size of G. Such graphs will be referred to as *easy* graphs. We will now formally define this graph class and investigate some of its properties. Recall that frequent subtrees can be mined efficiently in forest databases, or more generally, in graphs having polynomially many spanning trees; this follows from the results e.g. in (Chi et al, 2005; Horváth and Ramon, 2010). Such graphs will be referred to as *easy* graphs. Except for forests, the class of easy graphs is typically uninteresting from a practical viewpoint, as even for relatively simple graphs beyond forests, the number of spanning trees usually grows *exponentially* with the number of vertices. Our positive result extends to this practically and theoretically more interesting situation by requiring easiness *not* for the entire graph, but only for *local* surroundings of the vertices. Formally, a graph G is *locally easy* if for all $v \in V(G)$, the number of spanning trees of the union formed by the biconnected components of G containing v is bounded by a polynomial of $|V(G)|$, i.e., $f_{max}(G) = O\left(\text{poly}(|V(G)|)\right)$ (cf. Section 5.1). In particular, for the case that it is bounded by $p(|V(G)|)$ for some polynomial p (resp. by $|V(G)|$) we will speak of *locally p-easy* (resp. *locally linearly easy*) graphs. Clearly, all easy graphs are locally easy, but a locally easy graph may contain exponentially many spanning trees (see Figure 5.5 for an example). We have the following result:

Theorem 5.10. *The FTM problem can be solved with polynomial delay for locally easy transaction graphs.*

Proof. By Theorem 2.1, Algorithm 2.1 solves the FTM problem for locally easy transaction graphs with polynomial delay whenever all conditions required are fulfilled. Conditions 1 and 2 of Theorem 2.1 are straightforward when $\mathcal{P}$ is restricted to the class of trees and Condition 3 follows e.g. from (Shamir and Tsur, 1999). Finally, Theorem 5.1 immediately implies Condition 4 for tree patterns and locally easy transaction graphs. □

Below we discuss some important properties of locally easy graphs implying the theoretical and practical importance of Theorem 5.10 above.

Property 1

The *membership* problem for locally easy graphs (i.e., whether a graph is locally easy or not) can be decided in cubic time, implying that it can be checked in polynomial time for any graph database D whether or not Theorem 5.10 is applicable to D. More precisely, let G be a graph and p be some polynomial. One can decide in cubic time whether G is locally p-easy by performing the following steps: (i) Compute first the set of all biconnected components of G, (ii) calculate the number of spanning trees for all blocks of G, and (iii) check for all $v \in V(G)$ whether the product of these values for all biconnected components sharing v is at most $p(|V(G)|)$. The claim above then follows by noting that (i) can be solved in linear (Tarjan, 1972) and (ii) in cubic time using Kirchhoff's theorem (see, e.g., Chap. 5.6 in Stanley and Fomin, 1999).

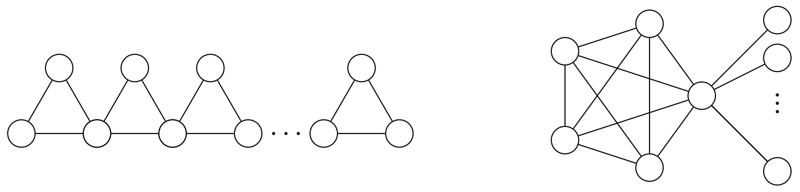


Figure 5.5.: A locally easy graph with exponentially many spanning trees on the left and a locally linearly easy graph of tree-width 4 on the right.

Property 2

Locally easy graphs may contain *exponentially* many spanning trees. As an example, consider the graph G given in the left-hand side of Figure 5.5. It is locally linearly easy (for all $v \in V(G)$ there are at most 9 spanning trees in the union of the blocks containing v), still it has altogether $3^{O(|V(G)|)}$ spanning trees. This and other examples show that our result formulated in Theorem 5.10 is non-trivial, as the brute-force pattern matching algorithm that decides whether a tree is subgraph isomorphic to a locally easy graph G by testing subtree isomorphism for *all* spanning trees of G becomes infeasible for such cases.

Property 3

The class of locally easy graphs contains some interesting graph classes for which the FTM problem is computationally tractable. As an example, we mention the class of almost k-trees of bounded degree; a graph G is an *almost k-tree* for some $k \geq 0$ integer, if $|E(B)| \leq |V(B)| + k$ for all blocks B of G. One can decide in polynomial time whether a tree is subgraph isomorphic to an almost k-trees of bounded degree (Akutsu, 1993). Combining this result with Theorem 2.1 we have that the FTM problem can be solved with polynomial delay for almost k-trees of bounded degree. We can obtain this result directly by Theorem 5.10 as well because the class of locally easy graphs properly contains that of almost k-trees of bounded degree. The strength of Theorem 5.10 is that it generalizes the positive mining result above also to almost k-trees of *unbounded* degree that are locally easy.

Property 4

The class of locally easy graphs is *orthogonal* to all monotone parameterized graph classes. More precisely, a *parameterized graph class $\mathcal{G}$* is a family of graph classes $\{\mathcal{G}_i : i \geq 0\}$ such that (i) for every graph G there exists a non-negative integer i with $G \in \mathcal{G}_i$ and (ii) $\mathcal{G}_j \subsetneq \mathcal{G}_{j+1}$ for all $j \geq 0$. The smallest integer i satisfying $G \in \mathcal{G}_i$ is denoted by $I_{\mathcal{G}}(G)$. A *parametrized graph class $\mathcal{G}$* is *monotone* if $I_{\mathcal{G}}(G_1) \leq I_{\mathcal{G}}(G_2)$ whenever $G_1 \preccurlyeq G_2$, for all graphs G_1, G_2. The size, order, maximum vertex degree, tree-width, number of spanning trees of a graph are some straightforward examples of $I_{\mathcal{G}}$ defining monotone parameterized graph classes. An example for some $I_{\mathcal{G}}$ resulting in a non-monotone graph class would be the number of connected or biconnected components. Using the above concepts, we are ready to formulate the following claim:

Claim 5.11. *For any monotone parameterized graph class $\mathcal{G} = \{\mathcal{G}_i : i \geq 0\}$ and for any integer $k \geq 0$ there are infinitely many locally linearly easy graphs that are not in $\mathcal{G}_k$.*

Proof. Let $\mathcal{G}$ be a monotone parametrized graph class and let $G_1, G_2, G_3, \ldots$ be a sequence of graphs with $I_{\mathcal{G}}(G_i) = i$. Such a sequence exists by definition. For any $i \in \mathbb{N}$, adding new leafs to G_i does not decrease $I_{\mathcal{G}}(G_i)$, as $\mathcal{G}$ is monotone. However, it will eventually decrease the local easiness of the resulting graph: For all $i \geq 0$, let $m(G_i)$ be the maximum number of spanning trees in the union of the biconnected components of G_i containing v for any $v \in V(G_i)$. By adding $\max(0, m(G_i) - |V(G_i)|)$ new leafs (i.e., vertices of degree 1) to G_i in an arbitrary way, we obtain a locally linearly easy graph G'_i with $I_{\mathcal{G}}(G'_i) \geq i$ and $m(G'_i) = m(G_i)$ for all $i \geq 0$. Thus, for any fixed $k \in \mathbb{N}$, $G'_{k+j} \notin \mathcal{G}_k$ for all $j \geq 1$, implying the claim. □

We illustrate the idea in the proof above on the monotone parametrized graph class induced by the tree-width (see, e.g., Diestel, 2012). Consider the graph G on the right-hand side of Figure 5.5. It is obtained from the complete graph K_k on k vertices for some $k \geq 3$ by adding $k^{k-2} - k$ leafs to some vertex of K_k. On the one hand, the construction does not increase the tree-width, i.e., the tree-width of G is equal to that of K_k. On the other hand, as K_k has exactly k^{k-2} spanning trees by Cayley's formula, G is a locally linearly easy graph. Since the construction in this example holds for any $k \geq 3$, local easiness implies no constant upper bound on the tree-width.

The choice of tree-width in the example above is especially interesting because frequent subtrees of *bounded degree* can be generated with polynomial delay from graphs of bounded tree-width. This follows from Theorem 2.1 together with the positive result of Matoušek and Thomas (1992) on subgraph isomorphism between bounded tree-width graphs. This and other examples provide evidence that our main result formulated in Theorem 5.10 extends (or complements) several results on the (fixed parameter) tractability of the FTM problem for various monotone parametrized graph classes for which subgraph isomorphism from a tree can be decided in polynomial time. We note, for example, that in the systematic overview of the parameterized complexity of subgraph isomorphism by Marx and Pilipczuk (2014), 9 out of the 10 parameters considered result in monotone graph classes. Hence, our result extends the positive results in their work to the case that the patterns are restricted to trees.

Property 5

Most of the molecular graphs considered in chemoinformatics are actually locally easy. To confirm this observation, we first provide a sufficient condition for local easiness. Let G be a graph and let $c, k \geq 0$ be integers. Then G is *degree-k easy* if each block of G has at most $O(|V(G)|^k)$ spanning trees and it is of ***block degree-c*** if each vertex v of G belongs to at most c distinct blocks.[7] Clearly, if G is degree-k easy and of block degree-c for some constants k and c, then G is locally easy.

Many of the chemical graphs of pharmacological compounds are d-tenuous outerplanar graphs for $d \leq 5$ (Horváth et al, 2010). Informally, each block of such a graph is a planar graph composed of a single Hamiltonian cycle and at most d non-crossing diagonals. Clearly, d-tenuous outerplanar graphs are degree-$(d+1)$ easy. Furthermore, chemical graphs have typically some very small block degree because they have small vertex degree. Thus, most chemical graphs are locally easy. To support this claim experimentally, we investigated local easiness for the graphs in the ZINC dataset[8]. Our version of the database contains 8 946 757 "lead-like" compounds. It turns out that for 8 640 166 (96.57%) graphs, the maximum number of spanning trees in all blocks containing any vertex (i.e., $m(G)$) is smaller than $|V(G)|$, for 302 541 (3.38%) it is smaller than $|V(G)|^2$, for 1 864 (0.02%) it is smaller than $|V(G)|^3$, and only for the remaining 2 186 (0.02%) graphs this number was larger than $|V(G)|^3$. Thus, by our result in Theorem 5.10, all frequent trees can be generated from such chemical graphs with polynomial delay. This complements the positive result of Horváth et al (2010) on mining frequent connected subgraphs from d-tenuous outerplanar graphs with respect to a constrained subgraph isomorphism operator.

5.4. Summary and Open Questions

In this chapter we have extended our probabilistic frequent subtree mining framework to increase the pattern recall. This was achieved by a novel embedding operator that is able to implicitly consider an exponential number of spanning trees in polynomial time. The resulting set of boosted probabilistic frequent subtrees of a graph database is at least as large as a corresponding set of (normal) probabilistic frequent subtrees while giving the same guarantees. While the amount of improvement in recall per runtime is impressive for threshold graphs and for other potential graph classes satisfying the structural properties discussed in Section 5.2, it is marginal e.g. for chemical or small neighborhood graphs extracted from social networks. This raises the practical question whether we can devise a fast practical method to decide for a given graph database (possibly for each graph separately), whether probabilistic frequent subtrees should be mined sampling global (Chapter 4), or local (Chapter 5) spanning trees.

As a second achievement we have proposed a polynomial delay FTM algorithm for graph databases consisting of locally easy graphs. This extends on previously known results w.r.t. the complexity of the FTM problem, as well as the SUBTREEISOMORPHISM problem. Finding non-trivial transaction classes where the FTM or FCSM problems have efficient solutions is an important challenge for graph mining, not only from theoretical, but also from practical aspects.

[7] Note that the vertex degree is an upper bound on the block degree.
[8] Obtained from http://zinc.docking.org

It would be interesting to understand how far the positive result of this chapter on exact frequent subtree mining can be generalized to other pattern classes beyond trees. Perhaps the first natural question towards this direction would be to ask whether it is possible to generate frequent *locally easy* subgraphs (in arbitrary transaction graphs) with polynomial delay. In order to calculate the v-characteristics for a root vertex v with respect to a vertex u in the pattern, our algorithm combines at most two sets of spanning trees at any time and assumes that neither u nor the vertices in its local environment are contained in a cycle. Therefore, in order to apply the algorithm to the more general patterns of locally easy graphs, we need to work with the spanning trees of certain local environments of u. However, in contrast to the transaction graphs, it may happen that such spanning trees are composed of the combination of the spanning trees of the blocks for a *non-constant* number of root vertices of the pattern graph. In such a case, an exponential number of spanning trees must be processed. This indicates that, if it is possible at all, such a generalization would require some more sophisticated approach.

Finally we give arguments clearly indicating the significance of generalizing the positive result in Theorem 5.10 to transaction graphs beyond locally easy graphs. We suspect that obtaining such a generalization is at least as hard as solving the millennium problem **P** versus **NP**. In particular, it is natural ask to whether frequent subtrees can be generated with *polynomial delay* also from transaction graphs for which we only require the number of spanning trees *per block* to be bounded by a polynomial in the size of the whole graph (i.e., we do not assume any constant upper bound on the block degree). In contrast to locally easy graphs, subgraph isomorphism from trees into this type of more general graphs becomes NP-complete, even for the very simple class of *cactus* graphs (i.e., in which each block is a simple cycle, Akutsu, 1993). We do not know the answer to the question above, not even to the case of cactus transaction graphs, as discussed in Section 2.2.2. That is, even if the blocks are restricted to cycles, subgraph isomorphism becomes **NP**-complete if the number of local spanning trees is not bounded by a polynomial, or, more generally, the block degree is not constant. This shows, that the exact algorithms discussed in this chapter is on the border between tractability and intractability of both the FTM and SubtreeIsomorphism problems.

6. Fast Computation in Probabilistic Subtree Feature Spaces

In the previous two chapters we have devised efficient methods to find probabilistic frequent subtrees in arbitrary graph databases. While these patterns may be of interest by themselves in some applications (Borgelt and Berthold, 2002), frequent subgraph mining usually tends to be only the first tool in a chain of processing steps from data to knowledge. That is, one common usage of frequent subgraphs is to use them as *features* to represent graphs in a feature space, equipped with some metric to employ distance-based learning methods. We have already seen such examples in Sections 1.1 and 4.2, where we have trained a support vector machine on the feature representations spanned by probabilistic frequent subtrees. We will propose practically fast and theoretically efficient methods to compute such a feature vector $\vec{f}$ for any graph G, given a set of (probabilistic frequent) tree patterns $\mathcal{F}$.

Of course, the generic levelwise algorithm presented in Section 2.2 can be easily extended to output not only the frequent patterns, but also their support sets, in a given graph database. This, however, does not suffice in most realistic scenarios: When training a model, we would like to apply it later to make predictions for novel data. That is, *given* an unseen graph G (usually drawn from the same distribution as the training dataset), we need to *compute* its embedding into the Hamming space spanned by frequent (probabilistic) subtrees. In the context of chemoinformatics, for example, one might want to predict whether a newly discovered or previously not synthesizable molecule is expected to be active against a certain disease based on a model learned on one of the datasets presented in Section 2.4. Another example might be traces of the browsing behavior of users that arrive as a stream and should be targeted by different ads. In these settings some (or most) graphs are not available at the time of mining and we hence need an efficient way to compute feature representations of such graphs in order to apply a model to them.

However, the embedding step is mostly neglected in the frequent subgraph mining community. Most papers in the graph mining context ignore this obvious second step and focus on the enumeration of frequent subgraph patterns only. The lack of interest might be explained by the existence of two immediate solutions, namely

Brute Force: We can compute the embedding $\vec{f}$ by evaluating the embedding operator for *all* patterns in $\mathcal{F}$ and the novel graph G.

Restricted FCSM: The embedding computation problem is a special case of the FCSM problem for a finite class of tree patterns $\mathcal{F}$ (namely the frequent trees identified in the previous mining step) and a graph database consisting of the single graph G.[1] $\vec{f}$ is the incidence vector of the 1-frequent elements of $\mathcal{F}$ in the database $\{G\}$.

Both views might have lead the authors in the graph mining community to believe the problem to be uninteresting; it is easy to code up both solutions, once the machinery for a frequent subgraph mining algorithm has been implemented. However, both approaches suffer from the necessity to evaluate the embedding operator on a large number of patterns. As we will show in this section, impressive practical speedups can be gained by using some additional structure on the pattern set and the fact that it is explicitly given.

Recall from Section 2.1 that the SubtreeIsomorphism problem is **NP**-complete. Hence, both formulations above are computationally intractable. If we give up the demand on the correctness of the pattern matching operator (i.e., subgraph isomorphism), as in the previous chapter, we can compute the set of all $H \in \mathcal{F}$ that are found to be subgraph isomorphic by one of our embedding operators in polynomial time using both the restricted FCSM and the brute force approach. However, each call to the embedding operator still induces a nontrivial cost and hence we can drastically improve the speed of the embedding computation by reducing the number of calls to the embedding operator. Subgraph isomorphism induces a partial order on the pattern set in which it is anti-monotone. Therefore 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 empirically show that both algorithms drastically reduce the number of embedding operator evaluations compared to the mentioned two immediate solutions.

The high dimensionality of the resulting feature space often results in practically infeasible time and space complexity for distance-based learning methods. Time and space requirements can, however, be significantly reduced by using *min-hashing* (Broder, 1997), an elegant and powerful probabilistic technique for the approximation of the Jaccard-similarity. Given a binary feature vector $\vec{f}$ and a permutation π of $\vec{f}$, the method is based on calculating the min-hash value $h_\pi(\vec{f})$, which is the position of the first pattern H matching G, i.e. the position of the first occurrence of a one in the permuted order of $\vec{f}$. 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 for a feature vector $\vec{f}$ and permutation π, $h_\pi(\vec{f})$ can be computed *without* calculating $\vec{f}$. We utilize the fact that we are interested in the first occurrence of a one in the order corresponding to π; once we have found it, we can stop the calculation, as all patterns after $h_\pi(\vec{f})$ are irrelevant for min-hashing. Beside this straightforward speed-up of the algorithm, the computation of the min-hash sketch can further be accelerated

[1] Using Algorithm 2.1, we can check efficiently whether a generated pattern is contained in the finite set $\mathcal{F}$ using a canonical string function and a trie, as described in Section 2.1. However, more sophisticated implementations of this check are possible, as discussed later in this chapter.

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 6.3 clearly demonstrate that our techniques can dramatically reduce the number of subtree isomorphism tests with respect to the algorithm performing the embedding by first explicitly computing $\vec{f}$ and then applying min-hashing. It is natural to ask how the predictive performance of the approximate similarities compares to the exact ones. We show that even for a few random spanning trees per chemical compound and a small memory requirements of the min-hash sketch, remarkable precisions of the active molecules can be obtained by taking the k nearest neighbors of an active compound for $k = 1, \dots, 100$. The 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.

Outline

The rest of this chapter is organized as follows: In Section 6.1 we first discuss how to compute complete embeddings. Next, in Section 6.2, we move on to partial embeddings for min-hashing in subtree feature spaces. We report experimental results evaluating the practical efficiency of our algorithms in Section 6.3 and also address the predictive performance of the min-hash based approximate distance function. Finally, in Section 6.4 we conclude and mention some interesting directions for further research.

6.1. Complete Embeddings into Subtree Feature Spaces

In this section we deal with the problem of computing *complete* embeddings into Hamming feature spaces spanned by a given set of 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:

TREE EMBEDDING COMPUTATION (TEC) PROBLEM: *Given* a set $\mathcal{F}$ of trees and a graph G, *compute* the incidence vector $\vec{f}$ 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 $F = (\mathcal{F}, E)$ with $(T_1, T_2) \in E$ if and only if $T_1, T_2 \in \mathcal{F}$, $|V(T_2)| = |V(T_1)| + 1$, and $T_1 \preccurlyeq T_2$.

Clearly, the TEC problem is NP-complete. Therefore, we relax it in a way similar to the relaxation of the FTM problem to the FTM $_{Relaxed}$ problem in Chapter 4 and approximate the incidence vector $\vec{f}$ of $\{T \in \mathcal{F} : T \preccurlyeq G\}$ by the incidence vector $\vec{f}'$ of $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}(G)\}$. Here, we allow the forest $\mathfrak{S}(G)$ to be either $\mathfrak{S}_k(G)$ for some $k \in \mathbb{N}$ using the techniques from Chapter 4 or $\mathfrak{S}(\mathbb{T})$ for some guidance tree $\mathbb{T}$ with bag size k using the techniques from Chapter 5. While this relaxation makes the TEC problem computationally tractable, each invocation of the probabilistic matching operator adds a non-negligible amount of work, i.e., $O\left(kn^{2.5}/\log n\right)$ time for computing subgraph isomorphism from trees into forests (Shamir and Tsur, 1999), respectively $O\left(k^2n^{2.5}/\log n\right)$ time, corresponding to the runtime of Algorithm 5.2. This super-quadratic and in practice notable complexity motivates us to minimize the number of calls of the probabilistic matching operator while computing $\vec{f}'$. We present three algorithms for computing $\vec{f}'$ that significantly reduce the number of probabilistic pattern matching evaluations in practice compared to the brute-force algorithm calling the embedding 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 $\vec{f}'$ for G is equivalent[2] 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 from now on, 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 their subgraphs in $\mathcal{F}$ do.

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 6.3. 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 6.1. 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. It encodes the value of the embedding operator on a pattern in a ternary $state$ variable,

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

Algorithm 6.1 Greedy

Input: A graph G and a directed graph $F = (\mathcal{F}, E)$ representing the poset $(\mathcal{F}, \preccurlyeq)$

Output: $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}(G)\}$

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

6: **procedure** DFS($\mathfrak{S}(G), T, state, F$)
7:     **if** $state[T] = unknown$ **then**
8:         **if** $T \preccurlyeq \mathfrak{S}(G)$ **then**
9:             **for** $T'$ with $(T, T') \in E$ **do** DFS($\mathfrak{S}(G), T', state, F$)
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 $F$.
```

which can take the values *match*, *noMatch*, and *unknown*. While Greedy is running, it updates the state of patterns according to the anti-monotonicity of subgraph isomorphism. 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 6.2. 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

Algorithm 6.2 BinarySearch

Input: A graph G and a directed graph $F = (\mathcal{F}, E)$ representing the poset $(\mathcal{F}, \preccurlyeq)$

Output: $\{T \in \mathcal{F} : T \preccurlyeq \mathfrak{S}(G)\}$

```
1:  set $state[T] := unknown$ for all $T \in \mathcal{F}$
2:  fix $\mathfrak{S}(G)$
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($\mathfrak{S}(G), P, state, F$)
7:  return $\{T \in \mathcal{F} : state[T] = match\}$

8:  procedure BinarySearch($\mathfrak{S}(G), P, state, F$)
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 $T \preccurlyeq \mathfrak{S}(G)$ 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 deducible state of 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 6.1 above, one can show that Algorithm 6.2 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 6.2. During our empirical evaluation, however, we saw that the runtime of BinarySearch was still dominated by the calls to the matching operator.

6.2. 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., Gärtner et al, 2003; Teixeira et al, 2012). 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:

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

$$\textsc{Sim}_{\text{Jaccard}}(\vec{f_1}, \vec{f_2}) \ ,$$

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

Instead of using the naive brute-force algorithm, i.e., performing first the explicit embeddings of $\mathfrak{S}(G_1)$ and $\mathfrak{S}(G_2)$ into the Hamming space spanned by $\mathcal{F}$ and calculating then the exact value of $\textsc{Sim}_{\text{Jaccard}}(\vec{f_1}, \vec{f_2})$, we follow Broder's probabilistic *min-hashing* technique (Broder, 1997) sketched in Section 2.3. 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 $\mathfrak{S}(G)$ into the Hamming space spanned by $\mathcal{F}$ would require $|\mathcal{F}|$ calls to the embedding operator, each of which induces superquadratic cost in the worst case. 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 $\mathcal{F}$, as we have seen in the previous section. 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 $\mathcal{F}$ and fix them for computing the min-hash values that will be used for similarity query evaluations (cf. Section 2.3). 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 $\pi_1, \dots, \pi_K$, and do not apply any implicit representations of them.[3] This is particularly true, as we compute $\mathcal{F}$ explicitly in the preprocessing step and invest time that is polynomial in $\mathcal{F}$ anyway.

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

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

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

$$\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G) = (h_{\pi_1}(G), \dots, h_{\pi_K}(G)) \quad .^4$$

The rest of this section is devoted to the following problem: Given $\pi_1, \dots, \pi_K$ and a graph G with forest $\mathfrak{S}(G)$ as above, compute $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$. The first observation that leads to an improved algorithm computing $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$ is that for any $i \in [K]$, the set $\mathcal{F}$ 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 $\mathfrak{S}(G)$ we have either

1. $T_1 \preccurlyeq \mathfrak{S}(G)$ and hence T_2 is not the first matching pattern in π_i or
2. $T_1 \not\preccurlyeq \mathfrak{S}(G)$ and hence $T_2 \not\preccurlyeq \mathfrak{S}(G)$ 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 6.3 implements this idea for a 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 6.1. *Let $\sigma = \langle T_1, \dots, T_l \rangle$ be the output of Algorithm 6.3 for a permutation π of $\mathcal{F}$. Then, for any graph G with $\mathfrak{S}(G)$,*

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

Proof. Let $H = h_\pi(G)$, i.e., $H = \operatorname{argmin}_{T \in \mathcal{F}} \{\pi(T) : T \preccurlyeq \mathfrak{S}(G)\}$ and let $H' = \operatorname{argmin}_{T_i \in \sigma}\{i : T_i \preccurlyeq \mathfrak{S}(G)\}$. The output of Algorithm 6.3 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, $\pi(H) \le \pi(H')$. 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, $H \preccurlyeq \mathfrak{S}(G)$ implies $T \preccurlyeq \mathfrak{S}(G)$, contradicting our assumption $H = h_\pi(G)$. □

Algorithm 6.3 runs in time $O(|\mathcal{F}|)$. The loop starting in Line 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.

[3] See (Broder et al, 2000) for the details of implicitly representing permutations via min-wise independent hash functions.

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

Algorithm 6.3 PosetPermutationShrink

Input: directed graph $F = (\mathcal{F}, E)$ representing a poset $(\mathcal{F}, \preccurlyeq)$ and permutation π of $\mathcal{F}$

Output: evaluation sequence $\sigma = \langle T_1, \dots, T_l \rangle \in \mathcal{F}^l$ for some $0 < l \le |\mathcal{F}|$ with $\pi(T_i) < \pi(T_j)$ for all $1 \le i < j \le l$

```
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 $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$ for a graph G with $\mathfrak{S}(G)$. A straightforward implementation of calculating $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$ for the evaluation sequences $\sigma_1, \dots, \sigma_K$ computed by Algorithm 6.3 for $\pi_1, \dots, \pi_K$ 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 $\mathfrak{S}(G)$, if T occurs in more than one permutation before or as the first match. Lemma 6.2 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 6.2. *Let G be a graph with $\mathfrak{S}(G)$ and let $\sigma_1, \dots, \sigma_K$ be the evaluation sequences computed by Algorithm 6.3 for the permutations $\pi_1, \dots, \pi_K$ of $\mathcal{F}$. Let $\mathfrak{A}$ be an algorithm that correctly computes $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$ by evaluating subgraph isomorphism in the pattern sequence $\Sigma = \langle \sigma_1[1], \dots, \sigma_K[1], \sigma_1[2], \dots, \sigma_K[2], \dots \rangle$. Then $\mathfrak{A}$ remains correct if for all $i \in [K]$ and $j \in [|\sigma_i|]$, $\mathfrak{A}$ skips the evaluation of $\sigma_i[j] \preccurlyeq \mathfrak{S}(G)$ whenever one of the following conditions hold:*

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

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 $T \preccurlyeq \mathfrak{S}(G)$ then $\sigma_i[j] \preccurlyeq \mathfrak{S}(G)$ by the transitivity of subgraph isomorphism. For the same reason, if $T \preccurlyeq \sigma_i[j]$ and $T \not\preccurlyeq \mathfrak{S}(G)$ then $\sigma_i[j] \not\preccurlyeq \mathfrak{S}(G)$. 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 6.4 MIN-HASH SKETCH

Input: graph G with forest $\mathfrak{S}(G)$, directed graph $F = (\mathcal{F}, E)$ representing a poset $(\mathcal{F}, \preccurlyeq)$ and K evaluation sequences $\sigma_1, \dots, \sigma_K$ computed by Algorithm 6.3 for the permutations $\pi_1, \dots, \pi_K$ of $\mathcal{F}$

Output: $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$

```
1:  Initialize $sketch := [\bot, \dots, \bot]$
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 $|\sigma_j| \geq i \wedge sketch[j] = \bot$ 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 $\sigma_j[i] \preccurlyeq \mathfrak{S}(G)$ 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 $sketch$
```

Algorithm 6.4 computes the sketch for a graph G with $\mathfrak{S}(G)$ along the conditions formulated in Lemma 6.2. Similarly to the algorithms in Section 6.1 it maintains a state for all $T \in \mathcal{F}$ defined as follows: $unknown$ encodes that $T \preccurlyeq G$ is unknown, $match$ that $T \preccurlyeq \mathfrak{S}(G)$, and $noMatch$ that $T \not\preccurlyeq \mathfrak{S}(G)$.

Theorem 6.3. *Algorithm 6.4 is correct, i.e., it returns* $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G)$. *Furthermore, it is non-redundant, i.e., for all patterns* $T \in \mathcal{F}$, *it evaluates at most once whether or not* $T \preccurlyeq \mathfrak{S}(G)$.

Proof. The correctness of Algorithm 6.4 is immediate from Lemmas 6.1 and 6.2. Regarding its non-redundancy, suppose $T \preccurlyeq \mathfrak{S}(G)$ has already been evaluated for some pattern $T = \sigma_i[j]$. 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 $T \preccurlyeq \mathfrak{S}(G)$ 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 $\textsc{Sketch}_{\pi_1,\dots,\pi_K}(G_1) = \textsc{Sketch}_{\pi_1,\dots,\pi_K}(G_2) = (\bot, \dots, \bot)$ by 0.

6.3. Experimental Evaluation

We have conducted various experiments on different real-world and artificial datasets to evaluate the methods described in the previous section. We evaluate their *speed* measured by the number of subtree isomorphism tests performed in Section 6.3.1 and show that our

Dataset	k	θ	$\vert\mathcal{F}\vert$	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 6.1.: Average number of subtree isomorphism tests per graph of the algorithms from Section 6.1 on different datasets and corresponding pattern sets $\mathcal{F}$ for varying number k of random spanning trees and frequency thresholds θ.

methods drastically reduce this number compared to the brute-force and the levelwise baseline algorithms discussed in Section 6.1. In Section 6.3.2 we finally evaluate the *predictive* and *retrieval* performance of probabilistic frequent subtrees applied combined with min-hashing in distance-based methods.

6.3.1. Efficiency Gains

We now empirically investigate the *speedup* of the methods proposed in Section 6.1 for computing complete and partial embeddings into probabilistic frequent subtree (PFS) feature spaces. (We recall that the methods in Section 6.1 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 aspect by investigating the average number of subtree isomorphism evaluations (i.e., deciding whether $T \preccurlyeq \mathfrak{S}(G)$) per graph on various real-world datasets.

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 Chapter 4 with different values of k and θ to a randomly sampled subset of 10% of the graphs in each dataset.[5] 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 testing $T \preccurlyeq \mathfrak{S}(G)$. Table 6.1 shows the average number of subtree isomorphism tests per graph for the Levelwise, Greedy, and BinarySearch algorithms (cf. Section 6.1). 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 super-quadratic complexity of the embedding operator.

We now compare our min-hash sketching technique (Algorithm 6.4) designed for probabilistic frequent subtree patterns with the best *naive* complete embedding algorithm from Table 6.1. 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 6.2 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 6.1. Column "best naive" shows the average number of evaluations performed by the best method from Table 6.1. 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 6.4 (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 embedding operator.

[5] We focus our exposition here on the simpler embedding operator from Chapter 4 and note that the methods behave similarly using the boosted embedding operator from Chapter 5.

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 6.2.: 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 6.4 for $K = 32, 64, 128$, and 256 (last four columns).

6.3.2. Predictive and Retrieval Performance

Finally we show 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 min-hash probabilistic frequent subtree 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 again 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 6.3 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 6.3 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.

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 6.3.: 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.

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 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% 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 6.1 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

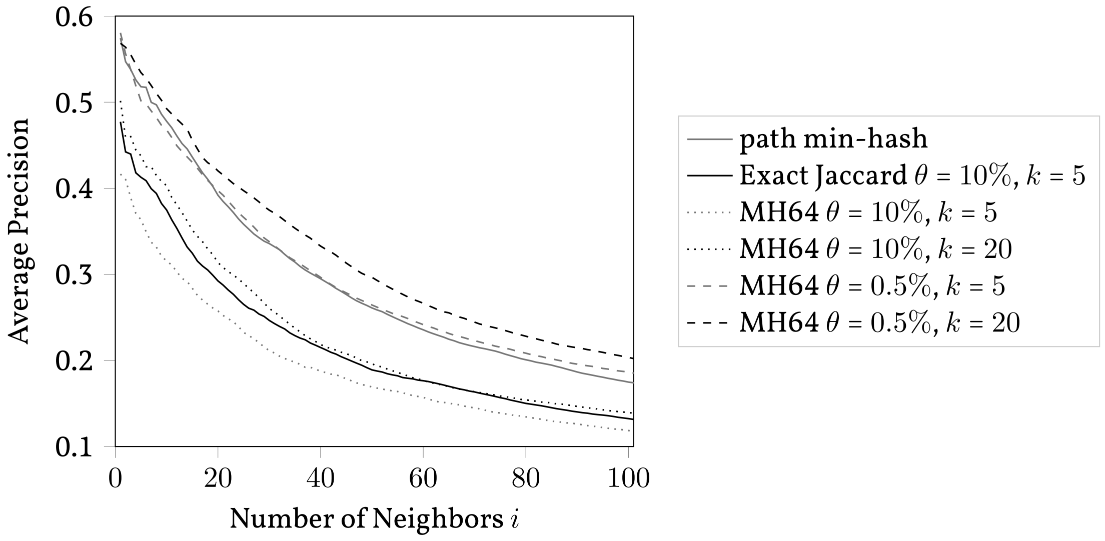


Figure 6.1.: Average fraction of "active" molecules among the i nearest neighbors of positive molecules in NCI-HIV dataset for path min-hash (Teixeira et al, 2012), exact Jaccard-similarity for frequent probabilistic tree patterns, and for our method with $K = 64$.

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 $\theta = 1\%$ and 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 inapplicable due to 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.

6.4. Summary and Open Questions

Many applications require to embed a large number of unseen graphs in a feature space. Being able to efficiently compute such feature embeddings for arbitrary unseen graphs is hence an important task. Though the probabilistic pattern matching operators discussed in the previous two chapters can be evaluated in polynomial time, each invocation during the embedding of a graph into probabilistic frequent subtree 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 (relaxed) 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 or graph streams into probabilistic frequent subtree feature spaces. This complements the results of the previous two chapters on the efficient mining of probabilistic frequent subtrees from arbitrary graph databases.

Our algorithms can easily be adapted to any finite pattern set and pattern matching operator if the pattern matching operator induces a partial order on the pattern set in which it is monotone or anti-monotone. They work, for example, if the pattern matching operator is defined by exact subgraph isomorphism or graph homomorphism. 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.

Furthermore, it would be interesting to investigate whether the evaluation strategies developed for the embedding computation have a positive effect in the context of ordinary frequent tree mining, as well. If we are only interested in frequent patterns and not in their support sets in a given database, a variant of the GREEDY algorithm can be used. That is, we find maximal frequent patterns by depth first search and then generate all subgraphs of maximal patterns, without checking their support in the database. Further work, however, is needed to generate these subgraphs nonredundantly (given that some of them might have been found already) and to efficiently identify the border and compute the support of extensions of this border. Recall that this is necessary, as we do not explicitly have the poset of frequent patterns given during the mining phase, but wish to compute it.

7. Conclusion

We now discuss the significance of our results in a broader context and outline some directions for future work. For a more detailed overview of the technical contributions, we refer the reader to Section 1.2.

7.1. Discussion

In this thesis we have proposed a system that allows to apply distance-based learning methods to arbitrary graph databases. This has been achieved by considering frequent subtrees as patterns and by relaxing the requirement on the completeness of the mining process and the embedding operator. In particular, we have defined probabilistic frequent subtrees and shown that they can be mined with polynomial delay in arbitrary graph databases. As a complementary contribution, we have shown how to quickly compute feature vectors for arbitrary graphs, given a set of tree patterns that span the feature space.

With these two steps, we have provided the required tools to apply probabilistic frequent subtrees in real life learning scenarios. Here first a suitable feature representation of an unknown graph distribution can be learned from a sample using the results from Chapter 4 and 5. Second, a model can be learned that is based on a suitable similarity measure on this feature representation. Finally, the model can be applied to new unseen graphs, by computing first a feature representation using the results from Chapter 6 and then feeding it to the model.

Our methods do not assume any structural or other restriction on the graph databases at hand. That is, the guarantee of polynomial delay holds for arbitrary graph databases and allows to mine frequent probabilistic subtrees also in such graph databases where state-of-the-art exact frequent subgraph and frequent subtree mining algorithms fail to produce any output in practically feasible time. Most of these algorithms were developed for chemical applications and stop working for even slightly more complex graph databases. Our method hence allows to compute frequent subtrees for graph databases where these patterns could not be computed previously. Furthermore, the efficient computation of feature representations for arbitrary graphs presented in this thesis allows not only to inspect such patterns qualitatively, but to use them in real-world machine learning applications.

Recall that the FCSM and FTM problems are computationally intractable. Hence any practical algorithm has to trade-off among speed, correctness, and general applicability. We have decided to maintain the general applicability and speed (in the sense of computational complexity) by giving up the correctness of the algorithm. Interestingly, as a byproduct of our methods, we obtain a positive result on the complexity of *exact* frequent subtree mining for locally easy graphs. That is, we propose a result that maintains the correctness and speed properties, but gives up the general applicability.

Locally easy graphs restrict the number of spanning trees in certain subgraphs of a graph without assuming any global structure. Its definition restricts only the block degree of the transaction graphs, and allows an arbitrary number of bridges to be incident to any vertex in the pattern and the transaction graph. Hence, we obtain the first positive result on the SubtreeIsomorphism problem that we are aware of, which allows unbounded vertex degree of the pattern tree for transaction graphs beyond forests. The vertex degree of the pattern is an important parameter of the complexity of the SubgraphIsomorphism problem (cf. Marx and Pilipczuk, 2014). With this result, we conjecture to be very close to the border between tractable and intractable restrictions of the FTM as well as the SubtreeIsomorphism problem.

7.2. Outlook

We have proposed two embedding operators, one which samples global spanning trees and the boosted algorithm which samples local spanning trees. Both can guarantee polynomial delay mining of frequent tree patterns, but differ in their runtimes and recall behaviors. It remains an open problem to decide which of the two algorithms should be chosen for a given graph database, or possibly even individually for each graph in a database. This idea can be extended even further: While we have shown that our methods are superior to exact frequent subgraph miners on complex graph databases, Gaston and the like are superior on chemical graphs (and probably on some other simple graph databases as well). It is possible to combine the potentially inefficient embedding computation with our method to get the best of both worlds: Introducing a new parameter, we allow a mining algorithm to store at most a certain number of embedding lists per graph. If a candidate pattern results in too many embeddings for a given graph, we discard them and switch to our probabilistic embedding operator for this graph. As the number of embeddings of a pattern is polynomial in the number of patterns of its predecessor for any given graph, this can be implemented efficiently. This adaptive algorithm could preserve the speed of Gaston on chemical graph databases, respectively that of our algorithm on other graph databases (with a small overhead). As this is mainly an engineering problem, we leave it for future work.

Another line of investigation would be to adapt the mining algorithm to the pattern class at hand. We have opted to consider only tree patterns in this thesis. Other classes of patterns might, however, allow faster algorithms: Using depth-first search, there is an immediate $O(|V(H)| \cdot |V(G)|)$ time algorithm to decide whether a pattern path H is subgraph isomorphic to a forest G, improving on the runtime of the algorithm for tree patterns. This implies that probabilistic frequent sub*paths* can be found faster than probabilistic frequent subtrees. Regarding an extension of our work in the other direction, there might be other simpler pattern classes, for which probabilistic frequent pattern mining can be solved efficiently.

Other directions for future work of course include novel application areas of frequent subtree mining in nontrivial graph classes that were previously inaccessible to frequent subgraph mining. For example, it might be possible to infer new nontrivial connections between input and output variables of a learning problem by finding frequent patterns in several neural networks that were trained independently for the same learning problems. For example, multiple echo state networks (Jaeger, 2002) can be trained easily using the same training data. Echo state networks have a fixed set of input and output vertices and a random hidden layer, only the weights of the edges containing vertices from the output layer are trained. Frequent tree patterns that contain output as well as input vertices might indicate relevant connections between the input and output variables. To this end, the algorithms presented in this thesis most likely need to be adapted to be able to process continuous edge labels, instead of only discrete edge labels.

Finally, the computational complexity of other restricted FCSM and FTM problems should be explored. While the complexities of the various pattern mining problems are closely related to the complexities of the corresponding embedding operators, there remains an interesting gap in our knowledge: For graph classes where the HamiltonianPath problem can be solved in polynomial time, but the SubgraphIsomorphism (resp. SubtreeIsomorphism) problem is **NP**-complete it is not clear whether a particular corresponding mining problem can be solved with polynomial delay, in incremental polynomial time, or if it cannot be solved in output polynomial time, unless **P** = **NP**. It would be interesting to see whether there are additional parameters (apart from the complexity of the HamiltonianPath and the complexity of the embedding operator) that influence the computational complexity of the pattern mining. Further results, both negative and positive would be important for a deeper understanding of the computational difficulties of frequent pattern mining.

A. The HamiltonianPath Problem for Cactus Graphs

In Section 2.2.2 we have discussed a connection between the complexity of the FTM problem for a transaction class $\mathcal{G}$ and the complexity of the SubtreeIsomorphism and HamiltonianPath problem. In particular, if the HamiltonianPath problem can be solved in polynomial time and the SubtreeIsomorphism problem is **NP**-complete then the complexity of the FTM problem is unclear. As discussed in Section 5.3, cactus graphs play an important role in the investigation of the border between polynomial delay frequent subtree mining and incremental polynomial time frequent subtree mining. The HamiltonianPath problem can be decided for cactus graph transactions in polynomial time. This follows from (Matoušek and Thomas, 1992) by noting that paths have vertex degree at most two and cactus graphs have tree-width at most two. The SubtreeIsomorphism problem, however, is already **NP**-complete for cactus graph transactions (Akutsu, 1993).

The above result is mainly of theoretical interest: Theorem 5.14 of Matoušek and Thomas (1992) gives a $O\left(|V(G)|^4\right)$ time algorithm for the HamiltonianPath problem for a given cactus graph G. It is possible to improve this runtime dramatically and to provide an easy-to-implement linear time algorithm for this problem. As we will see, the presented technique yields a fast linear time algorithm that decides the HamiltonianPath problem with one-sided error for arbitrary graphs.

Recall that a Hamiltonian path is a path in a graph G that contains each vertex of G exactly once. The HamiltonianPath problem (i.e., does there exist a Hamiltonian path in a given graph G?) is a well studied **NP**-complete problem with various applications (Garey and Johnson, 1979). Several algorithms have been proposed to find a Hamiltonian path in a graph, or to decide that none exists. For example, Held and Karp (1962) give a $O(n^2 \cdot 2^n)$ algorithm to compute a Hamiltonian path. Björklund (2014) gives a $O(1.657^n)$ time algorithm to count the number of Hamiltonian paths in a graph, which can also be used to decide the HamiltonianPath problem. Due to the exponential time complexity of those and other algorithms, it would be beneficial to derive simple, fast tests that can be run in advance to decide at least in some cases if there exists a Hamiltonian path, or not.

Many authors concentrated on sufficient conditions for a graph to be traceable (i.e., that it contains a Hamiltonian path). E.g. Dirac (1973) gives a lower bound on the number of edges in a graph that implies the existence of a Hamiltonian path. Also, there is a wide range of graph classes, where we know that a Hamiltonian path exists, e.g. complete graphs, cycles, paths, or graphs of the platonic solids.

We go a different way and consider situations which do not allow for a Hamiltonian path. That is, we define easily verifiable properties of graphs that prove that a graph is not traceable. To our knowledge, there is much less work in this direction. As a notable exception, Chvátal (1973) introduces weakly Hamiltonian graphs and derives necessary conditions for a graph to contain a Hamiltonian cycle. However, the paper uses quite involved concepts and the verification of the conditions for a given graph is not straightforward. Our conditions, on the other hand, can be checked in linear time and are easy to understand. They are based on partitioning a graph G into its biconnected components and defining a graph on those objects. In short, a Hamiltonian path in G can only exist if this graph is a path.

We start by considering trees and continue by defining a tree structure using the biconnected components of an arbitrary graph to devise conditions in Lemmas A.3 and A.4. As a direct application of our necessary conditions, we devise a linear time algorithm for cactus graphs in Theorem A.6. Finally, we give statistics of a molecular dataset that were obtained using our conditions.

A.1. Three Necessary Conditions

From now on, we only consider connected graphs, as otherwise there cannot be a Hamiltonian path. We start by considering the HamiltonianPath problem for trees. It is easy to see, that a tree T has a Hamiltonian path if and only if T is a path.

Lemma A.1. *A tree T has a Hamiltonian path if and only if T is a path.*

Proof. "$\Leftarrow$" is clear. "$\Rightarrow$" Let T be a tree and P a Hamiltonian path in T. P contains all vertices of T and has thus $|V(G)| - 1$ edges. Therefore, $E(T) = E(P)$ and thus T is a path. □

We will show that a generalized version of this holds for a graph defined on the *articulation* vertices of any graph G. We need the following definition:

Definition A.2. *Let G be a connected graph. A vertex $v \in V(G)$ is called articulation vertex if its removal disconnects G, i.e., the graph $G - v = (V', E')$ is disconnected, where $V' := V \setminus \{v\}$ and $E' := \{e \in E : v \notin e\}$. The criticality of v is the number of connected components of $G - v$.*

In a tree, every vertex that is not a leaf is an articulation vertex. We now prove the first necessary condition. In the case of trees, it follows directly from Lemma A.1.

Lemma A.3. *Let G be a traceable graph. Then all vertices have criticality at most 2.*

Proof. Suppose there is a vertex v with criticality at least 3. Then $G-v$ has three nonempty connected components C_1, C_2, C_3. Let P be a Hamiltonian path of G and u_1 (resp. u_2, u_3) be the first vertex in $V(C_1)$ (resp. $V(C_2), V(C_3)$) occurring in P (w.l.o.g. in this order). Any path connecting $u_1 \in V(C_1)$ to $u_2 \in V(C_2)$ in G needs to contain v. Otherwise, u and w would be contained in the same connected component of $G - v$. The same is true for a path from u_2 to u_3. Therefore, P contains v at least twice, which is a contradiction to P being a path. □

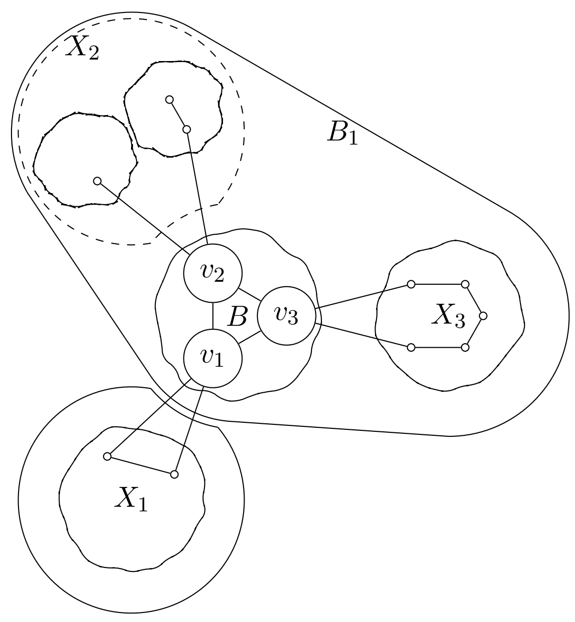


Figure A.1.: A cactus graph G without a Hamiltonian path. v_2 has criticality 3 (Lemma A.3) and the biconnected component B contains three articulation vertices (Lemma A.4).

Figure A.1 shows an illustration of the situation described in Lemma A.3. Vertex v_2 has criticality 3 and therefore does not allow for a Hamiltonian path in the graph. The next lemma focuses on biconnected components.

Lemma A.4. *Let G be a traceable graph. Then each biconnected component of G contains at most two articulation vertices.*

Proof. Suppose there is a biconnected component B of G that contains three articulation vertices v_1, v_2, v_3. Removing $v_i \in \{v_1, v_2, v_3\}$ from G results in a disconnected graph $G_i := G - v_i$. Now, there exists a connected component B_i in G_i such that $V(B_i) \cap V(B) = V(B) \setminus \{v_i\}$ and B_i is connected. Let X_i be the nonempty graph of all other connected components of $G - v_i$. Recall that B is a biconnected component, thus removing a single vertex does not disconnect B. Furthermore, all vertices in $V(B) \setminus \{v_i\}$ are contained in the same connected component of $G - v_i$. However, as v_i is an articulation vertex, $G - v_i$ is disconnected and thus $V(X_i) \neq \varnothing$. As an example, Figure A.1 shows B_i and X_i for the case $v_i = v_1$.

Claim: $V(X_i) \cap V(X_j) = \varnothing$ for all $i \neq j \in \{1, 2, 3\}$.

Using this claim, we can prove the lemma. A Hamiltonian path P of G needs to contain all vertices in $V(X_1), V(X_2), V(X_3)$. But to get from any vertex in $V(X_i)$ to a vertex $x \in V(X_j)$, it needs to pass through v_i. To get from v_i to x, the path must pass through v_j, as

$v_i \in V(B_j)$. Using the same argument as in the proof of Lemma A.3, we see that P needs to visit one of the articulation vertices v_1, v_2, v_3 at least twice, which is a contradiction to P being a path.

Proof of Claim: Suppose there exists $x \in V(X_i) \cap V(X_j)$. As $x \in V(X_i)$ there exists a path in X_i connecting x to a neighbor of v_i in G. Thus removing x_j would not disconnect x from $v_i \in V(B_j)$, which contradicts $x \in V(X_j)$. □

Lemma A.3 and Lemma A.4 together show that on any graph G, the existence of a Hamiltonian path implies a path-structure on the articulation vertices (respectively the biconnected components) of G. More exactly, let $\mathcal{A}(G)$ be the set of articulation vertices of G and $\mathcal{B}$ be the set of biconnected components of G. We define a new graph $A(G) = (\mathcal{A}(G), E')$ where E' is the set of all edges $\{v, w\}$ such that there exists $B \in \mathcal{B}$ with $v, w \in V(B)$. A similar definition yields a graph $B(G)$ on the biconnected components of G, where an edge exists between two biconnected components if and only if they share an articulation vertex. If G is traceable then $A(G)$ (respectively $B(G)$) must be a path.

The HAMILTONIANPATH problem hence reduces to checking if the two conditions formulated in Lemma A.3 and Lemma A.4 hold and if there is a Hamiltonian path in each biconnected component[1], that

- starts at the first articulation vertex and ends at the second articulation vertex (if there are two)
- starts at the articulation vertex (if there is one)
- starts and ends at arbitrary vertices (if there is no articulation vertex in G).

Finally, we call biconnected components that contain exactly one articulation vertex *leaf components* and finish this section with an easy corollary of the above considerations.

Corollary A.5. *Let G be a traceable graph. Then there are either zero or two leaf components.*

A.2. A Linear Time Algorithm for Cactus Graphs

The results of Section A.1 imply a polynomial time algorithm for the HAMILTONIAN-PATH problem for cactus graphs. A *cactus graph* is a connected graph where every biconnected component is either a single edge or a simple cycle. Figure A.2 shows a cactus graph and a graph that is no cactus graph.

Theorem A.6. *A cactus graph is traceable if and only if all of the following three conditions hold:*

- *Each vertex has criticality at most two*
- *Each biconnected component contains at most two articulation vertices*
- *If a biconnected component contains two articulation vertices, they share an edge.*

[1] Finding a Hamiltonian path in an arbitrary biconnected graph is of course still an **NP**-complete problem.

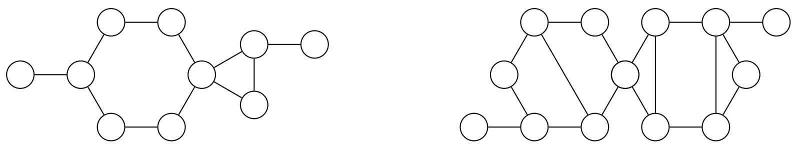


Figure A.2.: A cactus graph on the left and a graph that is not a cactus on the right.

Proof. Each cycle is traceable, and each Hamiltonian path of a cycle C starts at an arbitrary vertex of C and ends at one of its two neighbors. Edges are also traceable. "⇒" If a cactus graph G is traceable then, by Lemmas A.3 and A.4 the first two conditions hold. Let B be a biconnected component of G that contains two articulation vertices. If B is an edge, then the third condition holds trivially. If B is a cycle, then any Hamiltonian path must enter B through one articulation vertex v, leave it through the other w and can never enter B again. Therefore, the path from v to w must be a Hamiltonian path of B and therefore contains all edges in $E(B)$ except one, which must be $\{v, w\}$. "⇐" We construct a Hamiltonian path as follows: If G is biconnected (i.e., it has no articulation vertices), we construct a Hamiltonian path by removing an arbitrary edge. Otherwise, for each cycle, we remove the edge between the two articulation vertices or one of the edges incident to the unique articulation vertex in the cycle. Note that by this, each articulation vertex has degree two in the resulting graph P. As vertices with criticality zero have degree one or two in G, every vertex in P has degree less than three. We have removed exactly one edge from each cycle of G, thus P contains no cycles and is still connected. Therefore, P is a tree and by Lemma A.1 a path. $\square$

We can check the conditions of Theorem A.6 in linear time for a graph G as follows: First, we check if G is connected by a simple breadth first search in linear time. Next, we compute the biconnected components of G in linear time using Tarjans algorithm (Tarjan, 1972). Having the biconnected components (given as lists of edges), it is easy to compute the criticality of each vertex in G by counting the number of biconnected components each vertex occurs in as an endpoint of at least one edge. Having the criticality of each vertex, we can compute the number of critical vertices per biconnected components by a single pass over its edge list. To check if G is a cactus graph, we test if each biconnected component is either an edge or a simple cycle, which can also be done by a single pass over all edges in a biconnected component. If there are exactly two, by another pass we can check if the component contains an edge that contains both critical vertices. Therefore, the algorithm can be implemented to run in linear time with a small constant.

A.3. Some Statistics for Real-World Datasets

We have implemented some variants of the proposed algorithm and applied them to four large datasets: NCI-HIV, ZINC, POKEC and ENRON. These datasets are described in Section 2.4. For the latter two datasets we report results for all disks and all neighborhoods extracted from the monolithic graphs.

	NCI-HIV	ZINC	POKEC		ENRON	
			neighbors	disks	neighbors	disks
N	42 687	8 946 757	1 632 803	1 632 803	36 692	36 692
C	18 028	6 517 109	209 585	666 653	19 591	15 682
T	6	0	205 288	296 329	19 528	15 264
X	42 658	8 946 750	1 392 460	1 198 327	4 625	1 292
U	23	7	35 055	138 147	12 539	20 136
t_N	0.15s	28.97s	10.12s	15.72s	0.18s	0.22s
t_C	0.22s	44.22s	11.15s	22.85s	0.26s	0.36s
t_T	0.24s	55.08s	10.76s	20.64s	0.26s	0.36s
t_X	0.23s	49.98s	10.31s	20.24s	0.27s	0.38s

Table A.1.: Statistics for graph data sets. For each dataset, the number of graphs N, the number of cactus graphs C, the number of traceable cactus graphs T, and the number of untraceable graphs X. $U := N - T - X$ reports the number of graphs where our algorithm returns the uncertain answer that G might be traceable. Below the runtimes in seconds for computing these numbers are given.

Table A.1 shows the number of graphs N, the number of connected cactus graphs C, the number of traceable cactus graphs T, as well as the number of (arbitrary) graphs X that are definitively not traceable. $U := N - T - X$ reports the number of graphs where our algorithm returns the uncertain answer that G might be traceable. Furthermore, it reports the time t_i needed by our implementation to compute value $i \in \{N, C, T, X\}$. The numbers were computed by parsing the database from a text file and checking property i for each graph in the respective database. Times were measured using the GNU `time` command summing up `sys` and `user` times.

All experiments were done on an Intel Core i7-4470 with 8GB main memory running Ubuntu 14.04 64bit. The algorithms were implemented in C and compiled using `gcc 4.8.2` with optimization flag `-O3` enabled. No multi-threading was used. Furthermore, due to the fact that each graph can be processed separately, the maximum memory consumption at any time was less than 10MB.

t_N reports the time our implementation needs to parse the graph database, create graph objects in memory, and dump them again. The actual tests only add a small overhead in time compared to just parsing the data. On the other hand, by checking if a graph (a) is connected and (b) fulfills our three necessary conditions, we can declare most of the graphs from NCI-HIV, ZINC, and POKEC as non traceable. For ENRON, on the other hand, roughly half of the neighborhoods and roughly 40% of the disks are traceable cactus graphs. Most of the remaining graphs in the two ENRON variants are possibly traceable (i.e., we are not exactly sure and would need to verify using an exact algorithm). For NCI-HIV, ZINC, and POKEC, however, our algorithm correctly decides whether a graph is traceable or not for over 90% of the graphs. In particular, for the ZINC dataset we would only need to further investigate 7 out of almost 9 million graphs to check whether they are traceable, or not.

A.4. Summary

We have proposed three necessary conditions for a graph to be traceable that are easy and fast to check. Using them, we proposed a linear time algorithm that decides if a cactus graph is traceable. In more general practical settings, checking these conditions could be a first step that might, in many cases, make applying one of the exponential time exact algorithms obsolete. We evaluated our tests' effectiveness in that respect on three molecular data sets of varying size and showed that most molecular graphs can be easily identified as non-traceable using our conditions.

Our algorithm can be extended to yield an exact polynomial time algorithm for more general classes of graphs. Using our conditions, we can reduce the HAMILTONIANPATH problem in a non-biconnected graph G to smaller HAMILTONIANPATH problems in the biconnected components of G. We would only need to check if there is a Hamiltonian path in each biconnected component that connects the two articulation vertices or starts at the unique articulation vertex, respectively. This is possible in polynomial time if, for example, the number of spanning trees in each biconnected component is bounded by a polynomial p in the size of G. This is the case, for example, in locally easy graphs (compare Section 5.3). Here, the algorithm presented in this chapter runs in $O\left(p(G) \cdot |E(G)|\right)$ time. Algorithm 5.1 solves the HAMILTONIANPATH problem for locally easy graphs in $O\left(f^2(G) \cdot |E(G)| \cdot |V(G)|^{1.5}\right)$ time, where $f(G) \geq p(G)$ (compare Lemma 5.7).

B. Poissons Binomial Distribution

In the proof of Theorem 4.4 we have used a bound on the cumulative density function (CDF) of Poissons binomial distribution (see, e.g., Nedelman and Wallenius, 1986; Wang, 1993) by the CDF of a binomial distribution. While the claim is quite intuitive, we have not found a proof in any textbook or related article. We will hence prove the result here for completeness.

Let $n \in \mathbb{N}$ and let $p_i \in [0,1]$ be the success probability of a binary random variable X_i for all $i \in [n]$. The CDF of Poissons binomial distribution for the parameters $\{p_1, \ldots, p_n\}$ is then given by

$$\mathbf{P}\left(\sum_{i=0}^{n} X_i \leq k\right) = \sum_{\substack{A \subseteq [n] \\ |A| \leq k}} \prod_{i \in A} p_i \prod_{i \in \overline{A}} (1 - p_i)$$

where $\overline{A} := [n] \setminus A$ is the complement of A in $[n]$. The claim used in Theorem 4.4 is as follows:

Lemma B.1. *Let $p \in [0,1]$ such that $p \leq p_i$ for all $i \in [n]$. Then the CDF of Poissons binomial distribution for parameters $\{p_1, \ldots, p_n\}$ can be bounded by the CDF of a binomial distribution with parameters p and n, i.e., for all $k \in [n] \cup \{0\}$*

$$\boldsymbol{P}\left(\sum_{i=0}^{n} X_i \leq k\right) \leq \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}$$

Proof. Obviously, if $p_i = p$ for all $i \in [n]$, then Poissons binomial distribution is exactly the binomial distribution. We shall prove inequality for the case that we replace only one probability p_j by a smaller probability. More exactly we replace the parameters $\{p_1, \ldots, p_n\}$ by $\{p'_1, \ldots, p'_n\}$ with

$$p'_i := \begin{cases} p_i & \text{for } i \neq j \\ p & \text{for } i = j \end{cases}$$

and consider $\mathbf{P}\left(\sum_{i=0}^{n} X'_i \leq k\right)$ where X'_i is a binary random variable with success probability p'_i for all $i \in [n]$. The proof for the general case then immediately follows by induction on the number of replaced variables.

Let $P(A) := \prod_{i \in A} p_i \prod_{i \in \overline{A}} (1-p_i)$ and $P'(A) := \prod_{i \in A} p'_i \prod_{i \in \overline{A}} (1-p'_i)$ for any $A \subseteq [n]$. Then we can rewrite

$$\mathbf{P}\left(\sum_{i=0}^{n} X_i \leq k\right) = \sum_{\substack{A \subseteq [n] \\ |A| \leq k}} P(A) = \sum_{\substack{A \subseteq [n] \\ |A| \leq k \\ j \in A}} \left(P(A) + P(A \setminus \{j\})\right) + \sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P(A)$$

for the index $j \in [n]$ that we have fixed above. We can now bound this expression by noting that for any $A \subseteq [n]$ with $j \notin A$ it holds that $P(A) \leq P'(A)$, as $1 - p_i \leq 1 - p$. Hence

$$\sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P(A) \leq \sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P'(A) .$$

Furthermore, we can show for $A \subseteq [n]$ with $j \in A$ that

$$\begin{aligned}
P(A) + P(A \setminus \{j\}) &= \prod_{i \in A} p_i \prod_{i \in \overline{A}} (1 - p_i) + \prod_{i \in A \setminus \{j\}} p_i \prod_{i \in \overline{A \setminus \{j\}}} (1 - p_i) \\
&= p_j \prod_{i \in A \setminus \{j\}} p_i \prod_{i \in \overline{A}} (1 - p_i) + (1 - p_j) \prod_{i \in A \setminus \{j\}} p_i \prod_{i \in \overline{A}} (1 - p_i) \\
&= \prod_{i \in A \setminus \{j\}} p_i \prod_{i \in \overline{A}} (1 - p_i) \cdot (p_j + (1 - p_j)) \\
&= \prod_{i \in A \setminus \{j\}} p_i \prod_{i \in \overline{A}} (1 - p_i) \\
&= \prod_{i \in A \setminus \{j\}} p'_i \prod_{i \in \overline{A}} (1 - p'_i) \\
&= \prod_{i \in A \setminus \{j\}} p'_i \prod_{i \in \overline{A}} (1 - p'_i) \cdot (p + (1 - p)) \\
&= \dots \\
&= P'(A) + P'(A \setminus \{j\})
\end{aligned}$$

As a result, we have

$$\begin{aligned}
\mathbf{P}\left(\sum_{i=0}^{n} X_i \leq k \right) &= \sum_{\substack{A \subseteq [n] \\ |A| \leq k \\ j \in A}} (P(A) + P(A \setminus \{j\})) + \sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P(A) \\
&= \sum_{\substack{A \subseteq [n] \\ |A| \leq k \\ j \in A}} \left(P'(A) + P'(A \setminus \{j\})\right) + \sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P(A) \\
&\leq \sum_{\substack{A \subseteq [n] \\ |A| \leq k \\ j \in A}} \left(P'(A) + P'(A \setminus \{j\})\right) + \sum_{\substack{A \subseteq [n] \\ |A| = k \\ j \notin A}} P'(A) \\
&= \mathbf{P}\left(\sum_{i=0}^{n} X'_i \leq k \right)
\end{aligned}$$

$\square$

Bibliography

Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, A Inkeri Verkamo (1996) Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, AAAI/MIT Press, pp 307–328

Tatsuya Akutsu (1993) A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 76(9):1488–1493

Stefan Arnborg, Derek G Corneil, Andrzej Proskurowski (1987) Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic Discrete Methods 8(2):277–284, DOI 10.1137/0608024

Tatsuya Asai, Hiroki Arimura, Takeaki Uno, Shin-Ichi Nakano (2003) Discovering frequent substructures in large unordered trees. In: Gunter Grieser, Yuzuru Tanaka, Akihiro Yamamoto (eds) Discovery Science (DS) Proceedings, Springer, Lecture Notes in Computer Science, vol 2843, pp 47–61, DOI 10.1007/978-3-540-39644-4_6

Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroshi Sakamoto, Hiroki Arimura, Setsuo Arikawa (2004) Efficient substructure discovery from large semi-structured data. IEICE Transactions on Information and Systems 87-D(12):2754–2763, URL http://search.ieice.org/bin/summary.php?id=e87-d_12_2754

László Babai (2015) Graph isomorphism in quasipolynomial time. The Computing Research Repository (CoRR) abs/1512.03547, URL http://arxiv.org/abs/1512.03547

Andreas Björklund (2014) Determinant sums for undirected hamiltonicity. SIAM Journal on Computing 43(1):280–299, DOI 10.1137/110839229

Mario Boley (2011) The efficient discovery of interesting closed pattern collections. PhD thesis, University of Bonn, URL http://hss.ulb.uni-bonn.de/2011/2700/2700.htm

Christian Borgelt (2009) Graph mining: An overview, URL http://borgelt.net/papers/gmagi_09.pdf

Christian Borgelt, Michael R Berthold (2002) Mining molecular fragments: Finding relevant substructures of molecules. In: Vipin Kumar, Shusaku Tsumoto, Ning Zhong, Philip S Yu, Xindong Wu (eds) IEEE International Conference on Data Mining (ICDM) Proceedings, IEEE Computer Society, pp 51–58, DOI 10.1109/ICDM.2002.1183885

Christian Borgelt, Thorsten Meinl, Michael Berthold (2005) Moss: A program for molecular substructure mining. In: Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations, ACM, New York, NY, USA, OSDM '05, pp 6–15, DOI 10.1145/1133905.1133908

Björn Bringmann, Siegfried Nijssen (2008) What is frequent in a single graph? In: Takashi Washio, Einoshin Suzuki, Kai Ming Ting, Akihiro Inokuchi (eds) Advances in Knowledge Discovery and Data Mining, Pacific-Asia Conference (PAKDD) Proceedings, Springer, Lecture Notes in Computer Science, vol 5012, pp 858–863, DOI 10.1007/978-3-540-68125-0_84

Björn Bringmann, Albrecht Zimmermann, Luc De Raedt, Siegfried Nijssen (2006) Don't be afraid of simpler patterns. In: Johannes Fürnkranz, Tobias Scheffer, Myra Spiliopoulou (eds) European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD) Proceedings, Springer, Lecture Notes in Computer Science, vol 4213, pp 55–66, DOI 10.1007/11871637_10

Andrei Z Broder (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

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

Arthur Cayley (1889) A theorem on trees. Quarterly Journal of Pure and Applied Mathematics 23:376–378, DOI 10.1017/cbo9780511703799.010

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

Chen Chen, Cindy Xide Lin, Matt Fredrikson, Mihai Christodorescu, Xifeng Yan, Jiawei Han (2009) Mining graph patterns efficiently via randomized summaries. PVLDB 2(1):742–753, URL http://www.vldb.org/pvldb/2/vldb09-80.pdf

Yun Chi, Yirong Yang, Richard R Muntz (2003) Indexing and mining free trees. In: Xindong Wu, Alex Tuzhilin, Jude Shavlik (eds) IEEE International Conference on Data Mining (ICDM) Proceedings, IEEE Computer Society, pp 509–512

Yun Chi, Yirong Yang, R R Muntz (2004a) Hybridtreeminer: an efficient algorithm for mining frequent rooted trees and free trees using canonical forms. In: International Conference on Scientific and Statistical Database Management (SSDBM) Proceedings, IEEE Computer Society, pp 11–20, DOI 10.1109/SSDM.2004.1311189

Yun Chi, Yirong Yang, Yi Xia, Richard R Muntz (2004b) Cmtreeminer: Mining both closed and maximal frequent subtrees. In: Honghua Dai, Ramakrishnan Srikant, Chengqi Zhang (eds) Advances in Knowledge Discovery and Data Mining, Pacific-Asia Conference (PAKDD), Proceedings, Springer, Lecture Notes in Computer Science, vol 3056, pp 63–73, DOI 10.1007/978-3-540-24775-3_9

Yun Chi, Richard R Muntz, Siegfried Nijssen, Joost N Kok (2005) Frequent subtree mining - an overview. Fundamenta Informaticae 66(1–2):161–198

Moon Jung Chung (1987) $O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees. Journal of Algorithms 8(1):106–112, DOI 10.1016/0196-6774(87)90030-7

Václav Chvátal (1973) Edmonds polytopes and weakly hamiltonian graphs. Mathematical Programming 5(1):29–40

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento (1998) Subgraph Transformations for the Inexact Matching of Attributed Relational Graphs, Springer Vienna, pp 43–52. DOI 10.1007/978-3-7091-6487-7_5

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, Mario Vento (1999) Performance evaluation of the VF graph matching algorithm. In: International Conference on Image Analysis and Processing (ICIAP), IEEE Computer Society, pp 1172–1177, DOI 10.1109/ICIAP.1999.797762

Corinna Cortes, Vladimir Vapnik (1995) Support-vector networks. Machine Learning 20(3):273–297, DOI 10.1007/bf00994018

Víctor Dalmau, Phokion G Kolaitis, Moshe Y Vardi (2002) Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Pascal Van Hentenryck (ed) Principles and Practice of Constraint Programming (CP) Proceedings, Springer, Lecture Notes in Computer Science, vol 2470, pp 310–326, DOI 10.1007/3-540-46135-3_21

Asim K Debnath, Rosa L Lopez de Compadre, Gargi Debnath, Alan J Shusterman, Corwin Hansch (1991) Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. Journal of Medicinal Chemistry 34(2):786–797, DOI 10.1021/jm00106a046

Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, George Karypis (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

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

Gabriel A Dirac (1973) Note on hamilton circuits and hamilton paths. Mathematische Annalen 206(2):139–147

Pál Erdős, Alfréd Rényi (1959) On random graphs I. Publicationes Mathematicae 6:290–297

Edward Fredkin (1960) Trie memory. Communications of the ACM 3(9):490–499, DOI 10.1145/367390.367400

Michael R Garey, David S Johnson (1979) Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman

Thomas Gärtner, Peter Flach, Stefan Wrobel (2003) On graph kernels: Hardness results and efficient alternatives. In: Bernhard Schölkopf, Manfred K Warmuth (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

Johannes Gehrke, Paul Ginsparg, Jon M Kleinberg (2003) Overview of the 2003 KDD cup. SIGKDD Explorations 5(2):149–151, DOI 10.1145/980972.980992

Hanna Geppert, Tamás Horváth, Thomas Gärtner, Stefan Wrobel, Jürgen Bajorath (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

MohammadTaghi Hajiaghayi, Naomi Nishimura (2007) Subgraph isomorphism, log-bounded fragmentation, and graphs of (locally) bounded treewidth. Journal of Computer and System Sciences 73(5):755–768, DOI 10.1016/j.jcss.2007.01.003

Jiawei Han, Jian Pei, Yiwen Yin, Runying Mao (2004) Mining frequent patterns without candidate generation: A frequent-pattern tree approach. Data Mining and Knowledge Discovery 8(1):53–87, DOI 10.1023/b:dami.0000005258.31418.83

Frank Harary (1994) Graph Theory. Addison-Wesley series in mathematics, Perseus Books

Michael Held, Richard M Karp (1962) A dynamic programming approach to sequencing problems. Journal of the Society for Industrial & Applied Mathematics 10(1):196–210

John E Hopcroft, Richard M Karp (1973) An n^5/2 algorithm for maximum matchings in bipartite graphs. SIAM Journal on Computing 2(4):225–231, DOI 10.1137/0202019

John E Hopcroft, J K Wong (1974) Linear time algorithm for isomorphism of planar graphs (preliminary report). In: Robert L Constable, Robert W Ritchie, Jack W Carlyle, Michael A Harrison (eds) ACM Symposium on the Theory of Computing (STOC) Proceedings, ACM, pp 172–184, DOI 10.1145/800119.803896

Tamás Horváth, Jan Ramon (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

Tamás Horváth, Thomas Gärtner, Stefan Wrobel (2004) Cyclic pattern kernels for predictive graph mining. In: Won Kim, Ron Kohavi, Johannes Gehrke, William DuMouchel (eds) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Proceedings, ACM, pp 158–167, DOI 10.1145/1014052.1014072

Tamás Horváth, Björn Bringmann, Luc De Raedt (2007) Frequent hypergraph mining. In: Stephen Muggleton, Ramón P Otero, Alireza Tamaddoni-Nezhad (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

Tamás Horváth, Jan Ramon, Stefan Wrobel (2010) Frequent subgraph mining in outerplanar graphs. Data Mining and Knowledge Discovery 21(3):472–508, DOI 10.1007/s10618-009-0162-1

Tamás Horváth, Keisuke Otaki, Jan Ramon (2013) Efficient frequent connected induced subgraph mining in graphs of bounded tree-width. In: Hendrik Blockeel, Kristian Kersting, Siegfried Nijssen, Filip Zelezný (eds) European Conference on Machine Learning and Knowledge Discovery in Databases ECML PKDD Proceedings, Part I, Springer, Lecture Notes in Computer Science, vol 8188, pp 622–637, DOI 10.1007/978-3-642-40988-2_40

Jun Huan, Wei Wang, Jan Prins (2003) Efficient mining of frequent subgraphs in the presence of isomorphism. In: Xindong Wu, Alex Tuzhilin, Jude Shavlik (eds) IEEE International Conference on Data Mining (ICDM) Proceedings, IEEE Computer Society, pp 549–552, DOI 10.1109/ICDM.2003.1250974

Jun Huan, Wei Wang, Jan Prins, Jiong Yang (2004) SPIN: mining maximal frequent subgraphs from graph databases. In: Won Kim, Ron Kohavi, Johannes Gehrke, William DuMouchel (eds) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Proceedings, ACM, pp 581–586, DOI 10.1145/1014052.1014123

John J Irwin, Teague Sterling, Michael M Mysinger, Erin S Bolstad, Ryan G Coleman (2012) ZINC: A free tool to discover chemistry for biology. Journal of Chemical Information and Modeling 52(7):1757–1768, DOI 10.1021/ci3001277

Herbert Jaeger (2002) Adaptive nonlinear system identification with echo state networks. In: Suzanna Becker, Sebastian Thrun, Klaus Obermayer (eds) Advances in Neural Information Processing Systems (NIPS) Proceedings, MIT Press, pp 593–600, URL http://papers.nips.cc/paper/2318-adaptive-nonlinear-system-identification-with-echo-state-networks

Chuntao Jiang, Frans Coenen, Michele Zito (2013) A survey of frequent subgraph mining algorithms. Knowledge Engineering Review 28(1):75–105, DOI 10.1017/S0269888912000331

David S Johnson, Christos H Papadimitriou, Mihalis Yannakakis (1988) On generating all maximal independent sets. Information Processing Letters 27(3):119–123, DOI 10.1016/0020-0190(88)90065-8

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

Bryan Klimt, Yiming Yang (2004) The enron corpus: A new dataset for email classification research. In: Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, Dino Pedreschi (eds) European Conference on Machine Learning (ECML) Proceedings, Springer, Lecture Notes in Computer Science, vol 3201, pp 217–226, DOI 10.1007/978-3-540-30115-8_22

Bernhard Korte, Jens Vygen (2012) Combinatorial Optimization. Springer Berlin Heidelberg, DOI 10.1007/978-3-642-24488-9

Varun Krishna, N N R Ranga Suri, G Athithan (2011) A comparative survey of algorithms for frequent subgraph discovery. Current Science 100(2):190–198, URL https://www.jstor.org/stable/24073045

Joseph B Kruskal (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proceedings of the American Mathematical Society 7(1):48–50, DOI 10.1090/S0002-9939-1956-0078686-7

Michihiro Kuramochi, George Karypis (2001) Frequent subgraph discovery. In: Nick Cercone, Tsau Young Lin, Xindong Wu (eds) IEEE International Conference on Data Mining (ICDM), Proceedings, IEEE Computer Society, pp 313–320, DOI 10.1109/ICDM.2001.989534

Michihiro Kuramochi, George Karypis (2004) An efficient algorithm for discovering frequent subgraphs. Transactions on Knowledge and Data Engineering 16(9):1038–1051, DOI 10.1109/TKDE.2004.33

Ruirui Li, Wei Wang (2015) REAFUM: representative approximate frequent subgraph mining. In: Suresh Venkatasubramanian, Jieping Ye (eds) SIAM International Conference on Data Mining (SDM) Proceedings, SIAM, pp 757–765, DOI 10.1137/1.9781611974010.85

Andrzej Lingas (1983) An application of maximum bipartite c-matching to subtree isomorphism. In: Giorgio Ausiello, Marco Protasi (eds) Trees in Algebra and Programming (CAAP) Proceedings, Springer, Lecture Notes in Computer Science, vol 159, pp 284–299, DOI 10.1007/3-540-12727-5_17

Ulrike von Luxburg (2007) A tutorial on spectral clustering. Statistics and Computing 17(4):395–416, DOI 10.1007/s11222-007-9033-z

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

Dániel Marx, Michal Pilipczuk (2014) Everything you always wanted to know about the parameterized complexity of Subgraph Isomorphism (but were afraid to ask). In: Ernst W Mayr, Natacha Portier (eds) International Symposium on Theoretical Aspects of Computer Science (STACS), Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, LIPIcs, vol 25, pp 542–553, DOI 10.4230/LIPIcs.STACS.2014.542

Jiří Matoušek, Robin Thomas (1992) On the complexity of finding iso-and other morphisms for partial k-trees. Discrete Mathematics 108(1–3):343–364, DOI 10.1016/0012-365x(92)90687-b

David W Matula (1968) An algorithm for subtree identification. Siam Review 10:273–274

David W Matula (1978) Subtree isomorphism in o(n5/2). Annals of Discrete Mathematics 2:91–106, DOI 10.1016/S0167-5060(08)70324-8

Jerry Nedelman, Ted Wallenius (1986) Bernoulli trials, poisson trials, surprising variances, and jensen's inequality. The American Statistician 40(4):286–289, DOI 10.2307/2684605

Siegfried Nijssen, Joost N Kok (2003) Efficient discovery of frequent unordered trees. In: First international workshop on mining graphs, trees, and sequences

Siegfried Nijssen, Joost N Kok (2004) A quickstart in frequent structure mining can make a difference. In: Won Kim, Ron Kohavi, Johannes Gehrke, William DuMouchel (eds) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Proceedings, ACM, pp 647–652, DOI 10.1145/1014052.1014134

Siegfried Nijssen, Joost N Kok (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

André Petermann, Martin Junghanns, Erhard Rahm (2017) Dimspan - transactional frequent subgraph mining with distributed in-memory dataflow systems. CoRR abs/1703.01910, URL http://arxiv.org/abs/1703.01910, 1703.01910

Liva Ralaivola, Sanjay J Swamidass, Hiroto Saigo, Pierre Baldi (2005) Graph kernels for chemical informatics. Neural Networks 18(8):1093–1110, DOI 10.1016/j.neunet.2005.07.009

Ronald C Read, Robert Tarjan (1975) Bound on backtrack algorithms for listing cycles, paths, and spanning trees. Networks 5:237–252

Neil Robertson, Paul D Seymour (1986a) Graph minors. II. algorithmic aspects of tree-width. Journal of Algorithms 7(3):309–322, DOI 10.1016/0196-6774(86)90023-4

Neil Robertson, Paul D Seymour (1986b) Graph minors. V. excluding a planar graph. Journal of Combinatorial Theory, Series B 41(1):92–114, DOI 10.1016/0095-8956(86)90030-4

Ulrich Rückert, Stefan Kramer (2004) Frequent free tree discovery in graph data. In: Hisham Haddad, Andrea Omicini, Roger L Wainwright, Lorie M Liebrock (eds) ACM Symposium on Applied Computing (SAC), Proceedings, ACM, pp 564–570, DOI 10.1145/967900.968018

Till Hendrik Schulz, Tamás Horváth, Pascal Welke, Stefan Wrobel (2018) Mining tree patterns with partially injective homomorphisms. In: Michele Berlingerio, Francesco Bonchi, Thomas Gärtner, Neil Hurley, Georgiana Ifrim (eds) European Conference on Machine Learning and Knowledge Discovery in Databases ECML PKDD Proceedings, Part II, Springer, Lecture Notes in Computer Science, vol 11052, pp 585–601, DOI 10.1007/978-3-030-10928-8_35

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

John Shawe-Taylor, Nello Cristianini (2004) Kernel Methods for Pattern Analysis. Cambridge University Press

Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, Karsten M Borgwardt (2011) Weisfeiler-lehman graph kernels. Journal of Machine Learning Research 12:2539–2561

Qinfeng Shi, James Petterson, Gideon Dror, John Langford, Alexander J Smola, S V N Vishwanathan (2009) Hash kernels for structured data. Journal of Machine Learning Research 10:2615–2637, DOI 10.1145/1577069.1755873

Neil James Alexander Sloane (2016) The Online Encyclopedia of Integer Sequences. A000055: Number of trees with n unlabeled nodes. Online, URL http://oeis.org/A000055, accessed 2016-11-18

Richard P Stanley, Sergey Fomin (1999) Enumerative Combinatorics, Cambridge Studies in Advanced Mathematics, vol 2. Cambridge University Press, DOI 10.1017/CBO9780511609589

Lubos Takac, Michal Zabovsky (2012) Data analysis in public social networks. In: International Scientific Conference and International Workshop Present Day Trends of Innovations, pp 1–6

Robert Tarjan (1972) Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2):146–160

Carlos H C Teixeira, Arlei Silva, Wagner Meira Jr (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

Alexandre Termier, Marie-Christine Rousset, Michèle Sebag (2002) Treefinder: a first step towards XML data mining. In: Vipin Kumar, Shusaku Tsumoto, Ning Zhong, Philip S Yu, Xindong Wu (eds) IEEE International Conference on Data Mining (ICDM) Proceedings, IEEE Computer Society, pp 450–457, DOI 10.1109/ICDM.2002.1183987

Julian R Ullmann (1976) An algorithm for subgraph isomorphism. Journal of the ACM 23(1):31–42, DOI 10.1145/321921.321925

Katrin Ullrich, Jennifer Mack, Pascal Welke (2016) Ligand affinity prediction with multi-pattern kernels. In: Toon Calders, Michelangelo Ceci, Donato Malerba (eds) Discovery Science (DS) Proceedings, Lecture Notes in Computer Science, vol 9956, pp 474–489, DOI 10.1007/978-3-319-46307-0_30

Leslie G Valiant (1979) The complexity of computing the permanent. Theoretical Computer Science 8:189–201, DOI 10.1016/0304-3975(79)90044-6

Rakesh M Verma, Steven W Reyner (1989) An analysis of a good algorithm for the subtree problem, corrected. SIAM Journal on Computing 18(5):906–908, DOI 10.1137/0218062

Nikil Wale, Ian A Watson, George Karypis (2008) Comparison of descriptor spaces for chemical compound retrieval and classification. Knowledge and Information Systems 14(3):347–375, DOI 10.1007/s10115-007-0103-5

Y H Wang (1993) On the number of successes in independent trials. Statistica Sinica 3(2):295–312, URL http://www3.stat.sinica.edu.tw/statistica/j3n2/j3n23/j3n23.htm

Pascal Welke (2017) Simple necessary conditions for the existence of a Hamiltonian path with applications to cactus graphs. CoRR abs/1709.01367, URL http://arxiv.org/abs/1709.01367

Pascal Welke, Tamás Horváth, Stefan Wrobel (2015) On the complexity of frequent subtree mining in very simple structures. In: Jesse Davis, Jan Ramon (eds) Inductive Logic Programming (ILP) Revised Selected Papers, Springer, Lecture Notes in Computer Science, vol 9046, pp 194–209, DOI 10.1007/978-3-319-23708-4_14

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

Pascal Welke, Tamás Horváth, Stefan Wrobel (2016b) Probabilistic frequent subtree kernels. In: Michelangelo Ceci, Corrado Loglisci, Giuseppe Manco, Elio Masciari, Zbigniew Ras (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

Pascal Welke, Tamás Horváth, Stefan Wrobel (2018) Probabilistic frequent subtrees for efficient graph classification and retrieval. Machine Learning 107(11):1847–1873, DOI 10.1007/s10994-017-5688-7

Pascal Welke, Tamás Horváth, Stefan Wrobel (2019) Probabilistic and exact frequent subtree mining in graphs beyond forests. Machine Learning DOI 10.1007/s10994-019-05779-1, (online)

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

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

Marc Wörlein, Thorsten Meinl, Ingrid Fischer, Michael Philippsen (2005) A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and gaston. In: Alípio Jorge, Luís Torgo, Pavel Brazdil, Rui Camacho, João Gama (eds) European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD) Proceedings, Springer, Lecture Notes in Computer Science, vol 3721, pp 392–403, DOI 10.1007/11564126_39

Xifeng Yan, Jiawei Han (2002) gSpan: Graph-based substructure pattern mining. In: Vipin Kumar, Shusaku Tsumoto, Ning Zhong, Philip S Yu, Xindong Wu (eds) IEEE International Conference on Data Mining (ICDM) Proceedings, IEEE Computer Society, pp 721–724, DOI 10.1109/icdm.2002.1184038

Xifeng Yan, Jiawei Han (2003) Closegraph: mining closed frequent graph patterns. In: Lise Getoor, Ted E Senator, Pedro M Domingos, Christos Faloutsos (eds) ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), Proceedings, ACM, pp 286–295, DOI 10.1145/956750.956784

Mohammed Javeed Zaki (2002) Efficiently mining frequent trees in a forest. In: ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), ACM, pp 71–80, DOI 10.1145/775047.775058

Peixiang Zhao, Jeffrey Xu Yu (2007) Mining closed frequent free trees in graph databases. In: Kotagiri Ramamohanarao, P Radha Krishna, Mukesh K Mohania, Ekawit Nantajeewarawat (eds) International Conference on Database Systems for Advanced Applications (DASFAA) Proceedings, Springer, Lecture Notes in Computer Science, vol 4443, pp 91–102, DOI 10.1007/978-3-540-71703-4_10

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

Zhaonian Zou, Jianzhong Li, Hong Gao, Shuo Zhang (2010) Mining frequent subgraph patterns from uncertain graph data. IEEE Transactions on Knowledge and Data Engineering 22(9):1203–1218, DOI 10.1109/tkde.2010.80