On the Usefulness of Weight-Based Constraints in Frequent Subgraph Mining*

Frank Eichinger, Matthias Huber, and Klemens Böhm

Karlsruhe Institute of Technology (KIT), Germany
{eichinger, matthias.huber, klemens.boehm}@kit.edu

Abstract. Frequent subgraph mining is an important data-mining technique. In this paper we look at weighted graphs, which are ubiquitous in the real world. The analysis of weights in combination with mining for substructures might yield more precise results. In particular, we study frequent subgraph mining in the presence of weight-based constraints and explain how to integrate them into mining algorithms. While such constraints only yield approximate mining results in most cases, we demonstrate that such results are useful nevertheless and explain this effect. To do so, we both assess the completeness of the approximate result sets, and we carry out application-oriented studies with real-world data-analysis problems: software-defect localization, weighted graph classification and explorative mining in logistics. Our results are that the runtime can improve by a factor of up to 3.5 in defect localization and classification and 7 in explorative mining. At the same time, we obtain an even slightly increased defect-localization precision, stable classification precision and obtain good explorative mining results.

1 Introduction

Graph mining has drawn a lot of attention recently. One important technique is *frequent subgraph mining* [24], with applications in chemistry and web mining [8], databases [25] etc. It is often used as a building block of some higher-level analysis task such as *cluster analysis* or *graph classification* [10]. With the latter, frequent subgraph patterns are mined from a set of classified graphs. A standard classifier is then learned on the subgraph features discovered.

Though frequent subgraph mining is an established technique, relying on the pure graph structure is not always sufficient: Many real-world problems correspond to *weighted graphs*. For instance, think of transportation graphs [9]. Weights represent costs, the average speed etc. In software engineering, edge-weighted call graphs (see Figure 1 for an example) have turned out to be beneficial for software-defect localization [4]. To take both the graph structure and the weights into account when mining weighted graphs, one can analyze weights in a preprocessing or in a postprocessing step, before or after the actual (unweighted) subgraph mining takes place [5, 9, 12]. However, both variants have

* Technical Report 2010-10, Faculty of Informatics, Karlsruhe Institute of Technology.

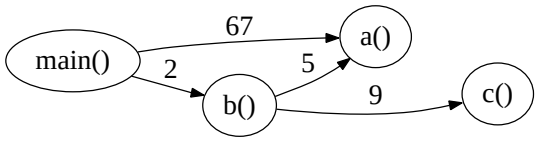


Fig. 1. Example call graph. Edges represent method calls, edge weights call frequencies.

issues: Discretizing numerical values to categorical labels during preprocessing might loose important information, as edges with similar weights can fall into different intervals. Postprocessing in turn is not always efficient. This is because the mining algorithm first ignores the weights and might generate a huge number of subgraphs. The second step however discards most of them.

A cheaper way to perform frequent subgraph mining with weights is *approximate graph mining* [7, 21]. Another approach is *constraint-based mining* [19, 27]. Constraints can be used to prune the search space and speed up mining. We for our part investigate approximate frequent subgraph mining with weight-based constraints. This is promising, since various higher-level analysis tasks imply meaningful weight-based constraints, as we will show. In a classification scenario, to give an example, a natural constraint would demand weights in the subgraph patterns with a high discriminativeness. While constraints lead to smaller result sets, we hypothesize that those application-specific constraints do not lower the result quality of the higher-level problem. However, not every constraint is good for pruning in a straightforward way. Literature has introduced *anti-monotone constraints* [11, 19, 27]: When using them for pruning, the algorithm still finds all patterns. However, most weight-based constraints are not anti-monotone, for the following reason: Graph topology and weights are independent of each other, at least in theory. Example 1 illustrates that weight-based properties of graphs may behave unpredictably when the support changes. Thus, pruning a pattern at a certain point bears the risk of missing elements of the result.

Example 1. Think of an upper-bound constraint defined as a numerical threshold t_u on the average weight of a certain edge $a{\rightarrow}b$ in all supporting graphs: $avg(a{\rightarrow}b) \leq t_\mathrm{u}$. This would prevent mining from expanding a pattern f where $avg(a{\rightarrow}b) > t_\mathrm{u}$. For the moment, suppose that f was expanded by one edge nevertheless, resulting in pattern f'. Then, fewer graphs in the database might support f'. Depending on the edge weights in the non-supporting graphs, the average weight of that edge in f' might decrease – f' might satisfy the constraint.

Despite this adverse characteristic, we study frequent subgraph mining with non-anti-monotone weight-based constraints in this paper. The rationale is that certain characteristics of real-world graphs give way to the expectation that results are good. Namely, there frequently is a correlation between the graph topology and the weights in real-world weighted graphs.

Example 2. Consider a road-map graph where every edge is attributed with the maximum speed allowed. Large cities, having a high node degree (a topological

property), tend to have more highway connections (high edge-weight values) than smaller towns. This is a positive correlation.

In software engineering, a similar observation holds: Think of a node in a weighted call graph representing a small method consisting of a loop. This method tends to invoke a few different methods only (low degree), but with high frequency (high weights). This is a negative correlation.

Our notion of *approximate constraint-based frequent subgraph mining* is as follows: Given a database of *weighted graphs*, find subgraphs satisfying a minimum frequency constraint and user-defined constraints referring to weights. Note that the subgraphs returned are unweighted – weights are considered only in the constraints. In this study, we investigate the following question:

Problem Statement. What is the completeness and the usefulness of results obtained from approximate weight-constraint-based frequent subgraph mining?

In concrete terms, we study the degree of *completeness* of mining results compared to non-constrained results. To assess the *usefulness* of an approximate result, we consider the result quality of higher-level analysis tasks, based on approximate graph-mining results as input.

To deal with this problem, this article features the following points:
Weight-Constraint-Based Mining. We say how to extend standard pattern-growth algorithms for frequent subgraph mining with pruning based on weight-based constraints. We do so for *gSpan* [22] and *CloseGraph* [23].
Application to Real-World Problems. We describe different data-analysis problems that make use of frequent subgraph mining of weighted graphs from domains as diverse as software engineering and logistics. We say how to employ weight-based constraints to solve these problems efficiently.
Evaluation. We report on the outcomes of a broad evaluation from very different domains and analysis settings. A fundamental result is that the correlation of weights with the graph structure indeed exists, and we can exploit it in real-world analysis problems. In particular, graph mining with constraints leads to a speedup of up to 3.5 with the same quality of results in classification and software-defect localization and 7 in explorative mining while obtaining satisfactory results.

Paper Organization: Section 2 contains preliminaries. Section 3 introduces weight-based constraints, Section 4 explains their integration into algorithms. Section 5 describes application settings, and Section 6 contains the evaluation. Section 7 discusses related work. Section 8 concludes.

2 Preliminaries

Definition 1. *A* labeled weighted graph *is a six-tuple:* $G := (V, E, L, l, W, w)$. V *is the set of vertices,* $E \subseteq V \times V$ *the set of edges,* L *a set of categorical labels,* $l : V \cup E \to L$ *a labeling function,* $W \subseteq \mathbb{R}$ *the domain of the weights and* $w : E \to W$ *a function which assigns weights.* $E(G)$ *denotes the set of edges etc.*

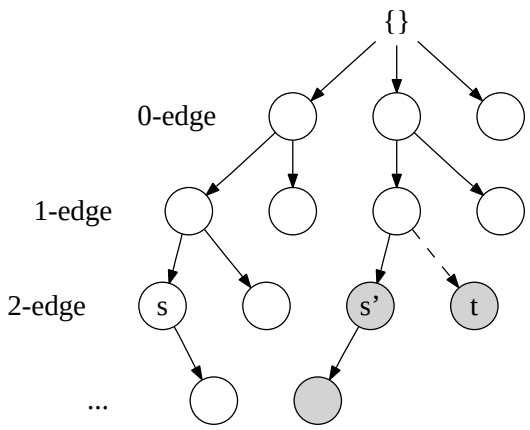


Fig. 2. A pattern-growth search space with conventional isomorphism-based pruning (s') and constraint-based pruning (t, new in this contribution).

All graphs can be directed or undirected ($e \in E$ ordered/unordered). All techniques discussed in this paper can easily be extended to cover weighted nodes ($w : V \cup E \rightarrow W$) and tuples of weights ($W \subseteq \mathbb{R}^n, n \in \mathbb{N}$).

Definition 2. Frequent subgraph mining *is the task of finding all subgraph patterns* $f \in F$ *with a support of at least* sup_{min} *in a graph database* $D := \{g_1, ..., g_{|D|}\}$. *The support of a subgraph* f *is* $support(f, D) := |\{g | g \in D \wedge f \subseteq g\}|$ *where '*$\subseteq$*' denotes a subgraph-supergraph relationship. In short,* $f \in F \iff support(f, D) \geq sup_{min}$.

Note that subgraph isomorphism considers labels but not weights.

Definition 3. Closed-graph-mining *algorithms discover only subgraph patterns which are closed. A graph* f *is closed if no other graph pattern* f' *is part of the result set* F *which has the same support and is a proper supergraph of* f *(*$f \subset f'$*).*

Closed mining algorithms produce more concise result sets and make use of pruning opportunities. *Pattern-growth-based algorithms* for (closed) frequent subgraph mining [24], such as *gSpan* [22] and *CloseGraph* [23], perform a depth-first search in a pattern-search space. Starting from some small graph pattern, they extend it by systematically adding edges and determining the support of the result. If it satisfies the $sup_{\min}$ criterion, the algorithm keeps extending it. Otherwise, the pattern is pruned, and the search backtracks.

Example 3. Figure 2 is a pattern-growth search space: The leaves cannot be extended further due to the $sup_{\min}$ criterion. s' is pruned as it is isomorphic to node s already discovered. The dashed edge with node t stands for constraint-based pruning, which we introduce in the following.

Definition 4. *A constraint c in* constraint-based mining *is a Boolean predicate which any $f \in F$ must fulfill. Formally, in constraint-based frequent subgraph mining, $f \in F \iff (support(f, D) \geq sup_{min} \wedge c(f) = true)$.*

Constraint predicates can be categorized into several classes. Most important is anti-monotonicity, as such constraints allow for effective pruning. (See Section 7 for further classes.)

Definition 5. *c is* anti-monotone $\iff (\forall f' \subseteq f : c(f) = true \Rightarrow c(f') = true)$

A prominent example of anti-monotone constraints is the frequency criterion: If a graph has a support of at least $sup_{\min}$, all its subgraphs have the same or a larger support. Therefore, anti-monotone constraints are the basis for all a priori and pattern-growth mining algorithms: They stop extending patterns when the current one does not satisfy the constraint, without missing any patterns. Most constraints based on weights are not anti-monotone [19].

3 Weight-Based Constraints

In this section, we define the weight-based constraints we investigate in this paper. We do not deal with anti-monotone constraints, since we are interested in investigating approximate mining results from non-anti-monotone constraints. However, the techniques would work with anti-monotone constraints as well.

Definition 6. *A* weight-based measure *is a function $E(p) \to \mathbb{R}$ which assigns every edge of a graph pattern p a numerical value. The function takes the weights of the corresponding edges in all embeddings of p in all graphs in D into account.*

Depending on the actual problem, one can assign some numerical or categorical value such as a class label to each graph. Measures like *InfoGain* and *PMCC* make use of such values, in addition to the weights. We discuss example measures later. – If labels are not unique, subgraphs can be embedded at several positions within a graph. We consider every single embedding of a subgraph to calculate a measure for an edge.

Definition 7. *A* lower bound predicate *c_l for a pattern p is a predicate with the following structure:*

$$c_l(p) := (\exists\, e_1 \in E(p) : measure(e_1) > t_l) \vee (|p| < size_{min})$$

An upper bound predicate *c_u in turn is as follows:*

$$c_u(p) := (\nexists\, e_2 \in E(p) : measure(e_2) > t_u) \vee (|p| < size_{min})$$

A weight-based constraint*, applied to a pattern p, is a set containing c_l, c_u, or both, connected conjunctively.*

The lower- and upper-bound predicates let the user specify a minimum and maximum interestingness based on the *measure* chosen. We comment on the two predicates as well as on parameter $size_{\min}$ in Section 4. Note that Definition 7 requires to consider *all* edges of a pattern p. This is necessary, as illustrated in Example 1. The value of the measure of *any* edge of p can change when the set of graphs supporting p changes.

Weight-Based Measures

Any function on a set of numbers can be used as a measure. We have chosen to evaluate three measures with a high relevance in real data-analysis problems – none of them is anti-monotone. Two of them, *InfoGain* and *PMCC*, require the existence of a class associated with each graph. Such classes are available, e.g., in any graph-classification task, and the goal of the mining process is to derive subgraph patterns for a good discrimination between the classes. We also use the *variance*, which does not depend on any class. It is useful in explorative mining scenarios where one is interested in subgraphs with varying weights.

Example 4. If one wants to search for patterns p with a certain minimum *variance* of weights, one would specify the measure '*variance*', the threshold value t_l and set $size_{\min}$ to 0. The constraint then is '$\exists\, e : variance(e) > t_l$'. This could be useful when analyzing logistics data, where one wants to find subgraphs with unbalanced load or highly varying transportation times.

Besides the measures described in the following, many further measures from statistics and data analysis can be used similarly to build weight-based constraints. This includes, say, different attribute-selection measures known from decision-tree induction [1, 16, 20].

Information Gain. The *InfoGain* [16], is a measure based on *entropy*. It is a value between 0 and 1 and quantifies the ability of an attribute A to discriminate between classes in a dataset (without a restriction to binary classes). It is frequently used in decision-tree induction and feature selection [16, 20]. In the context of weighted graphs, A refers to the weights of a certain edge of a subgraph pattern in all embeddings in all graphs in the graph database D.

Pearson's Product-Moment Correlation Coefficient (*PMCC*). The *correlation coefficient* is widely used to quantify the strength of the linear dependence between two variables. See [20] for a definition. In our graph-mining context, these two variables are the weight of a certain edge in a subgraph pattern in all embeddings in graphs in D and their binary classes. For our purposes, positive and negative correlation have the same importance, and we use the absolute value. Then *PMCC* is a value between 0 and 1 as well.

Variance. The *variance* quantifies the variation of the values of a random variable Y. It is a positive value without upper bound. In our scenarios, Y is the set of weights of a certain edge in all subgraph patterns in all embeddings in D.

4 Weight-Based Mining

We now describe how to integrate *weight-based constraints* into *pattern-growth-based frequent subgraph mining*. We first focus on vanilla pattern-growth algorithms before turning to closed mining. The basic idea is to use weight-based constraints – even if they are not anti-monotone – to prune the search space.

Example 5. Figure 2 illustrates pattern-growth mining with and without weight-based constraints. Without such constraints, s' and its successors are pruned, as s' is isomorphic to s. With weight-based constraints, the search is additionally pruned at pattern t. The dashed edge extends its parent, and t including the new edge violates a weight-based constraint. Note that it is not necessarily the newly added edge itself which violates the constraint, but any edge in t.

In concrete terms, we treat the lower and upper-bound predicates c_l and c_u (as defined in Definition 7) in weight-constraint-based mining as follows:

Approach. When a pattern p does not satisfy c_l or c_u, the search is pruned. If it is c_u that is not satisfied, p is added to the mining result, otherwise not.

The rationale behind an *upper bound* is to speed up mining by pruning the search when a sufficiently interesting edge weight is found. Therefore, we use it to prune the search, but save the current pattern. For example, if the user wants to use the graph patterns mined for classification, a pattern with one edge with a very discriminative weight will be fair enough. Clearly, larger graphs can still be more discriminative. Setting the threshold therefore involves a trade-off between efficient pruning and finding relevant graphs. Section 6.3 will show that small changes in the upper bound do not change the results significantly. It is therefore sufficient to rely on few different threshold values to obtain satisfactory results. With a *lower bound*, the user specifies a minimal interestingness. This bound stops mining when the value specified is not reached. The rationale is that one does not expect to find any patterns which are more interesting. However, this might miss patterns. The parameter $size_{\min}$ (cf. Definition 7) controls this effect.

Pattern-Growth Algorithms. Algorithm 1 describes the integration into pattern-growth-based frequent subgraph mining algorithms such as *gSpan* [22]. The algorithm works recursively, and the steps in the algorithm are executed for every node in Figure 2. Lines 1–2, 9–13 and 20 are the generic steps in pattern-growth-based graph mining [24]. They perform the isomorphism test (Lines 1–2), add patterns to the result set (Line 9) and extend the current pattern (Line 11), leading to a set of frequent patterns P. The algorithm then processes them recursively (Lines 12–13) and stops depth-first search when P is empty (Line 20).

Lines 4–7 and 15–17 are new in our extension. Instead of directly adding the current pattern into the result set, the algorithm first checks the $size_{\min}$ parameter (Line 4). Only if the minimum size is reached, it calculates the weight-based measures (Line 5). Line 7 checks the constraints (if c_l or c_u is not set, the thresholds are zero or ∞, respectively; cf. Definition 7). If they are not violated, or the minimum size is not reached, the algorithm saves the pattern to the

Algorithm 1 *pattern-growth*$(p, D, sup_{\min}, t_{\mathrm{l}}, t_{\mathrm{u}}, size_{\min}, F)$

Input: current pattern p, database D, $sup_{\min}$, parameters *measure*, t_{l}, t_{u} and $size_{\min}$
Output: result set F

```
 1: if p ∈ F then
 2:    return
 3: end if
 4: if |p| ≥ size_min then
 5:    calculate weight-based measures for all edges
 6: end if
 7: if (∃ e1 : measure(e1) > t_l ∧ ∄ e2 : measure(e2) > t_u) ∨ (|p| < size_min) then
 8:    if (algorithm ≠ CloseGraph ∨ p is closed) then
 9:       F ← F ∪ {p}
10:    end if
11:    P ← extend-by-one-edge(p, D, sup_min)
12:    for all p' ∈ P do
13:       pattern-growth(p', D, sup_min, t_l, t_u, size_min, F)
14:    end for
15: else
16:    if ∃ e : measure(e) > t_u then
17:       F ← F ∪ {p}
18:    end if
19: end if
20: return
```

result set (Line 9) and continues as in generic pattern growth (Lines 12–13). Otherwise, the algorithm prunes the search, i.e., it does not continue the search in that branch. Note that this step is critical, as it determines both the speedup and the result quality. As mentioned before, we always save the last pattern before we prune due to upper bounds (Lines 16–17). This leads to result sets which are larger than those from standard graph mining when the constraints are applied in a postprocessing step.

One can realize constraints on more than one measure in the same way, by evaluating several constraints instead of one, at the same step of the algorithm. As mentioned before, mining with weight-based constraints produces a result set with unweighted subgraph patterns. In case one needs weighted subgraphs in the result set, arbitrary functions, e.g., the average, can be used to derive weights from the supporting graphs in the graph database.

Closed Mining. Closed mining returns closed graph patterns only. When dealing with weight-based constraints, we deviate from this characteristic. We favor graphs which are interesting (according to the measures) over graphs which are closed. This is because the weight-based constraints might stop mining when 'interesting enough' patterns are found. Extending the *CloseGraph* [23] algorithm is slightly more complicated than pattern growth as described before. *CloseGraph* performs further tests in order to check for closedness (Line 8 in Algorithm 1). In our extension, these tests are done after weight-based pruning. Therefore, when the search is pruned due to a constraint, it might happen that the algorithm

misses a larger closed pattern. In this case it adds patterns to the result set which are not closed.

Implementation. The extensions we describe here are compatible with any pattern-growth graph miner. We for our part use the *ParSeMiS* graph-mining suite [15] with its *gSpan* [22] and *CloseGraph* [23] implementations.[1]

5 Weighted Graph Mining Applied

We now say how to exploit the information contained in the weights of graphs in different application scenarios building on weight-constraint-based frequent subgraph mining.

Software-Defect Localization. The purpose of defect localization is to help software developers finding defects.[2] In our case, the result is a list of suspicious methods, sorted by their likelihood to contain a defect. A developer can then inspect the code starting with the top-ranked method. More precisely, we focus on *non-crashing occasional bugs*, which are notoriously difficult to find. *Crashing bugs* in turn would be relatively easy, as stack traces are available. *Occasional bugs* are hard to localize as they only occur with some input data. Our approach builds on the comparison of *weighted call graphs* (cf. Figure 1), representing different executions of the same program. Every graph is labeled as *failing* or *correct*, depending on whether the program execution has returned a false or a correct result. As any approach for defect localization, the method described in the following cannot discover any kind of bug. It can however detect defects of a frequently occurring category: defects leading to infections that influence the control structure of a program, i.e., those changing the call-graph structure or a call frequency (an edge weight).

In order to obtain the likelihood of a method to contain a defect, we look at two kinds of evidence: weight-based measures and subgraph structures. Firstly, we consider the measure of edges, computed by the constraint-based-mining algorithm. In our implementation, a method (represented as a node) inherits the normalized maximum value from all outgoing edges in all patterns in the result as its weight-based likelihood:

$$P_{\mathrm{w}}(m) := \mathit{normalize}(\max(\mathit{measure}(\{(m,x)|(m,x) \in \mathcal{E} \wedge x \in \mathcal{V}\})))$$

where $\mathcal{V}$ and $\mathcal{E}$ are the unions of the vertex and edge sets of all subgraph patterns in the result set, and *measure* applied to a set calculates the *measure* of every element separately. We use the maximum, as one node might have many outgoing edges which are not related to the defect at all. From preliminary experiments with different measures applied in a postprocessing setting, we know that entropy-based measures such as *InfoGain* are best suited for defect localization.

[1] We provide our extensions for *ParSeMiS* online:
`http://sdqweb.ipd.kit.edu/wiki/ParSeMiS-Extensions`

[2] We use the following terminology [26]: *Defects* in source code can lead to an *infected* program state, which ultimately might become visible as a *failure*.

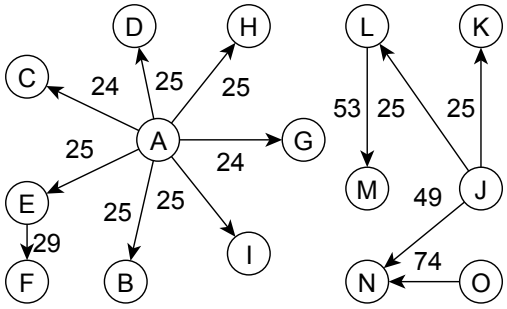


Fig. 3. Two typical fragments from a small unconnected graph in the logistics dataset.

Secondly, we look at the subgraph structures. The result sets mined with weight-based constraints let us define another likelihood based on support. They contain a higher number of interesting graphs with interesting edges (according to the measure chosen) than a result set from vanilla graph mining. Therefore, it seems promising not only to give a high likelihood to edges with interesting weights. We additionally consider nodes (methods) occurring frequently in the graph patterns in the result set. We calculate this structural likelihood similar to a support in the result set F:

$$P_{\mathrm{s}}(m) := \frac{|\{f | f \in F \wedge m \in f\}|}{|F|}$$

The next step is to combine the two likelihoods. We do this by averaging the normalized values. Preliminary experiments have shown that using these two kinds of evidence yields a more precise localization of defects.

Finding out how well defect-localization techniques perform requires an evaluation measure. Defining such a measure is not difficult, as we, the experimentators, know the defects. We use the position of the real defect in the generated list of suspicious methods. This position quantifies the number of methods to look into in order to find the defect.

Weighted-Graph Classification. Subgraph patterns from weighted graphs cannot directly be used for classification. With unweighted graphs, it is common to use binary feature vectors, indicating which subgraph is included in a graph [10]. Every such vector corresponds to a graph in the graph database. In the following, we explain how we assemble feature vectors including weights to use them for classification. We use one feature in the vector for every edge in every frequent subgraph mined. These features are numerical and stand for the corresponding weight in the original graph. If a graph does not contain a certain subgraph, the corresponding features are null values.

Example 6. We construct a feature vector for the graph in Figure 3. Imagine that there are two frequent subgraphs, $A{\rightarrow}E{\rightarrow}F$ and $L{\rightarrow}M$. The vector consists of the values of the edges $A{\rightarrow}E$, $E{\rightarrow}F$ and $L{\rightarrow}M$: $(25, 29, 53)$.

In cases where labels in the subgraph patterns are not unique, the position of an edge in a subgraph describes a certain edge. In case of multiple embeddings of a pattern, we use aggregates of the weights from all embeddings. This encoding allows to analyze every edge weight in the context of every subgraph.

Finally, any classifier featuring numerical attributes and null values can work with the vectors to learn a model or to make predictions. Arbitrary evaluation measures for classification can quantify the predictive quality of the weighted-graph-classification problem. We for our part use the established measures *accuracy* and *AUC* (area under the *ROC curve*; see, e.g., [20]).

Explorative Mining. Besides automated analysis steps following graph mining, another important application is explorative mining. Here, the results are interpreted directly by humans. One is interested in deriving useful information from a dataset. In our weight-constraint-based scenario, such information is represented as subgraphs with certain edge-weight properties in line with the constraints. For instance, the logistics dataset is well suited for explorative mining. As motivated in Example 4, one might be interested in subgraphs featuring edges with high or low variance.

Evaluation in this context is difficult, as it is supposed to provide information for humans. Therefore, it is hard to define a universal measure. In this study, we focus on basic properties of the dataset mined, in particular the size of the subgraphs. This size can be seen as a measure of expressiveness, as larger subgraphs tend to be more significant.

6 Experimental Evaluation

We now investigate the characteristics of pruning with non-anti-monotone constraints, given several real-world analysis problems. We do so by comparing different application-specific quality criteria with the speedup in runtime as well as by assessing the completeness of approximate result sets. While other solutions to the real-world problems (without weighted graph mining) might be conceivable as well, studying them is not the concern of this article. Furthermore, we do not aim at demonstrating that the analysis of weighted graphs is beneficial compared to non-weighted graphs. Other studies have shown the adequateness of weighted graphs for analysis problems in various domains [4, 9, 12]. We first describe the datasets in Section 6.1. We then present the experimental settings in Section 6.2 and the results in Section 6.3.

6.1 Datasets

Software-Defect Localization. We investigate the dataset from [5], which consists of classified *weighted call graphs*. In concrete terms, the dataset consists of 14 defective versions of a Java diff tool taken from [3]. Every version was executed exactly 100 times with different input data, resulting in roughly the same number of graphs representing *failing* and *correct* executions.[3] The graphs

[3] We provide the defective programs and the graphs used online:
`http://www.ipd.kit.edu/~eichi/papers/eichinger10on/`

are quite homogeneous; the following numbers describe one of the 14 datasets. The mean number of nodes is 19.6 (standard deviation $\sigma = 1.9$), the mean number of edges is 23.8 ($\sigma = 4.6$), but the edge weights are quite diverse with a mean value of 227.6 ($\sigma = 434.5$).

Logistics. This dataset is the one from [9]. It is origin-destination data from a logistics company, attributed with different information. The graphs are as follows: Transports fall into two classes with full truckload (TL) and less than truckload (LTL). The transports from the two classes form two sets of graphs, which we label accordingly. We further arrange transports (edges) with a similar weight of the load in one graph. Next, as the spatial coordinates in the dataset are fine grained, we combine locations close to each other to a single node, e.g., locations from the same town. We use the time needed to get from origin to destination as edge weight. The duration is a crucial parameter in transportation logistics, and there is no obvious connection to the class label. The dataset describes a weighted-graph-classification problem, i.e., predict if a graph contains fully or partly-loaded transports.

Finally, the dataset consists of 51 graphs. The two class labels are evenly distributed, the mean number of nodes is 234.3 ($\sigma = 517.1$), and the mean number of edges is 616.1 ($\sigma = 2418.6$). As indicated by the high standard deviations, this is a very diverse dataset, containing some very large graphs. The large graphs are not problematic for mining algorithms in this case, as most graphs are unconnected, and the fragments are quite small. Besides heterogeneous structural properties, the edge weights with a mean value of 73.2 ($\sigma = 50.9$) are quite close to each other. Figure 3 is a part of one of the logistics graphs.

6.2 Experimental Settings

In our experiments we compare a regular *CloseGraph* implementation to ours with weight-based constraints. In preliminary experiments, *CloseGraph* performed much better than *gSpan* with our datasets while generating subgraphs of the same predictive quality. We evaluate the quality of the results with scenario-specific evaluation measures (cf. Section 5) along with the runtime. We use a single core of an AMD Opteron 2218 with 2.6 GHz and 8 GB RAM for all experiments. We mine with a $sup_{\min}$ of 3 in all experiments with the defect-localization dataset and with a $sup_{\min}$ of 8 in all experiments with the logistics data. We set the $size_{\min}$ to 0 in all experiments, as we are interested in the pure results with the different lower and upper bounds.

Software-Defect Localization. In this scenario, we compare our results based on edge-weight-based pruning with a vanilla graph-mining technique. To be fair, we repeat the experiments from [5] with slight revisions[4] and the same $sup_{\min}$ (3). We use upper-bound constraints on the two class-aware measures.

Weighted-Graph Classification. For classification experiments, we use both datasets. In the software-defect-localization dataset, we predict the class labels

[4] In [5], a zero in the feature vectors indicates that a certain call does not occur. We now use null values, as this allows for a fair comparison to our new approach.

failing or *correct*, in the logistics dataset the truck-load labels *TL* and *LTL* (cf. Section 6.1). We mine the graph databases with different upper-bound-constraint thresholds on the two class-aware measures and assemble feature vectors, as described in Section 5. We then use them along with the corresponding class labels in a 10-fold-cross-validation setting with standard algorithms. In concrete terms, we use the *Weka* implementation [20] of the C4.5 decision tree classifier [16] and the *LIBSVM* support-vector machine [2] with standard parameters. For scalability reasons, we employ a standard *chi-squared feature-selection* implementation [20] for dimensionality reduction before applying *LIBSVM*.

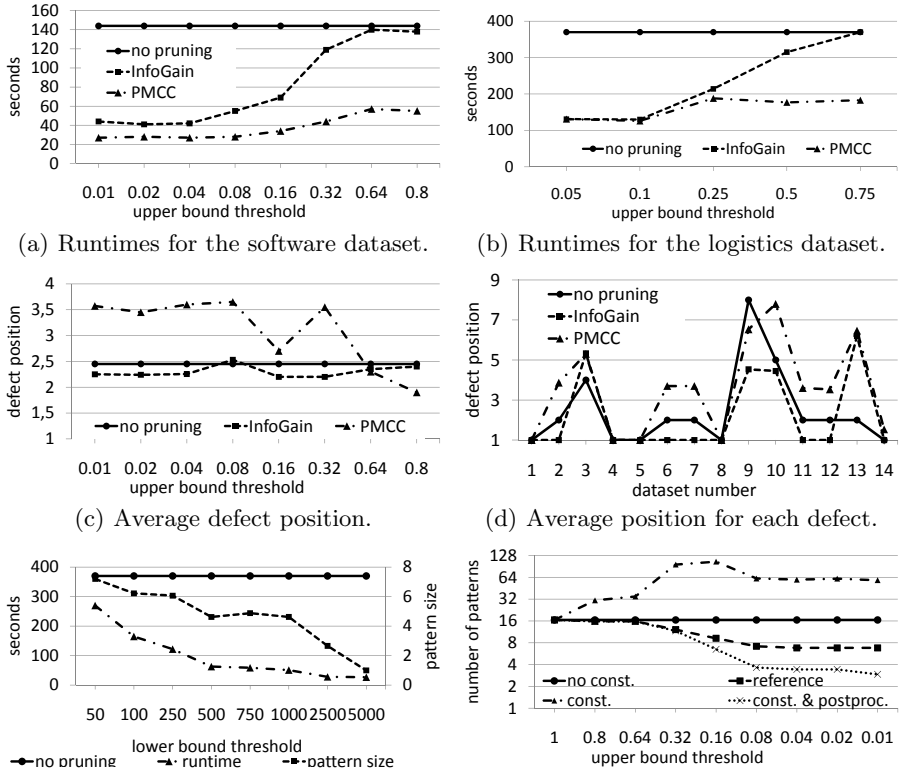
Explorative Mining. For explorative-mining experiments, we investigate different lower-bound-constraint thresholds on *variance* in the logistics dataset. We compare their quality and runtime with mining runs without constraints.

6.3 Experimental Results

Software-Defect Localization. Figure 4(a) displays the runtimes of *InfoGain* and *PMCC* with different upper-bound thresholds on all 14 versions of the dataset. The *InfoGain* constraint is always faster than the execution time without pruning, irrespective of the threshold. For low threshold values (0.01 to 0.04), *InfoGain* reaches speedups of around 3.5. *PMCC* in turn always performs better than *InfoGain*, and reaches speedups of up to 5.2. This is natural, as the calculations to be done during mining in order to derive the measures are more complicated for *InfoGain* (involving logarithms) than for *PMCC*. For high thresholds (0.32 to 0.8) on both measures, the runtime increases significantly. This is caused by less pruning with such thresholds.

Figure 4(c) contains the results in defect localization without pruning and with *InfoGain* and *PMCC* pruning with various upper bounds. The figure shows the average position of the defect in the returned ranking of suspicious methods, averaged for all 14 versions. The *InfoGain* almost always performs a little bit (a fifth ranking position for the two lowest thresholds) better than the baseline ('no pruning'). As the baseline approach uses *InfoGain* as well, we explain this effect by the improved structural likelihood computation (P_s, cf. Section 5), which takes advantage of the edge-weight-based pruning. The *PMCC* curve is worse in most situations. This is as expected, as we know that entropy-based measures perform well in defect localization (cf. Section 5). Figure 4(d) contains the defect-localization results for the 14 different versions. We use the average of the three executions with the best runtime (thresholds 0.01 to 0.04). The figure reveals that the precision of localizations varies for the different defects, and the curve representing the *InfoGain* pruning is best in all but two cases. Concerning the threshold values, observe that small changes always lead to very small changes in the resulting defect-localization precision, with mild effects on runtime.

Next to the defect-localization results, the performance of classifiers learned with the software dataset is very high. The values with *InfoGain*-pruning only vary slightly for the different thresholds on both classifiers, the SVM (*accuracy*: 0.982–0.986; *AUC*: 0.972–0.979) and the decision tree (*accuracy*: 0.989–0.994; *AUC*: 0.989–0.994). Although the variance is very low, higher thresholds yield



(a) Runtimes for the software dataset.

(b) Runtimes for the logistics dataset.

(c) Average defect position.

(d) Average position for each defect.

(e) Runtimes & quality, logistics dataset.

(f) Comparison of approximate result sets.

Fig. 4. Experimental results.

slightly higher values in most cases. This is as expected, as less pruning leads to larger graphs, encapsulating potentially more information. With *PMCC*, the values are very close to those before, and one can make the same observation.

Logistics. Figure 4(b) shows the runtimes of both measures with different upper-bound thresholds. With an upper bound of up to 0.10 on *InfoGain* or *PMCC*, our extension runs about 2.9 times faster than the reference without pruning. For larger upper bounds on *PMCC*, graph mining with our extension still needs only half of the runtime. *InfoGain* becomes less efficient for larger values, and for a high threshold of 0.75 it needs the same time as the algorithm without edge-weight-based pruning. As before, *PMCC* performs better than *InfoGain*.

In the experiments, the performance of classifiers does not depend on the upper bound, independent of the threshold. We evaluated the same values as in Figure 4(b). For the *InfoGain* measure, *accuracy* and AUC of the SVM are 0.902 and 0.898, and they are a little lower with the decision tree: 0.863 and 0.840. For *PMCC*, the results are the same for most upper bounds. Only for the bounds 0.50

and 0.75, where less pruning takes place and more subgraphs are generated, the results are slightly better (decision tree only). Next to classification performance, the runtimes change only slightly when the threshold values change.

These results demonstrate that the edge weights in this dataset are well suited for classification. Further, the degree of edge-weight-based pruning did not influence the results significantly. Therefore, *InfoGain* and *PMCC* obviously are appropriate measures. With low upper-bound values on both measures, the runtime can be improved by a factor of about 2.9, while the classifiers have almost the same quality. On the other side, these results also show that the graph structure of this particular dataset is less important to solve the classification problem than the edge weights.

Besides the performance of classification, we also evaluate the *variance* measure in an explorative mining setting on the logistics dataset. Figure 4(e) shows the runtimes with several lower bounds along with the corresponding averaged subgraph-pattern sizes (in edges) in the result set. At the lowest threshold (50), the runtime already decreases to 73% of the runtime without pruning. At the highest value (5,000), the runtime decreases to 7% only, which is a speedup of 13. At the same time, the average subgraph size decreases from 7 to 1. Therefore, values between 250 and 1,000 might be good choices for this dataset (depending on the user requirements), as the runtime is 3 to 7 times faster, while the average subgraph size decreases moderately from 7.4 to 6.1 and 4.6.

Completeness of Approximate Result Sets. We now investigate the completeness of our result sets and look at the defect-localization experiments with *InfoGain*-constraints another time. Figure 4(f) refers to these experiments with the approximate constraint-based *CloseGraph* algorithm, but displays the sizes of result sets (averaged for all 14 versions). We compare these results with a non-approximate reference, obtained from a non-constrained execution, where we remove all subgraph patterns violating an upper bound afterwards. Our constraint-based mining algorithms save all patterns violating upper bounds before pruning the search (cf. Section 4). For comparison, we apply the same postprocessing as with the reference and present two variants of constraint-based mining: The pure variant ('const.') and the postprocessed one ('const. & postproc.'). Comparing the two postprocessed curves, for thresholds of 0.64 and larger, constraint-based result sets have the same size as the reference and are smaller for thresholds of 0.32 and lower. Preliminary experiments with different $sup_{\min}$ values have revealed that the difference between the curves decreases ($sup_{\min}$ of around 20 instead of 3) or vanishes ($sup_{\min}$ of 70). The pure result sets (those we used in the experiments before), are always larger than closed mining, even if no constraints are applied. To conclude, our approximate result sets contain less than half of the patterns as the non-approximate reference, for small $sup_{\min}$ and upper bound values. However, the pure result sets obtained from constraint-based mining in a shorter runtime (cf. Figure 4(a)) contain many more interesting subgraph patterns (see curve 'const.'), which is beneficial for the applications.

7 Related Work

Weighted-Graph Mining. Even though weighted graphs are ubiquitous in the real world, we are only aware of a few studies analyzing weighted graphs with frequent subgraph mining. They focus on the specific analysis problem, rather than proposing general weighted graph mining techniques.

Jiang et al. [9] consider logistic networks where edges represent single transports and are annotated with several weights such as distance between two nodes and the weight of the load. With each weight, a different weighted graph can be constructed. In order to derive labels for graph mining from the edge weights, the authors use a binning strategy. Each weight is partitioned into ranges of the same size, giving a few (7 to 10) distinct labels. The binning strategy for discretization may curb result accuracy, for two reasons: (1) The particular scheme does not take the distribution of values into account. Thus, close values may be assigned to different bins. (2) The discretization leads to a number of ordered (ordinal) intervals, but the authors treat them as unordered categorical values. For example, the information that 'medium' is between 'small' and 'large' is lost.

Nowozin et al. [12] do discretization as well before it comes to frequent subgraph mining. They study image-analysis problems, and images are represented as weighted graphs. The authors represent each point of interest by one vertex and connect all vertices. They assign each edge a vector consisting of image-analysis-specific measures. Then they discretize the weights, but with a method more sophisticated than binning. The weight vectors are clustered, resulting in categorical labels of edges with similar weight vectors. However, the risk of losing potentially important information by discretization is not eliminated: (1) It might still happen that close points in an n-dimensional space fall into different clusters. (2) Even when value distributions are considered, the authors do so in the context of the original graphs. When frequent subgraph mining is applied afterwards, the distributions within the different subgraphs can be very different, and other discretizations could be more appropriate.

Eichinger et al. [5] have proposed a postprocessing approach. They use weighted call graphs for software-defect localization, representing correct and failing program executions. The edge weights represent call frequencies (cf. Figure 1). The authors first assemble a table containing every edge weight in every subgraph mined. They then use feature selection to obtain an ordered list of edges representing potentially erroneous pieces of code. This approach avoids the shortcomings of discretization and analyzes numerical weights instead of discrete intervals. However, the two steps of graph mining and feature selection are executed sequentially. This gives way to further improvements in terms of runtime when the features selected later are mined directly. In this paper we investigate such an approach, based on approximate weight-based constraints.

Constraint-Based Mining. Pioneering work [11] has introduced constraints for frequent itemset mining. The authors define the two constraint properties *anti-monotonicity* (cf. Section 2) and *succinctness*. Both help to speed up mining. [13] has introduced *convertible constraints* for itemsets, focusing on aggregate constraints. [6, 14] has carried forward constraint-based mining to sequences.

More recently, constraint-based graph mining has been proposed. Wang et al. [19] build on the constraint classes introduced in [11], extended by *monotone constraints*, and categorize various graph-based constraints into these classes. Then the authors develop a framework to integrate the different constraint classes into a pattern-growth graph-mining algorithm. They use *anti-monotone constraints* to prune the search space and *monotone constraints* to speed up the evaluation of further constraints. Further, they use the *succinctness* property to reduce the size of the graph database. Wang et al. also propose a way to deal with some weight-based constraints. For the average-weight constraint, they propose to omit nodes and edges with outlier values from the graphs in the database. They do so to shrink the graph size and to avoid the evaluation of such 'unfavorable' elements. This can lead to incomplete result sets. Furthermore, situations where such constraints lead to significant speedups are rare, according to the evaluation of the authors with one artificial dataset, and they do not make any statements regarding result quality. In [27], Zhu et al. extend [19] by refining the classes of constraints and integrating them into mining algorithms, but they do not consider weights.

Although the techniques proposed work well with *monotone*, *anti-monotone* or *succinct* constraints and their derivations, most weight-based constraints, as defined in Section 3, do not fall into these categories [19]. (See Example 1 for an illustration.) They are not *convertible* as well, even if such constraints might seem to be similar. The weights considered in convertible constraints stay the same for every item in all transactions, while weights in graphs can be different in every graph in D. Therefore, the established constraint-based-mining schemes cannot use weight-based constraints for pruning while guaranteeing completeness.

Mining Significant Graph Patterns. In many settings, frequent subgraph mining is followed by a feature-selection step. This is to ease subsequent processes such as classification and to identify the most significant features. The different proposals use various objective functions for feature selection. Work such as [21] has identified this two-step approach of mining and selecting to be the computational bottleneck in many graph-mining applications. On the one side, generating large numbers of frequent subgraphs to choose from is expensive. On the other side, the selection process can be expensive as well. Recent studies to investigate scalable algorithms demonstrate this [7, 10, 17, 18, 21]. They deal with the direct mining of patterns satisfying an objective function, instead of following the two-step approach. In other words, the subgraph sets mined might be incomplete with regard to the frequency criterion, but contain all (or most) graphs with regard to some other objective function. One can consider these functions to be constraints, as they narrow down the mining results. But they do not necessarily fall into any of the constraint classes mentioned before. Objective functions are either based on their ability to discriminate between classes or numerical values associated with the graphs [10, 18] or on some topological similarity measures [7, 17, 21]. To sum up, various researchers have studied scalable mining of graph patterns, with much success. However, they have not taken weights into account.

In our work, we use measures building on edge weights as objective functions, to decide which graphs are significant. The usage of weights allows for a more detailed analysis, as compared to only the graph structure. Like the previous approaches, ours does not necessarily produce graph sets which are complete with regard to frequency or some other hard constraint.

8 Conclusions

In this paper we have dealt with mining of weighted graphs, which are ubiquitous in the real world. The analysis of weights in addition to the graph structure bears the potential of more precise mining results. We have integrated non-anti-monotone constraints based on weights into pattern-growth frequent subgraph mining algorithms. This leads to improved runtime and approximate results. The goal of our study was to investigate the quality of these results. Besides an assessment of result completeness, we have evaluated its usefulness, i.e., the result quality of higher-level real-world analysis problems based on this data.

Our study shows that a correlation of weights with the graph structure exists and can be exploited. Frequent subgraph mining with weight-based constraints has proven to be useful – at least for the problems investigated. With the software dataset, we have obtained speedups of 3.5. This allows for analyses of larger software projects. At the same time, the results in defect localization even are a little more precise, and the classification performance is stable. In the logistics dataset, we have achieved a speedup of 2.9 while obtaining the same classification performance. In explorative mining, the speedup is around 7 while obtaining good results.

Acknowledgments

We thank Zahir Balaporia (Schneider National, Inc.) and Chris Clifton (Purdue University) for providing us with the logistics dataset [9].

References

1. C. Borgelt. A Decision Tree Plug-In for DataEngine. In *European Congress on Intelligent Techniques and Soft Computing (EUFIT)*, 1998.
2. C.-C. Chang and C.-J. Lin. *LIBSVM: A Library for Support Vector Machines.* Available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`.
3. I. F. Darwin. *Java Cookbook.* O'Reilly, 2004.
4. F. Eichinger and K. Böhm. Software-Bug Localization with Graph Mining. In C. C. Aggarwal and H. Wang, editors, *Managing and Mining Graph Data.* Springer, 2010.
5. F. Eichinger, K. Böhm, and M. Huber. Mining Edge-Weighted Call Graphs to Localise Software Bugs. In *ECML PKDD*, 2008.
6. M. N. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential Pattern Mining with Regular Expression Constraints. In *VLDB*, 1999.

7. M. A. Hasan, V. Chaoji, S. Salem, J. Besson, and M. J. Zaki. ORIGAMI: Mining Representative Orthogonal Graph Patterns. In *ICDM*, 2007.
8. A. Inokuchi, T. Washio, and H. Motoda. Complete Mining of Frequent Patterns from Graphs: Mining Graph Data. *Mach. Learn.*, 50(3):321–354, 2003.
9. W. Jiang, J. Vaidya, Z. Balaporia, C. Clifton, and B. Banich. Knowledge Discovery from Transportation Network Data. In *ICDE*, 2005.
10. T. Kudo, E. Maeda, and Y. Matsumoto. An Application of Boosting to Graph Classification. In *NIPS*, 2004.
11. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory Mining and Pruning Optimizations of Constrained Associations Rules. In *SIGMOD*, 1998.
12. S. Nowozin, K. Tsuda, T. Uno, T. Kudo, and G. Bakir. Weighted Substructure Mining for Image Analysis. In *Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2007.
13. J. Pei, J. Han, and L. V. S. Lakshmanan. Pushing Convertible Constraints in Frequent Itemset Mining. *Data Min. Knowl. Discov.*, 8(3):227–252, 2004.
14. J. Pei, J. Han, and W. Wang. Mining Sequential Patterns with Constraints in Large Databases. In *CIKM*, 2002.
15. M. Philippsen et al. *ParSeMiS: The Parallel and Sequential Mining Suite.* Available at `http://www2.informatik.uni-erlangen.de/EN/research/ParSeMiS/`.
16. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, 1993.
17. S. Ranu and A. K. Singh. GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In *ICDE*, 2009.
18. H. Saigo, N. Krämer, and K. Tsuda. Partial Least Squares Regression for Graph Mining. In *KDD*, 2008.
19. C. Wang, Y. Zhu, T. Wu, W. Wang, and B. Shi. Constraint-Based Graph Mining in Large Database. In *Asia-Pacific Web Conf. (APWeb)*, 2005.
20. I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations.* Morgan Kaufmann, 2005.
21. X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining Significant Graph Patterns by Leap Search. In *SIGMOD*, 2008.
22. X. Yan and J. Han. gSpan: Graph-Based Substructure Pattern Mining. In *ICDM*, 2002.
23. X. Yan and J. Han. CloseGraph: Mining Closed Frequent Graph Patterns. In *KDD*, 2003.
24. X. Yan and J. Han. Discovery of Frequent Substructures. In D. J. Cook and L. B. Holder, editors, *Mining Graph Data.* Wiley, 2006.
25. X. Yan, F. Zhu, P. S. Yu, and J. Han. Feature-Based Similarity Search in Graph Structures. *Trans. Database Syst.*, 31(4):1418–1453, 2006.
26. A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann, 2009.
27. F. Zhu, X. Yan, J. Han, and P. S. Yu. gPrune: A Constraint Pushing Framework for Graph Pattern Mining. In *PAKDD*, 2007.