J. Yaghob (Ed.): ITAT 2015 pp. 51–58
Charles University in Prague, Prague, 2015

A Versatile Algorithm for Predictive Graph Rule Mining

Karel Vaculík

KD Lab, FI MU Brno, Czech Republic
xvaculi4@fi.muni.cz

Abstract: Pattern mining in dynamic graphs has received a lot of attention in recent years. However, proposed methods are typically limited to specific classes of patterns expressing only a specific types of changes. In this paper, we propose a new algorithm, DGRMiner, which is able to mine patterns in the form of graph rules capturing various types of changes, i.e. addition and deletion of vertices and edges, and relabeling of vertices and edges. This algorithm works both with directed and undirected dynamic graphs with multiedges. It is designed both for the single-dynamic-graph and the set-of-dynamic-graphs scenarios. The performance of the algorithm has been evaluated by using two real-world and two synthetic datasets.

1 Introduction

Data mining of complex structures has been extensively studied for quite a while. Recently, more attention has been drawn to the area of dynamic graphs, i.e. graphs evolving through time. A lot of research has been carried out both from the global and local perspectives of dynamic graphs. For instance, global characteristics such as density and diameter were studied in real dynamic graphs in [6]. On the other hand, graph mining on the local level is most frequently focused on pattern mining. Such patterns are represented by subgraphs and their evolution through a short period of time [1, 2, 8].

Nevertheless, most of the pattern mining methods for dynamic graphs impose various restrictions, such as the type of the dynamic graphs or the type of changes captured by such patterns. For example, GERM algorithm [1] assumes that vertices and edges are only added and never deleted in the input dynamic graph. Furthermore, it mines patterns representing only edge additions.

In this paper, we propose a new algorithm DGRMiner for mining frequent patterns that can capture various changes in dynamic graphs. Specifically, the patterns are in the form of predictive rules expressing how a subgraph can be changed into another subgraph by adding new vertices and edges, deleting specific vertices and edges, or relabeling vertices and edges. An example of a dynamic graph and two predictive rules are illustrated in Fig. 1.

The algorithm is able to mine patterns from a single dynamic graph and also from a set of dynamic graphs. Such graph rules are useful for prediction in dynamic graphs, they can be used as pattern features representing dynamic graphs or simply for gaining an insight into internal processes of the graphs.

The paper is organized as follows. Section 2 gives the necessary definitions and presents the representation of dynamic graphs. Section 3 then provides a short description of the gSpan algorithm [13], whose framework is employed in our new algorithm. In Section 4, we describe the new algorithm for graph rule mining. Experimental evaluation is presented in Section 5. Finally, related work and conclusion can be found in Section 6 and 7, respectively.

2 Predictive Graph Rules

In this section, we provide definitions of a dynamic graph and predictive rules. Definitions of significance measures support and confidence are also part of this section.

2.1 Dynamic Graphs

Before defining a dynamic graph, we need to consider the notion of a static graph. In this paper, by a static graph we will denote a directed labeled multigraph without loops and with a restriction that no two edges with the same source and target vertices can have the same label. The proposed mining algorithm, however, can work with undirected edges too. For the sake of simplicity, we will restrict ourselves only to directed graphs in this section.

Definition 1 (Static graph)**.** *A* static graph *is a 5-tuple $G = (V_G, E_G, f_G, l_{G,V}, l_{G,E})$, where V_G is a set of vertices, E_G is a set of edges, $f_G : E_G \to V_G \times V_G$ is a map assigning a pair of vertices (u,v), $u \neq v$, to every edge, $l_{G,V}$ and $l_{G,E}$ are two maps describing labeling of the vertices and edges, respectively. Furthermore, $\forall e_1, e_2 \in E_G(f(e_1) = f(e_2) \Rightarrow l_E(e_1) \neq l_E(e_2))$.*

A dynamic graph is then given by a finite sequence of static graphs in which no two adjacent graphs are identical as we want to capture only the changes in the dynamic graph. Moreover, we extend each static graph G by *timestamp* functions $t_{G,V} : V_G \to T$ and $t_{G,E} : E_G \to T$, which map each vertex and edge to a point in time. We will work with discretized time and thus T will be the set of integers, i.e. $T = \mathbb{Z}$.

Definition 2 (Dynamic graph)**.** *A* dynamic graph *is a finite sequence $DG = (G_1, G_2, ..., G_n)$, where G_i is a static graph extended by timestamp functions $t_{G_i,V}$, $t_{G_i,E}$ for all $1 \leq i \leq n$, and $G_j \neq G_{j+1}$ for all $1 \leq j \leq n\text{-}1$. Graph G_i is referred to as the* snapshot *of DG at time i. In addition, for each $1 \leq i \leq n$, the timestamp functions $t_{G_i,V}$, $t_{G_i,E}$ assign to*

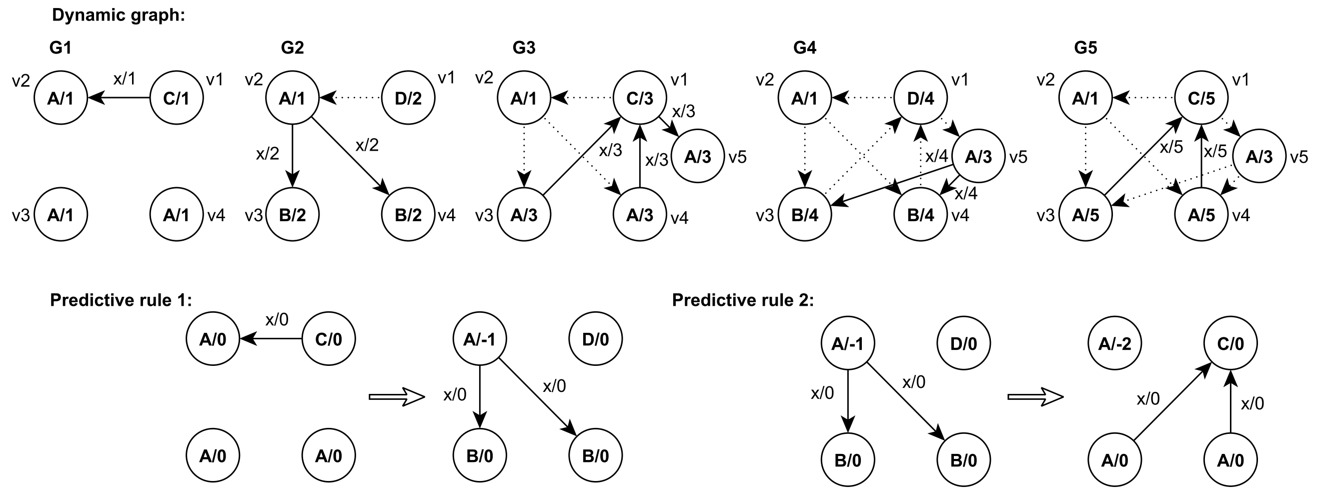

Figure 1: An example of a dynamic graph and two predictive graph rules. Numbers after slash symbols represent timestamps and dotted edges represent deleted edges.

each vertex and edge the time from which they have their current label, i.e. $t_{G_i,V}(v) = min(j|1 \leq j \leq i \wedge \forall k, j \leq k \leq i, v \in V_{G_k} \wedge l_{G_k,V}(v) = l_{G_i,V}(v))$ *and similarly for* $t_{G_i,E}(e)$.

As we want to capture patterns with rich information, we will also assume that the dynamic graph keeps track of the deleted vertices and edges, but only until they are added back into the graph. For example, consider the dynamic graph in Fig. 1 with five snapshots. Then $t_{G_1,V}(v_1) = 1$, $t_{G_4,V}(v_5) = 3$, etc. Also notice the edge between vertices $v3$ and $v1$ in snapshot G_3. It is deleted in snapshot G_4 but we keep information about this deleted edge in this snapshot. This information is discarded in snapshot G_5 because a new edge with exactly the same information is added there.

2.2 Predictive Rules

The aim of the mining algorithm is to find predictive graph rules, i.e. rules expressing how a subgraph of a snapshot will most likely change in future. As we want to incorporate the time information into the rules and at the same time we are interested in mining general patterns which are not tied to absolute time, we need to use relative timestamps for rules. A relative timestamp equal to 0 will denote a current change and timestamp equal to $-t$ will denote a change that happened t snapshots earlier. Now we can define predictive graph rules as follows.

Definition 3 (Predictive Graph Rule). *Let* G_A, G_C *be two static graphs with timestamp functions* $t_{G_A,V}$, $t_{G_A,E}$, $t_{G_C,V}$, $t_{G_C,E}$ *restricted to range* $(-\infty, 0]$ *such that union graph*[1] *of* G_A *and* G_C *is a connected graph and exactly one of the following conditions holds:*

i. $V_{G_A} = \emptyset \wedge V_{G_C} \neq \emptyset \wedge \forall v \in V_{G_C}(t_{G_C,V}(v) = 0) \wedge \forall e \in E_{G_C}(t_{G_C,E}(e) = 0)$

ii. $V_{G_A} \neq \emptyset \wedge V_{G_C} = \emptyset$

iii. $(V_{G_A} \cap V_{G_C} \neq \emptyset) \wedge$
$(\exists v \in V_{G_C}(t_{G_C,V}(v) = 0) \vee \exists e \in E_{G_C}(t_{G_C,E}(e) = 0)) \wedge$
$(\forall v \in V_{G_C} \setminus V_{G_A}(t_{G_C,V}(v) = 0)) \wedge$
$(\forall e \in E_{G_C} \setminus E_{G_A}(t_{G_C,E}(e) = 0)) \wedge$
$(\forall v \in V_{G_A} \cap V_{G_C}((t_{G_C,V}(v) = t_{G_A,V}(v) - 1 \wedge l_{G_C,V}(v) = l_{G_A,V}(v)) \vee (0 = t_{G_C,V}(v) \geq t_{G_A,V}(v) \wedge l_{G_C,V}(v) \neq l_{G_A,V}(v))) \wedge$
$(\forall e \in E_{G_A} \cap E_{G_C}(f_{G_C}(e) = f_{G_A}(e) \wedge ((t_{G_C,E}(e) = t_{G_A,E}(e) - 1 \wedge l_{G_C,E}(e) = l_{G_A,E}(e)) \vee (0 = t_{G_C,E}(e) \geq t_{G_A,E}(e) \wedge l_{G_C,E}(e) \neq l_{G_A,E}(e))))$

Then we say that $G_A \Rightarrow G_C$ *is a* predictive graph rule*, where* G_A *is called* antecedent *and* G_C consequent*.*

The first two conditions in the above definition cover situations in which the rules express either addition of an isolated graph into a dynamic graph or a deletion of a subgraph from a dynamic graph. The third condition covers situations in which one subgraph is transformed into another subgraph. Here, we require $V_{G_A} \cap V_{G_C} \neq \emptyset$ because we are not interested in rules consisting of unrelated graphs. In addition, we require the rule to contain at least one change related to a vertex or an edge. Vertices and edges occurring only in consequent must have timestamp equal to 0 as they represent an addition. For vertices and edges common for both graphs we require that they either were not changed a thus their relative timestamps differ by one, or they were changed and thus their timestamp cannot be lower in the consequent. Moreover, we cannot change edges by reorienting them, i.e. we would have to delete the original edge and add a new one with the opposite orientation. Lastly, as we keep track of the deleted edges and

[1]A graph created from union of vertices and edges.

vertices in the dynamic graph, the predictive rules can also contain these deleted vertices and edges. It does not pose any restriction to the patterns, contrariwise it can only help us capture more information in the patterns in case such information is present in the dynamic graph.

In Fig. 1 we can see two examples of graph prediction rules. Both rules depict changes in connection and also label changes. There is also a vertex with label A in both rules which is not changed and thus its timestamp is decreased by one in the consequent.

In order to select only interesting rules, various measures of significance are typically used. Here, we use support and confidence. As we use relative timestamps for graphs in rules, we also need to provide the notion of an occurrence of such a graph in the given dynamic graph.

Definition 4 (Occurrence of a rule graph)**.** *Let G be a graph used in a rule, say in antecedent without loss of generality, with timestamp functions $t_{G,V}$, $t_{G,E}$, and let $DG = (G_1, ..., G_i, ..., G_n)$ be a dynamic graph. We say that G* occurs *in snapshot G_i, written as $G \sqsubseteq G_i$, if there exists a function $\varphi : V_G \to V_{G_i}$ such that:*

i. $\forall u \in V_G(\varphi(u) \in V_{G_i} \wedge l_{G,V}(u) = l_{G_i,V}(\varphi(u)) \wedge t_{G,V}(u) = t_{G_i,V}(\varphi(u)) - i)$,

ii. $\forall e \in E_G(f_G(e) = (u,v) \Rightarrow \exists! e' \in E_{G_i}(f_{G_i}(e') = (\varphi(u), \varphi(v)) \wedge l_{G,E}(e) = l_{G_i,E}(e') \wedge t_{G,E}(e) = t_{G_i,E}(e') - i)$

Definition 5 (Support and Confidence)**.** *Let $G_A \Rightarrow G_C$ be a rule and $DG = (G_1, ..., G_i, ..., G_n)$ a dynamic graph. We define* support *of $G_A \Rightarrow G_C$,* support *of G_A, and* confidence *of $G_A \Rightarrow G_C$ as follows:*

$$\sigma_{DG}(G_A \Rightarrow G_C) = |\{i | G_A \sqsubseteq G_i, G_C \sqsubseteq G_{i+1}, 1 \le i \le n-1\}| / (n-1)$$

$$\sigma_{DG}(G_A) = |\{i | G_A \sqsubseteq G_i, 1 \le i \le n-1\}| / (n-1)$$

$$conf_{DG}(G_A \Rightarrow G_C) = \sigma_{DG}(G_A \Rightarrow G_C) / \sigma_{DG}(G_A)$$

For a set of dynamic graphs $DGS = \{DG_1, DG_2, ..., DG_m\}$ we extend these definitions as follows:

$$\sigma_{DGS}(G_A \Rightarrow G_C) = |\{i | \sigma_{DG_i}(G_A \Rightarrow G_C) > 0, 1 \le i \le m\}| / m$$

$$\sigma_{DGS}(G_A) = |\{i | \sigma_{DG_i}(G_A) > 0, 1 \le i \le m\}| / m$$

$$conf_{DGS}(G_A \Rightarrow G_C) = \sigma_{DGS}(G_A \Rightarrow G_C) / \sigma_{DGS}(G_A)$$

Thus, support of a rule for a single dynamic graph expresses the fraction of snapshots that were changed by the rule. For a set of dynamic graphs, we count the fraction of dynamic graphs that had at least one snapshot changed by the rule. Confidence expresses the frequency of such a change if we observe an occurrence of the antecedent. For example, both rules in Fig. 1 have support equal to 0.25 and confidence equal to 1.

Given a minimum support value σ_{min} and a minimum confidence value $conf_{min}$, the task is to find all predictive graph rules for which $\sigma \ge \sigma_{min}$ and $conf \ge conf_{min}$.

3 gSpan Revisited

The novel algorithm for mining predictive graph rules employs the framework of the gSpan algorithm [13]. We modified and further extended this framework for the purpose of mining graph rules from dynamic graphs. First, we revise the main ideas of gSpan and then we provide the details of the new algorithm.

gSpan [13] is an algorithm for mining frequent patterns (subgraphs) from a set of simple undirected static graphs. Simple means that it does not handle multiedges. The output frequent subgraphs are connected.

gSpan starts from single-edge patterns and extends these patterns edge by edge to create larger patterns. Each such pattern can be encoded by a *DFS (Depth-First Search) code*. A DFS code of a pattern represents a specific DFS traversal of the pattern and it is represented by a list of 5-tuples $(i, j, l_i, l_{(i,j)}, l_j)$. Such a 5-tuple represents an edge between the i-th and j-th discovered vertices by the DFS traversal, l_i and l_j are labels of those vertices, and $l_{(i,j)}$ is the label of the edge. Thus, the first 5-tuple has always $i = 0$ and $j = 1$, and it holds for other 5-tuples that $i < j$ if it is a forward edge in the DFS traversal and $i > j$ if it is a backward edge.

As there are more ways the DFS traversal can be performed on a single pattern, there are also more DFS codes for each pattern. A lexicographic order is defined on DFS codes and the minimum one is maintained for each pattern. This lexicographic order is also applied on codes of different patterns to represent the search space as a tree, called *DFS Code Tree*. In this DFS Code Tree, each vertex represents one DFS code and children of a vertex can be obtained by all possible single-edge extensions of the vertex. Therefore all codes on the same level of the tree have the same number of edges. Moreover, children of a vertex are ordered according to the lexicographic order. gSpan generates larger patterns in such a way that it corresponds to a depth-first search traversal of this DFS Code Tree, i.e. it generates patterns according to the lexicographic order. gSpan does not have to extend each pattern in all possible ways, it is enough to grow edges only from vertices on the rightmost path[2]. Specifically, it grows either a backward edge from the rightmost vertex[3] to another vertex on the rightmost path or a forward edge from a vertex on the rightmost path to a newly introduced vertex. When traversing the search space, gSpan checks whether the pattern of the considered DFS code is frequent. If not, it prunes the search space tree on this vertex and backtracks. This is possible because of the anti-monotonicity of the support measure. It also checks whether the considered code is the minimum one for the corresponding pattern. If it is not minimum, the search space tree is pruned on this vertex because all patterns in this pruned subtree were already found earlier.

[2]The *rightmost path* is given by the DFS code and it is the path from the root to the lastly discovered vertex by the DFS traversal.

[3]The last vertex on the rightmost path from root.

Algorithm 1 gSpan($\mathbb{D}$,$\mathbb{S}$)

```
1: sort the labels in $\mathbb{D}$ by their frequency;
2: remove infrequent vertices and edges;
3: relabel the remaining vertices and edges;
4: $\mathbb{S}^1 \leftarrow$ all frequent 1-edge graphs in $\mathbb{D}$;
5: Sort $\mathbb{S}^1$ in DFS lexicographic order;
6: $\mathbb{S} \leftarrow \mathbb{S}^1$;
7: for each edge $e \in \mathbb{S}^1$ do
8:     initialize $s$ with $e$, set $s.D$ by graphs containing $e$;
9:     Subgraph_Mining($\mathbb{D}$,$\mathbb{S}$,$s$);
10:    $\mathbb{D} \leftarrow \mathbb{D} - e$;
11:    if $|\mathbb{D}| < \sigma_{min}$ then
12:        break;
```

Algorithm 2 Subgraph_Mining($\mathbb{D}$,$\mathbb{S}$,s)

```
1: if $s \neq min(s)$ then
2:     return;
3: $\mathbb{S} \leftarrow \mathbb{S} \cup \{s\}$;
4: enumerate $s$ in each graph in $\mathbb{D}$ and count its children;
5: for each $c$, $c$ is $s$' child do
6:     if $\sigma(c) \geq \sigma_{min}$ then
7:         $s \leftarrow c$;
8:         Subgraph_Mining($\mathbb{D}_s$,$\mathbb{S}$,$s$);
```

The pseudocode of gSpan is given in Algorithm 1. By removing the infrequent vertices and edges, the input graphs can be significantly reduced and the overall efficiency increased. Frequent vertices are appended to results as the smallest frequent patterns. The main part of the algorithm starts from single-edge patterns. Specifically, Subgraph_Mining 2 procedure is recursively called on each such pattern. This procedure first tests whether the code s is minimum. If it is minimum, it enumerates its children by taking single-edge extensions. The procedure is then called on the frequent children.

4 DGRMiner Algorithm

In this section we describe the new algorithm called DGRMiner. It is based on the framework of gSpan, however, the framework is modified and extended. First, we provide necessary details about main modifications used in DGRMiner and then we present the pseudocode of the whole algorithm with description of remaining building blocks.

4.1 Static Representation of the Dynamic Graphs by Union Graphs

The first step is a transformation of an input dynamic graph[4] into a data structure that can be considered as a set of static graphs. The idea is that we are able to represent the graph rules by single graphs and the input dynamic graph as a set of static graphs in such a way that a modified static subgraph mining algorithm can be employed.

Let us start with the transformation of rules. In order to create a single graph from a rule, we take the union of the vertices and edges from its antecedent and consequent. Edges and vertices that do not represent any change in the rule will keep their labels and timestamps from the consequent. Let us remind that rules have relative timestamps less than or equal to 0 and thus these edges and vertices will have timestamps less than 0. Edges and vertices representing addition will keep their consequent timestamp, which is 0, but their labels will contain flag representing addition, for example label A will be changed to $+A$. However, timestamps of vertices and edges that were deleted or relabeled will have timestamps that are opposites of the antecedent timestamps. We know that consequent timestamps of such changes are equal to 0 so we can easily get the original value of the antecedent. We take the opposite values because later it will help us recognize current changes simply by timestamps greater than or equal to 0. Vertices and edges that were deleted or relabeled will also have new labels that can be easily decoded, for example $-A$ for deletion of an object with label A and $A => B$ for relabeling from A to B. Transformed rules from Fig. 1 are shown in Fig. 2.

Transformation of the input dynamic graph is very similar. Suppose that we have n snapshots in the dynamic graph. As the first snapshot does not represent any changes by itself, we create $n-1$ new graphs in the following way. When creating the k-th graph, consider union of vertices and edges from snapshots 1 to k as an antecedent, where vertices and edges have their last assigned labels and timestamps of last changes relative to k. We can assume that all vertices and edges from the first snapshot had timestamps equal to 1. Similarly, use snapshots 1 to $k+1$ to create a consequent. Then we use the method for rule transformation to create the k-th graph. All $n-1$ graphs can be computed in a single pass as we can update the i-th graph to get the $(i+1)$-th one.

Union of all graphs from the beginning may contain vertices and edges with very old changes that are not useful for the predictive rules. We use a window parameter to remove such vertices and edges from the union graphs. As edges cannot exist without their adjacent vertices, we do not remove old vertices adjacent to edges that are not old. Union graphs of the dynamic graph from Fig. 1 are shown in Fig. 2. In this case, window size is not set.

4.2 Modified DFS Code

Now that we have the dynamic graph represented by union graphs, which can be viewed as a set of static graphs, we made a large step towards mining the graph rules. There are, however, still several issues to be addressed.

Let us start with a richer representation of edges. In Section 3, we showed that gSpan uses 5-tuples of the

[4]Here, we assume that there is only one dynamic graph on the input. Extension to a set of dynamic graphs is described in Subsection 4.4.

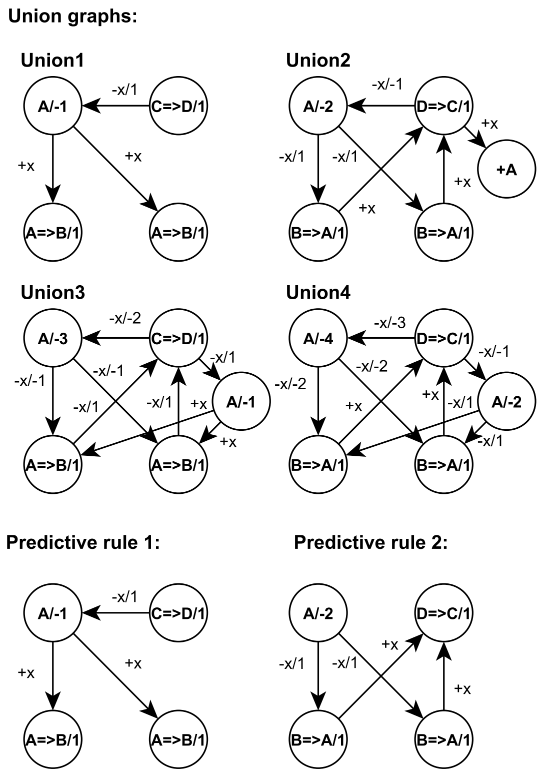

Figure 2: The union graph representation of the dynamic graph and the rules from Fig. 1.

form $(i,j,l_i,l_{(i,j)},l_j)$ to represent edges of patterns. In order to incorporate relative timestamps of rules and orientation of the edges, we simply extend these 5-tuples to 9-tuples of the form $(i,j,l_i,t_i,d_{(i,j)},l_{(i,j)},t_{(i,j)},l_j,t_j)$. It is the same as the original 5-tuple except for the new elements. Specifically, t_i, t_j, $t_{(i,j)}$ are used for the relative timestamps of vertex i, vertex j, and the edge between i and j. Element $d_{(i,j)}$ represents the orientation of the edge between i and j, and it is one of the following: $\leftarrow$, $\rightarrow$, $-$. The last one is used for undirected edges. Each pattern, i.e. graph rule in the condensed representation, can be represented as a list of such 9-tuples. Furthermore, it is easy to extend the gSpan's DFS Code for these 9-tuples and thus we can create ordering between patterns and find a minimum DFS Code for each pattern. For example, suppose that we obtained the following order of vertex labels: A, +x, C=>D, A=>B, -x. Then the minimum code of the Predictive Rule 1 from Fig. 2 is $(0,1,A,-1,\rightarrow,+x,0,A=>B,1),(0,2,A,-1,\rightarrow,+x,0,A=>B,1),(0,3,A,-1,\leftarrow,-x,1,C=>D,1)$.

4.3 Time Abstraction

In order to be able to deal with a broader class of dynamic graphs, we extended the mining algorithm to include two time abstraction methods. By time abstraction we mean usage of coarser timestamp values of union graphs in situations where exact values are not required or suitable.

The first method helps us to ignore timestamps of vertices. Specifically, we apply the signum function to all relative timestamps of vertices. Thus, negative timestamps become equal to -1 and positive timestamps become equal to 1. This method is useful for dynamic graphs in which all or almost all changes are caused by edges and vertices remain more or less intact.

The second method also uses the signum function but now it converts timestamps of both vertices and edges. It is useful in situations where patterns in dynamic graphs are very diverse and it is not possible to find many frequent patterns with exact timestamps.

4.4 The Complete DGRMiner

In this section we provide remaining details and a description of the whole DGRMiner algorithm for predictive graph rule mining. The pseudocode of DGRMiner is given in Algorithm 3.

First, DGRMiner converts the input dynamic graph into a set of union graphs as described in Section 4.1. In the case of a set of dynamic graphs, the algorithm simply computes union graphs for each one of them and then concatenates the results. It only needs to keep the mapping of those union graphs into the original dynamic graphs in order to be able to compute their support and confidence correctly. Optional application of an abstraction method, described in Section 4.3, follows next. Then the algorithm removes infrequent vertices and edges but only those that represent changes as the other ones may be used later for confidence computation. When computing frequencies, it takes labels, timestamps, and edge orientations into account. Before moving to single-edge patterns, DGRMiner outputs frequent single-vertex patterns with high enough confidence. To compute confidence, it needs to decode antecedents of the patterns and then compute their support. After that, the algorithm takes frequent initial edges and sort them according to the extended version of the DFS lexicographic order of gSpan. An *initial* edge is such an edge that represents a change or at least one of its vertices does.

Now for each initial edge we recursively call subprocedure DGR_Subgraph_Mining, which searches for patterns growing from a given initial edge. This subprocedure, described in Algorithm 4, uses several arguments. s denotes the current pattern, which is represented by its DFS Code. In $\mathbb{D}$ and $\mathbb{A}$ we keep union graphs in which current pattern and its antecedent can be found. Finally, when growing patterns from the i-th initial edge, we keep the first i initial edges in $\mathbb{E}_{start}$. This last argument is used in function *min*, which can be found in the first line of Algorithm 4. The purpose of this function is to check whether the DFS code of the given pattern is minimum, i.e. it was not found earlier when traversing the search space. Because all patterns grow only from the initial edges $\mathbb{S}^1$, it is enough to check whether we cannot represent the current pattern by a smaller DFS Code which starts by one of the edges in

Algorithm 3 DGRMiner($\mathbb{DG}$)

```
1:  convert the input dynamic graph(s) $\mathbb{DG}$ into the union graph representation $\mathbb{D}$;
2:  optional: apply a time abstraction method on union graphs;
3:  remove infrequent vertices and edges;
4:  output frequent change vertices with high enough confidence;
5:  $\mathbb{S}^1 \leftarrow$ all frequent initial edges in $\mathbb{D}$ sorted in DFS lexicographic order;
6:  for $i \leftarrow 1$ to $|\mathbb{S}^1|$ do
7:      $p \leftarrow i$-th edge from $\mathbb{S}^1$
8:      $p.D \leftarrow$ graphs which contain $p$;
9:      $p.A \leftarrow$ graphs which contain antecedent of $p$;
10:     $\mathbb{E}_{start} \leftarrow$ first $i$ edges from $\mathbb{S}^1$
11:     DGR_Subgraph_Mining($p$,$p.D$,$p.A$,$\mathbb{E}_{start}$);
```

Algorithm 4 DGR_Subgraph_Mining(s,$\mathbb{D}$,$\mathbb{A}$,$\mathbb{E}_{start}$)

```
1:  if $s \neq min(s, \mathbb{E}_{start})$ then
2:      return;
3:  enumerate $s$ in each graph in $\mathbb{D}$ and count its children;
4:  remove children of $s$ which are infrequent;
5:  enumerate antecedent of $s$ in graphs given by $\mathbb{A}$;
6:  set s.A by graphs which contain antecedent of $s$;
7:  $conf \leftarrow$ confidence of $s$;
8:  if $conf \geq conf_{min}$ then
9:      output $s$;
10: sort remaining children in DFS lexicographic order;
11: for each child $c$ do
12:     DGR_Subgraph_Mining($c$,$c.D$,$s.A$,$\mathbb{E}_{start}$);
```

$\mathbb{E}_{start}$. If we can find such a smaller code, then the pattern must have been discovered earlier a thus we backtrack.

If the code is minimum, we continue by enumerating the pattern in relevant graphs given by $\mathbb{D}$ and searching for its children candidates. This step is similar to the one in gSpan. Also, all infrequent children are removed. Before saving the current pattern, we need to compute its confidence. As we described in Section 4.1, we are able to extract the antecedent from the current pattern and then count its occurrences. Set $\mathbb{A}$ represents a set of candidate graphs, in which we should search for the antecedent occurrences. The actual set of graphs containing the antecedent is then saved to $s.A$, where $s.A \subseteq \mathbb{A}$, and it used as the $\mathbb{A}$ set for the pattern's children.

Before recursive processing of the children of the current pattern, we need to sort the children according to the extended version of the DFS lexicographic order. Set $c.D$ was created when the pattern s was enumerated and its children were counted.

5 Experiments

In this section we present results of experiments on three datasets. All the experiments were conducted by a C++ implementation of DGRMiner on a PC equipped with CPU Intel i5-4570, 3.2GHz, 16GB of main memory, and running 64-bit version of Windows 8.1. For all experiments, we set $conf_{min} = 0$ and window size for union graphs equal to 10.

Dataset	Dynamic graphs	Snapshots
ENRON	1	895
RESOLUTION	103	2911
SYNTH	1	101
SYNTH 20	20	2020

Table 1: Datasets used for experiments.

Rank	Vertex label
Employee	Emp
Vice President	VP
Director	Dir
President	Pres
Manager	Man
Trader	Trad
CEO	CEO
Managing Director Legal Department	MDLP
In House Lawyer	Law
Managing Director	MD

Table 2: Ranks of employees in the Enron dataset and the corresponding vertex labels.

5.1 Enron

The first dataset used in experiments is based on the email correspondence in the Enron company [3]. For our experiments we used preprocessed version of this email traffic [10]. Specifically, we used the data containing information about time, sender, receiver, and LDC topic. From the set of senders and receivers we created vertices of our dynamic graph. These vertices do not change through time. Each email message sent between a sender and a receiver is represented by addition of a directed edge in the dynamic graph. If the graph already contains the same edge, we just update its time of addition. Snapshots of the dynamic graph corresponds to days. As there were messages with anomalous dates, we removed all messages sent before 1998 and got 894 days of activity. With one extra day for vertex initialization we got 895 snapshots, as can be seen in Table 1. We used LDC topics as edge labels. There are 32 regular LDC topics expressing the topics of the messages plus two special topics used to label outlier messages and messages with non-matching topic. We also used rank of employees [10] to label vertices. Vertices with unknown rank were removed from the graph and thus only 130 vertices remained. Ranks and corresponding labels are available in Table 2.

Results on ENRON with $\sigma_{min} = 0.1$ can be found in the first row in Table 3. We decided to apply the time abstrac-

Dataset	Union vertices	Union edges	σ_{min}	$conf_{min}$	Time abstraction vertices	Time abstraction all	1-vertex rules	All rules	Running time (sec)
ENRON	46290	182720	0.10	0	✓	×	0	187	24.6
ENRON DEL	47612	196653	0.10	0	✓	×	0	233	29.3
RESOLUTION	15966	5275	0.05	0	×	×	26	36	0.3
RESOLUTION	15966	5275	0.05	0	×	✓	17	321	3.4
SYNTH	1124	2404	0.10	0	×	✓	6	82	0.3
SYNTH 20	31455	52112	0.10	0	×	×	121	1604	50.9

Table 3: Results of experiments. Number of union vertices and edges is taken over all union graphs of the given dataset. 1-vertex rules are rules whose union graph consists of only one vertex. Running time is averaged over five runs. For all experiments we set window of size 10 when building union graphs.

tion method for vertices because they are only added in the first snapshot and never changed. We found 187 frequent rules, none of which was a single-vertex rule.

We also modified the previous dataset by deleting edges which were not updated immediately the next day. This modified dataset is named ENRON DEL in Table 3. The change allows us to capture patterns which could not be captured only by edge additions. Examples of two rules from this dataset are shown in Fig. 3.

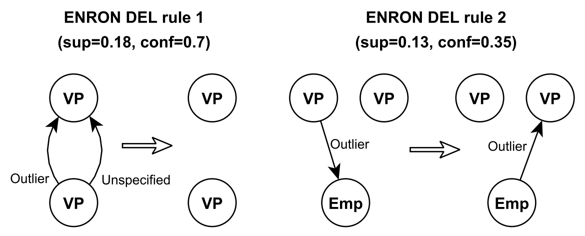

Figure 3: Examples of two rules from ENRON DEL.

5.2 Resolution Proofs in Propositional Logic

We used the set of graphs representing resolutions in propositional logic from [11] as the second dataset. Vertices of these graphs contain lists of literals in propositional logic. All edges are directed and have the same label in these graphs. These dynamic graphs are evolving by vertex and edge addition or deletion, and by change of vertex labels. Time of these events was transformed into a discrete sequence. Because there were 19 different assignments in total, the dynamic graphs had quite distinct vertex labels. In order to find frequent patterns, we restricted the dataset to only one assignment. Specifically, we took the assignment with the greatest number of solutions. This set of graphs contained 103 dynamic graphs with 2911 snapshots in total, see RESOLUTION dataset in Table 1. The initial snapshot of each dynamic graph in an empty graph. For more details about this data refer to [11].

We conducted two experiments on this dataset, both with $\sigma_{min} = 0.05$ as there were not many frequent patterns. One with no time abstraction, and one with time abstraction of both vertices and edges. From Table 3 we can see that the time abstraction helped us to find ten times more frequent rules for the same value of the minimum support. Furthermore, most of the rules were 1-vertex rules when the abstraction was not applied. Such rules capture vertex additions, deletions and relabelings without any context and may not be very informative.

5.3 Synthetic Datasets

Besides real-world data, we also tested our method on a synthetic datasets. One of this dataset, SYNTH in Table 1, was generated in the following way. First, a graph with 10 vertices and 20 randomly assigned edges was created. Then we iteratively built 100 snapshots, each snapshot from the previous one by randomly chosen changes. The number of changes ranged uniformly 0–1 for vertex deletion, 0–1 for edge deletion, 0–1 for vertex addition, 0–3 for edge addition, 0–2 for vertex label change, and 0–2 for edge label change. All newly selected vertex (edge) labels were chosen from a uniform distribution over set $\{A,B\}$ ($\{y,z\}$). Each new snapshot had to be different from the previous one. In order to keep approximately the same number of vertices (edges) through time, additions or deletions of vertices (edges) were suspended if the number of vertices (edges) was not in the $[k/2,2k]$ interval, where k = 10 (and k = 20 for edges). The second dataset, SYNTH 20, was created from 20 dynamic graphs, each one of them built by the process just described.

For SYNTH and $\sigma_{min} = 0.1$, the time abstraction of both vertices and edges was applied because there were almost no frequent patterns without the abstraction. On the other hand, experiments on SYNTH 20 with 20 dynamic graphs did not require any time abstraction and approximately 1600 frequent rules were found for the same value of the minimum support. This suggests that the support definition for a single dynamic graph is stricter than the one for a set of graphs.

6 Related Work

Several algorithms have been proposed for graph rule mining. However, a lot of these algorithms are expecting that the only changes in a dynamic graph are caused by edge additions, or by vertex additions if the vertices belong to the edges being added. These algorithms include GERM [1], LFR-Miner [7], and the algorithms presented in [5, 9]. These algorithms also pose various restrictions on the form of the rule graphs.

The work most related to ours is probably the algorithm for mining interesting patterns and rules proposed in [8]. This algorithm allows multiedges and also performs similar time abstraction, however, it also supposes that labels do not change and rules express only edge addition. Moreover, antecedents of the rules have to be connected. This is a stricter requirement than the one given by our algorithm, which requires only the union graphs of rules to be connected.

Predictive graph rules can be also seen as subgraph sequences of length two. Mining of subgraph subsequences from a set of graph sequences (dynamic graphs) is considered in [4], where algorithm FRISSMiner is proposed. This algorithm also works with union graphs, however, it builds the union graph from the whole sequence, i.e. the whole dynamic graph. FRISSMiner allows all types of changes in the input graph sequences but the subgraph sequences are not required to include the changes and thus user would still need to search for patterns representing changes in graphs. In addition, the algorithm is not designed for a single-dynamic-graph scenario.

Dynamic GREW [2] and the algorithm from [12] mine also patterns capturing information from several snapshots. They assume that the input dynamic graph has a fixed set of vertices, and edges are inserted and deleted over time, which makes them quite restricted.

Except for FRISSMiner, the algorithms presented in this section are designed only for the single-dynamic-graph scenario. On the contrary, FRISSMiner is designed only for the set-of-dynamic-graphs scenario.

Experimental comparison of the above algorithms with DGRMiner is difficult due to the fact that each algorithm mines different type of patterns. Also, different definitions of support and confidence affect which patterns are frequent. For example, GERM computes absolute support and counts multiple occurrences of a pattern in a snapshot. On the other hand, DGRMiner computes support of a pattern as a fraction of snapshots containing at least one occurrence of the pattern. Another difficulty arises from the fact that implementations of the algorithms, with the exception of GERM, are not freely available for download. Lastly, the datasets used for experimental evaluation of the above algorithms are often either not available or it is not possible to reproduce the same data.

7 Conclusion

We proposed a new algorithm DGRMiner for mining frequent predictive graph rules from both single dynamic graphs and sets of dynamic graphs. This algorithm is able to capture various changes in dynamic graphs. Edges in dynamic graphs are allowed to be directed or undirected multiedges. DGRMiner uses support and confidence as significance measures of the rules. Such graph rules are useful for prediction in dynamic graphs, they can be used as pattern features representing dynamic graphs or simply for gaining an insight into internal processes of the graphs. We evaluated the algorithm on two real-world and two synthetic datasets.

As future work, we plan to further investigate time abstraction methods and other ways of support computation. We also intend to modify the method for mining frequent closed patterns.

Acknowledgments. We would like to thank to Luboš Popelínský and other members of KDLab FI MU for their help and comments.

References

[1] Berlingerio, M., Bonchi, F., Bringmann, B., Gionis, A.: Mining graph evolution rules. In: ECML PKDD'09. Springer-Verlag, Berlin, Heidelberg, 2009, 115–130

[2] Borgwardt, K. M., Kriegel, H. -P., Wackersreuther, P.: Pattern mining in frequent dynamic subgraphs. In: IEEE ICDM'06. Washington, DC, USA, 2006, 818–822

[3] Cohen, W. W.: Enron email dataset. Web. Accessed 2 June 2015. urlhttps://www.cs.cmu.edu/~./enron/.

[4] Inokuchi, A., Washio, T.: Mining frequent graph sequence patterns induced by vertices. In: SIAM SDM, 2010.

[5] Kabutoya, Y., Nishida, K., Fujimura, K.: Dynamic network motifs: evolutionary patterns of substructures in complex networks. In: Proceedings of the APWeb'11. Springer-Verlag, Berlin, Heidelberg, 2011, 321–326

[6] Leskovec, J., Kleinberg, J., Faloutsos, C.: Graphs over time: densification laws, shrinking diameters and possible explanations. In: ACM KDD'05, New York, NY, USA, 2005.

[7] Leung, C. W. -K., Lim, E. -P., Lo, D., Weng, J.: Mining interesting link formation rules in social networks. In: ACM CIKM'10, New York, NY, USA, 2010, 209–218

[8] Miyoshi, Y., Ozaki, T., Ohkawa, T.: Mining interesting patterns and rules in a time-evolving graph. In: IMECS'11, Vol. I, March 16–18, 2011.

[9] Ozaki, T., Etoh, M.: Correlation and contrast link formation patterns in a time evolving graph. In: IEEE ICDMW'11, Washington, DC, USA, 2011, 1147–1154

[10] Priebe, C. E., Conroy, J. M. , Marchette, D. J., Park, Y.: Scan statistics on Enron graphs. Web. Accessed 8 June 2015. `<http://www.cis.jhu.edu/ parky/Enron>`

[11] Vaculík, K., Nezvalová, L., Popelínský, L.: Graph mining and outlier detection meet logic proof tutoring. In: G-EDM 2014, London, CEUR-WS.org, 2014, 43–50

[12] Wackersreuther, B., Wackersreuther, P., Oswald, A., Böhm, C., Borgwardt, K. M.: Frequent subgraph discovery in dynamic networks. In: MLG'10, ACM, New York, NY, USA, 2010, 155–162

[13] Yan, X., Han, J.: gSpan: Graph-based substructure pattern mining. In: IEEE ICDM'02, Washington, DC, USA, 2002.