Masaryk University
Faculty of Informatics

Pattern Mining in Dynamic Graphs

Doctoral Thesis

Karel Vaculík

Brno, Fall 2018

Masaryk University
Faculty of Informatics

Pattern Mining in Dynamic Graphs

Doctoral Thesis

Karel Vaculík

Brno, Fall 2018

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Karel Vaculík

Advisor: doc. RNDr. Lubomír Popelínský, Ph.D.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Lubomír Popelínský for his guidance, helpful comments, motivation and persistent patience. There are also many other people with whom I had an opportunity to discuss this thesis. Therefore, I would like to thank to Jan Ramon, Pavel Brázdil, Rui Camacho, Luis Torgo and other members of LIAAD Porto. I am also very thankful to my family and friends for their moral support, which helped me a lot to carry on in hard times.

Abstract

This thesis deals with mining in dynamic graphs where dynamic graphs are graphs changing in time.

Frequent pattern mining in binary trees is presented first. We show a method that allows us to represent binary trees by a set of subgraphs and then apply classification or anomaly detection methods on them. This method performs a generalization that helps us deal with vertices with diverse labels. We applied this method to resolution proof trees and it found unusual proofs that were not detected by an automatic corrector. Another presented method for binary trees is based on sequence mining and clustering and it allows us to group and analyse different strategies for solving the resolution proofs.

Most of the existing pattern mining algorithms so far has been restricted to specific classes of patterns expressing only specific types of changes in the graphs. We present DGRMiner, an algorithm which is able to mine more general frequent patterns than the other algorithms. The patterns are in the form of graph rules capturing various types of changes, i.e. addition and deletion of vertices and edges, and relabelling of vertices and edges. Furthermore, we show an extension of this algorithm that is able to mine anomalous patterns that deviate from the frequent ones. The frequent patterns then serve as an explanation for the anomaly patterns. Usefulness of DGRMiner is demonstrated by extraction of frequent and anomalous patterns from ENRON email network and resolution proof trees.

Next, we present WalDis, an algorithm for discriminative pattern mining in dynamic graphs. It differs from other approaches, which typically discriminate whole static graphs. The algorithm is able to discriminate between different events on a local level of dynamic graphs, i.e. events related to vertices or edges. The algorithm uses random walks and either a greedy method or a genetic algorithm to find patterns through inexact matching. Inexact matching helps us deal with intrinsic variability of timestamps and attributes in graphs. Discovered patterns allow us to understand the context of one group of events in contrast to another group. We verified this property on real-world datasets DBLP and ENRON, in which we discriminated machine learning conferences and email messages, respectively.

Keywords

data mining, graph mining, dynamic graphs, frequent patterns, anomaly detection, anomaly explanation, discriminative patterns, sequence mining, clustering

Contents

1 Introduction **1**
1.1 *Problem Statement* 1
1.2 *Contribution Summary* 3
1.3 *Organisation of the Thesis* 5

2 Graph Mining in Static and Dynamic Graphs **7**
2.1 *Pattern Mining in Static Graphs* 7
2.1.1 Basic Algorithms for Frequent Pattern Mining . 10
2.1.2 Other Approaches 12
2.2 *Pattern Mining in Dynamic Graphs* 13
2.2.1 Dynamic Subgraph Mining 15
2.2.2 Rule Mining in Dynamic Graphs 18
2.2.3 Sequence Graph Mining 24
2.2.4 Anomaly patterns 27
2.2.5 Discriminative patterns 30
2.3 *Dynamic Graph Datasets* 30
2.3.1 Resolution Proofs in Propositional Logic 31
2.3.2 ENRON 31
2.3.3 Synthetic Dynamic Graphs 32
2.3.4 DBLP 33
2.3.5 Phone Call Network 34

3 Frequent Pattern Mining **35**
3.1 *Domain-specific Pattern Mining* 35
3.1.1 Data 36
3.1.2 Generalized Subgraphs 36
3.1.3 Use of Generalized Subgraphs 43
3.1.4 Classification of Resolution Proofs 43
3.1.5 Detection of Outlying Resolution Proofs 44
3.1.6 Discussion 49
3.2 *Sequence Mining on Dynamic Graphs* 50
3.2.1 Resolution Proofs as Sequences 51
3.2.2 Sequence Clustering 52
3.2.3 Analysis of the Clusters 53
3.2.4 Discussion 55

4 DGRMiner for Mining Rule Patterns in Dynamic Graphs 57
4.1 *Predictive Graph Rules* 58
4.2 *gSpan Revisited* 62
4.3 *DGRMiner Preliminaries* 63
4.4 *DGRMiner Algorithm* 67
4.5 *Frequent Patterns Extracted from Dynamic Graph Datasets* . 70
4.5.1 ENRON 70
4.5.2 Resolution Proofs in Propositional Logic 72
4.5.3 Synthetic Datasets 73
4.6 *Discussion* 73

5 DGRMiner for Anomaly Detection and Explanation 75
5.1 *Anomaly Pattern Mining* 76
5.1.1 Single-vertex Anomalies 78
5.1.2 Enumeration of Anomaly Patterns in General . . 79
5.2 *Anomaly Detection Experiments* 81
5.2.1 ENRON 81
5.2.2 Resolution Proofs in Propositional Logic 82
5.3 *Discussion* 84

6 WalDis for Discriminative Pattern Mining 85
6.1 *Preliminaries* 87
6.2 *WalDis Algorithm* 90
6.2.1 Computing Statistics by Random Walks 91
6.2.2 Pattern Construction by a Greedy Approach . . 93
6.3 *Patterns Extracted by WalDis* 95
6.3.1 DBLP 98
6.3.2 Phone Call Network 98
6.3.3 ENRON 99
6.4 *WalDis Based on a Genetic Algorithm* 101
6.5 *Patterns Extracted by the Genetic Version of WalDis* 104
6.5.1 DBLP 106
6.5.2 ENRON 108
6.6 *Time Complexity* 110
6.7 *Discussion* 111

7 Conclusion and Future Work 113

A Author's Contribution **129**
A.1 *Publications* 129
A.2 *Invited Talks* 130
A.3 *Supervising Works* 131

List of Tables

2.1 Ranks of employees in the ENRON dataset and the corresponding vertex labels. *33*

3.1 Higher-level patterns with support ≥ 10. *41*

3.2 Classification results for frequent subgraphs. Precision and recall are stated for the class *incorrect*. *44*

3.3 Top outliers for data grouped by error E3. *46*

3.4 Internal evaluation of CE-2, CE-8, CET-2, and CET-8 clusterings by Dunn index (DI) and average silhouette width (SIL). *53*

3.5 Cluster representatives of CE-2, CE-8, CET-2, and CET-8 clusterings. *54*

4.1 Datasets used for experiments. *70*

4.2 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. *71*

5.1 Results of experiments. Running time is averaged over five runs, $conf_{min} = 0.6$ and $out_{min} = 0.8$. *82*

6.1 Experiment results of WalDis for different parameter settings. 10 independent runs were executed for each setting and score on positive and negative event sets were averaged over these 10 runs. *97*

6.2 Experiment results of EWalDis with accuracy computed for both training and test datasets. *105*

List of Figures

2.1 Illustration of isomorphism on graphs. Letters in vertices and next to the edges represent vertex and edge labels, respectively. *9*
2.2 An example of a set of graphs and a pattern. Letters in vertices represent labels. There are no edge labels for the sake of simplicity. The occurrences of the pattern are depicted by dashed lines. *10*
2.3 An example of a dynamic graph with four snapshots. Vertices represent users of a social network and edges friendship relationships among them. *15*
2.4 An example of a correct (on the left) and an incorrect (on the right) resolution proof. *32*

3.1 Distribution of clause labels ordered by frequency. *37*
3.2 An example of pattern unification. *38*
3.3 Drawings of the outlying instances from Table 3.3. *48*

4.1 An example of a dynamic graph and two predictive graph rules. Numbers after slash symbols represent timestamps and dotted edges represent deleted edges. *58*
4.2 The union graph representation of the dynamic graph and the rules from Fig. 4.1. *66*
4.3 Examples of two rules from ENRON DEL. Vertex labels VP and Emp stand for Vice President and Employee, respectively. *72*

5.1 An illustration of a dynamic graph with six snapshots, a frequent pattern with occurrences P1, P2, P3, and an anomaly pattern with occurrences A1, A2. *76*
5.2 An example of anomaly pattern enumeration. *80*
5.3 An example of anomaly pattern enumeration with regard to a frequent pattern with an empty antecedent. *80*
5.4 Examples of anomaly patterns from experiments. *83*

6.1 An example of a dynamic graph and a discriminative pattern with respect to positive and negative events. Parameter l denotes labels or attributes, and parameter t timestamps. *86*

6.2 A pattern found by WalDis in DBLP network dataset. *99*
6.3 A pattern found by WalDis in TELCO network dataset. *100*
6.4 A pattern found by WalDis in ENRON network dataset. *100*
6.5 An illustrative example of notions used in the genetic algorithm. Highlighted edges e11 and e21 are assumed to be already selected. *103*
6.6 A pattern found by EWalDis in DBLP network dataset with *kdd* publication as a positive event and *icml* as a negative event. *106*
6.7 A pattern found by EWalDis in DBLP network dataset with *nips* publication as a positive event and *kdd* as a negative event. *107*
6.8 A pattern found by EWalDis in DBLP network dataset with *nips* publication as a positive event and *icml* as a negative event. *108*
6.9 A pattern found by EWalDis in ENRON network dataset with *California bankruptcy* email topic as a positive event and *California business* as a negative event. *109*
6.10 Matching score distributions of discriminative patterns found by EWalDis on ENRON dataset. *109*

1 Introduction

Importance of data mining and machine learning techniques increases in a large number of domains due to availability of data. For instance, data mining is used in areas such as biology [21], linguistics [3], or World Wide Web [61]. The majority of standard data mining algorithms assumes data instances to be independent. Attribute-value representation, e.g. a table in a relational database or a spreadsheet, is very common for such data. However, there is a lot of real-world scenarios where relationships between data instances exist and *relational learning* [43] can be used. For example, there are relationships between people in social networks, between chemical elements in chemical compounds, etc. Algorithms considering also the relationships may achieve significantly higher performance than algorithms that take only the information about individual objects into account.

In many situations, graphs can be used for data representation. There is already a lot of algorithms designed for graphs. Moreover, graphs do not have to represent only static or persistent relationships but also dynamic ones, such as communication between people or electronic devices. Besides communication, dynamic graphs can also represent a general evolution of networks in time, e.g. addition and removal of individuals, change of linking between individuals, change of values associated with individuals, etc. In the context of data mining, terms *graph* and *network* are often interchanged. Nevertheless, term *graph* is more common in the sense of mathematical or computer representation, and term *network* is more common for real instances, such as *computer networks*.

1.1 Problem Statement

In this thesis, we are interested in the area of **graph mining** [29] on **dynamic graphs**, which is a data mining field considering dynamic graphs as the input. Graph mining can be performed at a global or a local level. **Global-level mining** focuses on properties of whole graphs and their evolution in time. An example of an analysis of global properties is described in [59], where shrinking diameter and network densification were observed in real-world networks through time.

On the other hand, **local-level mining** is concerned with graph evolution on the node, the edge, or the subgraph level. Similarly as in the previous case, we can track some properties of these elements, such as node degree or node betweenness. In order to capture more complex dependencies, we can also mine graph patterns. In static graphs, these patterns typically correspond to subgraphs, which are mostly connected. However in dynamic graphs, the locality can be expressed also from the perspective of time. Thus, patterns in dynamic graphs can be represented by subgraphs and their evolution in a short time interval. This thesis focuses on mining methods of various types of local-level patterns in dynamic graphs.

Frequent Patterns. The most basic patterns are the *frequent patterns* and they are the primary focus of this thesis. From the perspective of graph mining on static graphs, the frequent patterns are typically subgraphs that occur in a large portion of the input set of graphs or frequently in a single graph. For example, given a set of chemical compounds, a Benzene ring may be considered a frequent pattern [27]. Frequent patterns in dynamic graphs, such as email communication networks, can represent patterns of communication. For example, if a manager receives several emails concerning some problems, he or she may frequently send some more emails to deal with the problems. Frequent patterns are useful for several reasons. First, they can help with the interpretation of the graphs in the process of exploratory analysis. Second, frequent patterns can be exploited for other tasks of data mining and machine learning. They are appropriate as features representing the graphs and thus can be used for graph classification, clustering, or indexing [29].

Anomaly Patterns. Frequent patterns represent common behaviour in graphs. Contrariwise, *anomaly patterns* occur infrequently and they characterise deviations from the common behaviour. Although infrequent, these patterns may be of a great significance. For example, they can be used for fraud detection, network intrusion detection or identification of suspicious behaviour.

Discriminative Patterns. Previous two types of patterns are mainly mined in an *unsupervised* manner. This means that there are no classes or other output variables that are crucial in the mining process. Contrarily, mining of *discriminative patterns* belongs to the group of *supervised* methods because the patterns are extracted in such a way that

different sets of graphs can be distinguished by them. For example, considering the chemical compounds once again, there may be substructures that typically occur in the compounds having a toxic effect and do not occur in those that do not have such an effect. Discriminative patterns can be used for distinguishing different types of vertices, edges, or subgraphs as well.

1.2 Contribution Summary

This section summarises the contribution of the thesis. Several methods for pattern mining in dynamic graphs have been created as a part of author's research. Here we give a brief overview of the most significant publications. Full list can be found in Appendix A.

- K. Vaculík, L. Nezvalová, and L. Popelínský. Graph mining and outlier detection meet logic proof tutoring. In *Proceedings of EDM 2014 Ws Graph-based Educational Data Mining (G-EDM)*, CEUR-WS.org, pp. 43–50, ISSN 1613-0073, 2014.

  K. Vaculík, L. Nezvalová, and L. Popelínský. Educational data mining for analysis of students' solutions. In *Artificial Intelligence: Methodology, Systems, and Applications* (AIMSA), London, Springer, pp. 150–161, 2014.

  These two publications deal with analysis and patter mining in resolution proof graphs created by students. The first one presents a method for subgraph pattern mining that uses domain knowledge about resolution proofs. Extracted patterns are used for classification of proofs and class-based anomaly detection. The second one describes a method that employs a sequence mining approach, whose results are then used for clustering of resolution proofs. The methods and their results are presented in Chapter 3.

- K. Vaculík. A Versatile Algorithm for Predictive Graph Rule Mining. In *Proceedings ITAT 2015: Information Technologies - Applications and Theory*, Prague, CEUR-WS.org, pp. 51–58, 2015.

This publication proposes a new algorithm DGRMiner for frequent pattern mining in dynamic graphs. These patterns are in the form of predictive rules and, in comparison to other existing approaches, they are able to capture various changes in graphs, such as addition and deletion of vertices and edges, and relabelling of vertices and edges. The algorithm as well as patterns found in real-world and synthetic dynamic graphs can be found in Chapter 4.

- K. Vaculík, and L. Popelínský. DGRMiner: Anomaly Detection and Explanation in Dynamic Graphs. In *Advances in Intelligent Data Analysis XV - 15th International Symposium* (IDA), Stockholm, Sweden, pp. 308–319, 2016.

  The previous publication presents DGRMiner for frequent patterns. In this publication, we present an extension of DGRMiner for anomaly detection and explanation. These anomaly patterns are deviating from the frequent ones and, in addition, the frequent ones serve as an explanation of the deviations. The algorithm is described and again checked on a set of dynamic graphs in Chapter 5.

- K. Vaculík, and L. Popelínský. WalDis: Mining Discriminative Patterns within Dynamic Graphs. In *Proceedings of the 22nd International Database Engineering & Applications Symposium* (IDEAS), ACM, New York, NY, USA, pp. 95–102, 2018.

  The last publication describes WalDis, a novel method for discriminative pattern mining in dynamic graphs. The patterns discriminate events on the level of vertices and edges. For example, events may represent changes of vertices' attributes. Given two different sets of events, positive and negative, WalDis mines subgraph patterns that appear in the neighbourhood of positive events but not negative ones. Discriminative patterns allow us to understand the context of positive events and how these events differ from the negative events. WalDis uses random walks as a sampling method and a greedy approach to extract patterns. More information about WalDis can be found in Chap-

ter 6. This chapter also presents EWalDis, an extension of WalDis that uses a genetic algorithm instead of the greedy approach.

1.3 Organisation of the Thesis

The thesis is organized as follows. Chapter 2 provides basic definitions regarding the dynamic graphs and an overview of the state-of-the-art in the area of patter mining in dynamic graphs. In the rest of the thesis, we present several methods for pattern mining in dynamic graphs. Specifically, we present methods for mining resolution proofs in logic in Chapter 3. Chapter 4 describes DGRMiner, an algorithm for frequent pattern mining in dynamic graphs. An extension of DGRMiner for anomaly detection and explanation is introduced in Chapter 5. Next, Chapter 6 presents WalDis, an algorithm for discriminative pattern mining in dynamic graphs. Finally, conclusion and directions for future work can be found in Chapter 7.

2 Graph Mining in Static and Dynamic Graphs

This chapter contains basic definitions and the state of the art of pattern mining in dynamic graphs. It also covers description of all graph datasets mentioned in the subsequent chapters. The first section provides definitions for static graphs and an overview of techniques for static graphs, which serve as a base case and preliminaries for dynamic graphs. The second section covers the main part – dynamic graphs. It contains necessary definitions regarding dynamic graphs and an overview of research on the topic of pattern mining in dynamic graphs. Specifically, it provides an overview of frequent, anomaly, and discriminative pattern mining. The third section closes this chapter with a description of the datasets.

2.1 Pattern Mining in Static Graphs

Before describing particular techniques, we give formal notations and definitions regarding the *static* graphs. A large number of mining algorithms work with *labelled* graphs so we define a static graph as a labelled graph. Let us denote a set of all 2-element subsets of A by $[A]^2$.

Definition 1 (Static labelled graph)**.** *A* static labelled undirected 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 (also called nodes),* E_G *is a set of edges,* $f_G : E_G \to [V_G]^2$ *is a map assigning a set of two vertices* $\{u, v\}$*,* $u \neq v$*, to every edge,* $l_{G,V} : V \to L_V$ *and* $l_{G,E} : E \to L_E$ *are two maps describing labelling of the vertices and edges, respectively. In the same way, we define a* static labelled directed graph*, with the exception of function* f_G*, which now assigns a pair of vertices* (u, v)*,* $u \neq v$*, to every edge, i.e.* $f_G : E_G \to V_G \times V_G$*.*

This definition is general and covers *multigraphs* as well. If a graph has no multi-edges, i.e. $f(e_1) \neq f(e_2)$ for all $e_1 \neq e_2$, we say that the graph is *simple*. If all vertices of a graph are pairwise adjacent, then the graph is *complete*.

Plain (*unlabelled*) graphs are graphs without labels, i.e. $L_V = L_E = \{\emptyset\}$. If we use numeric values instead of nominal labels, we get *weighted* graphs. Graphs with nominal or numeric values on vertices

and/or edges are generally denoted as *attributed* graphs. Moreover, this broader class of graphs allows also sets of key-value pairs on vertices and edges, or even more complex structures. From now on, we will assume all graphs to be *simple* and *labelled*, unless stated otherwise. Furthermore, for the sake of simplicity, if it is clear from the context or a particular claim holds for both directed and undirected graphs, we will use *static graph* expression without further specification.

Let $G = (V_G, E_G, f_G, l_{G,V}, l_{G,E})$ and $G' = (V_{G'}, E_{G'}, f_{G'}, l_{G',V}, l_{G',E})$ be two static undirected graphs. G and G' are said to be *isomorphic*, written as $G \simeq G'$, if there exist bijections $\varphi : V_G \to V_{G'}$ and $\psi : E_G \to E_{G'}$ satisfying $f_{G'}(\psi(\{x,y\})) = \{\varphi(x), \varphi(y)\}$ for all $x, y \in V_G$. Next, we say that G is a *subgraph* of G' (or G' is a *supergraph* of G), written as $G \subseteq G'$, if $V_G \subseteq V_{G'}$, $E_G \subseteq E_{G'}$, and $f_G(e) \subseteq V_G$ for all $e \in E_G$. That means that all ends of E_G must be in V_G for G to be a graph. Furthermore, if $G \subseteq G'$ and G contains all the edges $\{x,y\} \in E'$ with $x, y \in V$, then G is an *induced subgraph* of G'. A subgraph that is complete is called a *clique*. Let G and H be two static undirected graphs. If there is a subgraph G_0 of G such that H and G_0 are isomorphic, we say that H is *subgraph isomorphic* to G and denote this relationship by $H \sqsubseteq G$. If we want to point out that a specific functions φ and ψ are used, we write $H \sqsubseteq_{\varphi,\psi} G$. We also say that G_0 is an *embedding* (or *occurrence*) of H in G. Clearly, $G \simeq G'$ implies $G \sqsubseteq G'$ and $G' \sqsubseteq G$. Determining whether G contains such an embedding is known as the *subgraph isomorphism* problem, which is NP-complete [42].

The above definition of isomorphism and subgraph isomorphism does not take into account the labels of the vertices and edges. However, graph mining algorithms for pattern mining typically check the labels too when checking isomorphism. We define *label-preserving isomorphism* $\simeq^*$ and *label-preserving subgraph isomorphism* $\sqsubseteq^*$ by extending the previous definitions by adding the following two conditions: $l_{G,V}(u) = l_{G',V}(\varphi(u))$ for all $u \in V_G$ and $l_{G,E}(e) = l_{G',E}(\psi(e))$ for all $e \in E_G$. The above definitions can be easily modified for directed graphs.

Examples of isomorphisms are show in Fig. 2.1. For $G1$ and $G2$ relation $G1 \simeq G2$ holds. However, $G1 \not\simeq^* G2$ because of the labels on edges. Next, $G3 \sqsubseteq^* G1$ and $G3 \sqsubseteq G2$, but $G3 \not\sqsubseteq^* G2$.

Given a graph G, the *support* (or *frequency*) of a graph H in G is the number of edge-disjoint occurrences of H in G [29]. There are

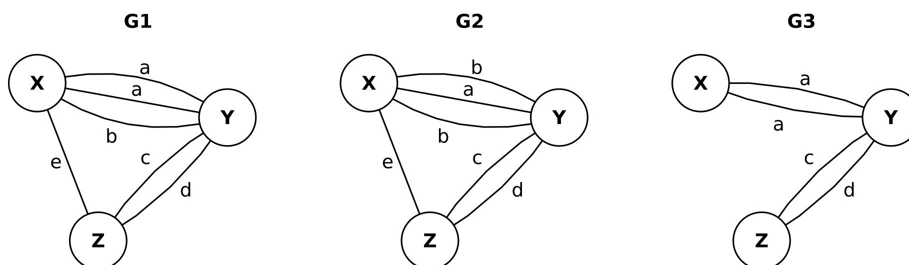


Figure 2.1: Illustration of isomorphism on graphs. Letters in vertices and next to the edges represent vertex and edge labels, respectively.

also definitions of support using vertex-disjoint or other types of constraints in the literature, e.g. [19, 39]. The definitions can vary for different types of graphs. However, it is important for a support definition to satisfy the *anti-monotonic property*, which states that the support of a graph cannot be larger than the support of any of its subgraphs. This property allows algorithms to search for the frequent patterns effectively – see below. For a set of graphs $\mathcal{G} = \{G_1, G_2, ..., G_k\}$, the support of a graph H is typically defined as the number of graphs of $\mathcal{G}$ having an embedding of H [29]. In this case, it is possible to express the support relatively as percentage. Given a *minimum support* value, graph H is frequent if its support is greater than or equal to the *minimum support* threshold.

An example of a set of three graphs with a pattern found in these graphs is given in Fig. 2.2. The occurrences of the pattern are depicted by dashed lines. The pattern occurs in two graphs and thus its support is 2/3. Now if we consider a single-graph scenario with graph $G3$, we can see that there are two occurrences of the pattern. However, these occurrences are not disjoint and we have to count only one into support. Otherwise anti-monotonic property would not have to hold for some patterns.

Task of *frequent subgraph mining* is to find all frequent subgraphs in a given graph or a set of graphs for a given *minimum support* value. In the context of graph mining, subgraphs are also frequently called *patterns*. The anti-monotonic property allows an algorithm to search the space of all patterns in an effective manner. Such an algorithm can

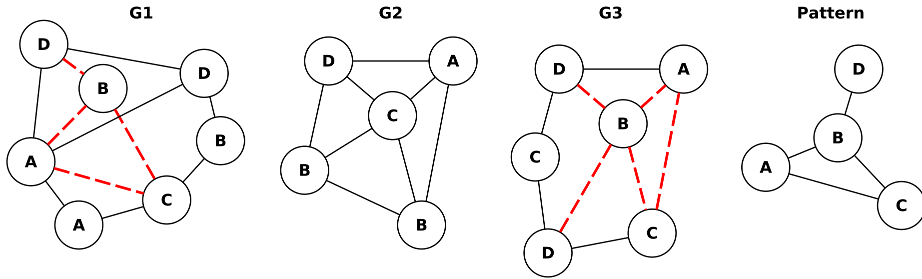


Figure 2.2: An example of a set of graphs and a pattern. Letters in vertices represent labels. There are no edge labels for the sake of simplicity. The occurrences of the pattern are depicted by dashed lines.

start with smaller patterns and it does not have to check superpatterns (supergraphs) of a pattern whose support is too low. Nevertheless, finding all frequently occurring patterns is generally nontrivial because in order to check an existence of a pattern, it is necessary to solve the subgraph isomorphism problem as we stated earlier and it has been shown that the problem be NP-complete [42].

Moreover, graphs can contain a large number of frequent patterns in practice. One way of avoiding this is to use a higher value of *minimum support*. However this approach is not always appropriate and it is better to restrict the frequent patterns to the closed or the maximal ones. A frequent pattern is *closed* if and only if there does not exist a superpattern of this pattern that has the same support. A frequent pattern is *maximal* if and only if it does not have a frequent superpattern.

In the following paragraphs, we discuss several existing algorithms and approaches for pattern mining in static graphs.

2.1.1 Basic Algorithms for Frequent Pattern Mining

gSpan. Given a set of undirected graphs and a value of *minimum support*, the gSpan algorithm [108] is able to find all frequent subgraphs in the given set. gSpan maps subgraphs to a unique minimum depth-first search (DFS) code and uses a lexicographic order on these codes to order subgraphs. Based on this lexicographic or-

der, gSpan employs a DFS strategy to mine frequent subgraphs efficiently. Specifically, gSpan traverses a DFS Code Tree, where the code of a node corresponds to the parent's code extended by one edge and the siblings are ordered according to the lexicographic order. Using this approach, the traversal starts from the smallest subgraphs and it backtracks if the corresponding subgraph is not frequent. Nevertheless, NP-completeness of the subgraph isomorphism problem makes the runtime of gSpan, and also similar algorithms, exponential. Fortunately, in practice, graphs with diverse labels can decrease the runtime substantially. Another problem of gSpan, and algorithms for frequent subgraph mining in general, is that for large or dense graphs, the number of frequent subgraphs is very large and it is not practical to mine all of them [29].

Subdue. The previous algorithm evaluates subgraphs according to their support. Subdue algorithm [28], on the other hand, searches for subgraphs that can best compress the input graph or the set of input graphs[1]. The compressibility is evaluated by the Minimum Description Length (MDL) principle [91]. The best substructure is the one that minimizes $DL(S) + DL(G|S)$, where $DL(S)$ is the description length of the substructure and $DL(G|S)$ is the description length of the input graph G after compression of S. After the best substructure is found, the input graph is compressed by replacing the occurrences of this substructure with pointers and the whole process repeats. This method yields a hierarchical description of the input graph in terms of discovered substructures.

Sleuth. Another algorithm based on depth-first search strategy is Sleuth [120], which is designed for mining frequent subtrees from a set of rooted trees. An extended version of Sleuth, described in [29], is able to mine both *ordered* and *unordered* and both *induced* and *embedded* trees. Ordered trees differ from the unordered ones in that the order of sibling nodes matters in the ordered trees. Sleuth uses term *induced trees* for trees whose embeddings in the input trees preserve the parent-child relationship, i.e. if two nodes are in a parent-child relationship in the induced tree then they are in this relationship also in the input trees. On the other hand, *embedded trees* require only

1. A set of graphs can be seen as one graph having several components.

the ancestor-descendant relationship to be preserved, i.e. if two nodes are in a parent-child relationship in the embedded tree then the path connecting these two nodes in the input graphs can contain other nodes.

2.1.2 Other Approaches

Network motifs. There is also another type of patterns similar to frequent subgraphs, known as *network motifs* [68]. Network motifs are n-node subgraphs in a directed network, where n typically ranges over small values (e.g., $3 \leq n \leq 7$). For each value n from a given range of values, all non-isomorphic weakly connected subgraphs with n nodes are found in the given network. Number of occurrences of each subgraph is compared to the numbers of occurrences of the same subgraph in randomly generated networks. If the number of occurrences in the original graph is significantly higher, the subgraph is regarded as a network motif.

Graph Queries. The problem of pattern mining in graphs can be also formulated from a different perspective. A user can specify a query in the form of a graph and the task is to find the occurrences of such a query graph in an input graph or in a set of graphs. This topic is close to graph databases and there is a lot of research done, for example [14, 110, 113, 114] just to name a few research papers. Although the task of graph querying may seem straightforward, the algorithms can be quite diverse. For example, the algorithm presented in [114] searches for top-k *diversified* subgraphs, i.e. k occurrences with the minimum number of overlapping nodes.

Discriminative patterns. Besides algorithms for frequent patterns, there are also ones that search for other types of patterns. There is a number of methods for discriminative pattern mining in static graphs [41, 54, 98, 109]. These methods assume two sets of either directed or undirected graphs and they search for subgraphs that are frequent in one set and not in the other set.

Anomaly patterns. Anomaly detection in static graphs can be tackled from different perspectives. The most straightforward methods compute various features from graphs, such as node centralities, or

compute distances between considered entities, e.g. nodes or edges, and then employ rudimental anomaly detection algorithms to find outlying nodes, edges, or graphs [7]. The problem of finding unusual or infrequent subgraphs, i.e. patterns, in static graphs was addressed, for example, by Subdue algorithm [79] mentioned earlier. Subdue performs anomaly detection on labelled graphs with regard to a measure that is inversely related to Minimum Description Length principle. Graphs with numerical attributes can be processed by a discretization technique [31] so that it is easier to search for anomaly patterns. A different approach is presented in [37], where patterns that differ only a little from regular patterns are mined. This approach is different from the one focusing on infrequent patterns and it can be useful when searching for fraudulent behaviour, for example, because frauds are typically done in such a way so that they do not arouse suspicion. More information about anomaly detection in graphs can be found in survey [7].

2.2 Pattern Mining in Dynamic Graphs

This section on pattern mining in dynamic graphs is further divided into several parts. First, we provide a notion of a *dynamic graph* (also known as *time-evolving graph* [13] or *temporal graph* [90]). Then we give an overview of mining techniques for various types of patterns. Specifically, we go through subgraph, rule, sequence, anomaly and discriminative patterns.

Before defining a dynamic graph, we extend each static graph G by *timestamp* functions $t_{G,V} : V_G \rightarrow T$ and $t_{G,E} : E_G \rightarrow T$, which map each vertex and edge to a point in time, respectively. We will work with discretized time and thus T will be the set of integers, i.e. $T = \mathbb{Z}$. A dynamic graph is then given by a finite sequence of such extended static graphs.

Definition 2 (Dynamic labelled 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 $1 \leq i \leq n$. Graph G_i is referred to as the* snapshot *of DG at time i.*

Even though the definition is somewhat specific, it still gives us a number of possibilities how to represent network relationships evolv-

ing in time and we will show in subsequent chapters that it is useful enough for many graph mining problems. For example, graph snapshots can represent states of a network obtained at fixed time periods, or states after individual changes in a network. The vertices and edges can appear and disappear in such a network and also their labels can change over time.

An example of a dynamic graph representing a simple social network is shown in Fig. 2.3. There are four users in the network: Alice, Bob, Carol, and Dave. Edges represent friendship relationships among these users in four consecutive months, depicted by four graph snapshots. In this example, Carol and Dave connected with Alice and Bob within the second and the third month. However, Bob left the social network in the fourth month.

By using the timestamp functions, we can add more information about the processes in the graphs and it is also possible to compare the processes by looking at the relative differences between timestamps. For example, we can discover that some changes in graph typically happen at the same or similar time. Vertex and edge timestamps can represent the creation of these vertices and edges or the change time of their labels. If the timestamps are not necessary for a specific scenario, it is possible to omit them or use a constant value. It is also possible to create a simple temporal graph from a single static graph extended by the timestamp functions. Timestamps in such a single graph allow us to distinguish timestamps at which edges and vertices appeared in the graph. This simplified version of a dynamic graph still captures a great deal of temporal information and as we shall see shortly, some algorithms use similar representation for dynamic graphs.

Nevertheless, there is a lot of other definitions of dynamic graphs in the literature. Moreover, there is also a lot of different definitions of patterns in dynamic graphs. We present various types of patterns and dynamic graphs in the following subsections. The simplest patterns, which are similar to those in static graphs, are covered in the first subsection and we call them *dynamic subgraphs*. The following subsections describe patterns that are composed of two or more graphs and we denote them as *rule* and *sequence graph patterns*. Separate subsections are also dedicated to discriminative and anomaly patterns.

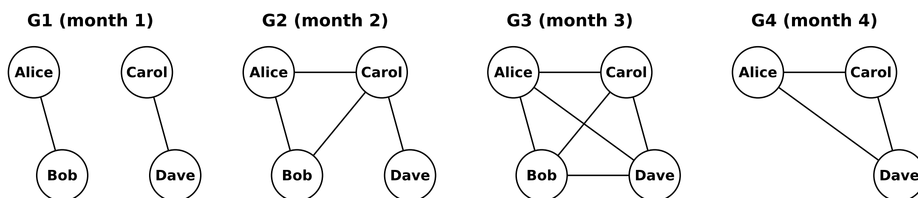


Figure 2.3: An example of a dynamic graph with four snapshots. Vertices represent users of a social network and edges friendship relationships among them.

2.2.1 Dynamic Subgraph Mining

We start the overview with methods that focus on subgraph mining from dynamic graphs and are not very different from those used for static graphs. Briefly put, the mining process typically involves solving a subgraph isomorphism problem similar to the one for static graphs. In this case, however, it is also necessary to take into consideration various time constrains. More complex patterns are discussed in later sections.

Dynamic GREW. Mining of frequent dynamic subgraphs is considered in [16], where Dynamic GREW algorithm is proposed. It is assumed that the input dynamic graph has a fixed set of nodes, and edges are inserted and deleted over time. Presences and absences of edges are expressed by sequences of 1s and 0s called *existence strings*. A dynamic subgraph of length k of a dynamic graph is then a subgraph from the topological view and also the dynamic view, i.e. the existence strings of the dynamic subgraph have length k and they are substrings of existence strings of the original graph starting from the same position. Frequent dynamic subgraph is a dynamic subgraph with at least t occurrences for a given value of t. Dynamic GREW is based on pattern mining on static graphs and it allows the user to integrate other algorithms for this task too.

Such a dynamic graph can also be viewed as a union graph, in which an existence string is assigned to each edge. This approach is further elaborated in [106], where a novel framework for frequent subgraph discovery is proposed. Here, the existence strings are over an alphabet

$L \cup \varepsilon$, where L is a set of edge labels and ε denotes an edge absence. In this way, the existence strings are able to capture also the edge labels and their change. This framework also allows to use node labels and, as in the case of Dynamic GREW, the user can choose to integrate a different algorithm for subgraph mining on static graphs.

Dynamic subgraph enumeration. An algorithm for solving the *Dynamic Subgraph Enumeration* (DSE) problem is presented in [1]. This problem is stated as follows. Given a subgraph H and a sequence $\mathcal{G}$ of graphs $G_1, G_2, \ldots$, in which G_{t+1} is obtained by modifying a single edge in G_t, the goal is to maintain a dynamic data structure for each G_t so that the number of subgraphs of G_{t+1} isomorphic to H can be estimated efficiently, significantly faster than recomputing them in G_{t+1} from scratch.

Time-respecting subgraphs. Subgraph mining from *interaction temporal graphs* is considered in [90]. An interaction temporal graph is a static directed graph, where edges represent interactions between nodes and each edge has assigned two values standing for the start time and the duration of the interaction. Presented algorithm mines connected subgraphs, called *time-respecting subgraphs*, in which the interaction of an edge does not start long after the interactions of the adjacent edges.

Motifs. Using the terminology from the previous paragraph, mining of maximal motifs that are simultaneously time-respecting subgraphs is considered in [58]. The algorithm is focused on interaction networks too, but now there can be more interactions between a pair of nodes with the restriction that at any given time at most one event can be assigned to a node. A different type of temporal motifs, called *trend motifs*, is considered in [53]. A trend motif describes a recurring connected subgraph where each of its vertices or edges exhibits similar dynamics over a user defined period. Vertices have weights and a trend on a vertex over a time interval is given either by increasing or decreasing weight. Each vertex in a trend motif has a constant trend, labelled either by symbol $+$ or $-$. Frequent subgraphs describing trends are also mined by the MINTAG algorithm [32], which takes undirected attributed graphs as input. In this case, vertices do not have only one numeric value but a set of values and each value corresponds

to one ordinal attribute. *Trend dynamic subgraph* is an induced subgraph of limited diameter, whose vertices follow the same trend over a subset of attributes. Different algorithm for fast motif mining in temporal networks is presented in [84]. These temporal networks possess edge timestamps but there are no attributes. The instances of each motif pattern must satisfy the same time order on the edges, i.e. isomorphism bijection preserves order of edges' timestamps. Furthermore, the timestamps of the motifs have to fit into a time window of a given size. A similar problem is solved by algorithm COMMIT [44] for motif mining in dynamic interaction networks. Each edge of these interaction networks possesses a timestamp but no duration. Motifs have to satisfy the following constraints imposed on edges. First, timestamp differences of adjacent edges in a motif cannot exceed a given timestamp threshold. Second, the isomorphism of two patterns requires the same time order on edges. For a given value of minimum support and timestamp threshold, the task is to find all motifs. COMMIT can also find top-k motifs with the highest support.

Closeness-based patterns. Trends in a graph express certain type of stability. The opposite, changes in a graph, is investigated in [62]. An algorithm for finding significant changing subgraphs is proposed for dynamic undirected graphs whose edges can be deleted or added. The idea of changing subgraphs is based on a closeness of vertices and on the assumption that only a part of the graph is changed between two consecutive snapshots. Specifically, the algorithm finds those subgraphs in which closeness of nodes is significantly changed between two consecutive snapshots by the change of edges. It is motivated by the fact that not only the change itself, but also the context is important in some scenarios.

DATA-PEELER. Considering general n-ary relations, the DATA-PEELER algorithm [20] is designed to find all closed patterns satisfying given piecewise (anti)-monotonic constraints. Informally, a constraint $\mathcal{C}$ is monotonic (antimonotonic) on an argument with attribute domain D if and only if $\forall E \subseteq D$ for which $\mathcal{C}$ holds it also holds for its supersets (subsets). Since dynamic graphs can be represented by ternary relations, i.e. adjacency matrices with time as the third dimension, DATA-PEELER is appropriate also for dynamic subgraph mining.

Durable patterns. Patterns appearing for the longest period of time in a dynamic graph are mined by an algorithm proposed in [94]. In particular, sequence of snapshots is used to represent a directed labelled graph evolving in time. Given such a sequence of graph snapshots, authors of the paper seek to find the most durable matches of an input graph pattern query. This means the matches that exist for the longest period of time. More specifically, they seek pattern matches with *largest continuous duration*, i.e. with largest number of consecutive snapshots in which the pattern occurs, and with *largest collective duration*, where snapshots containing the pattern occurrences do not have to be consecutive. This approach is further extended to mine only top-k most durable patterns [95].

Diversified subgraph patterns. The last algorithm for temporal subgraphs we present in this section is one searching for *diversified* patterns [112]. Each edge in this temporal network has a time interval instead of a timestamp associated with it. A pattern is diversified if the vertices of the pattern occurrences do not overlap too much. For each pattern, the algorithm also measures the longest time interval in which the pattern occurs according to the intervals of the input graph's edges. The overlaps and the duration of each pattern are measured by *coverage* and the algorithm searches for top-k diversified patterns with the highest coverage. Furthermore, the algorithm searches only for dense subgraphs, in which each vertex is adjacent to at least a given fraction of other vertices. In other words, a pattern is represented by a set of vertices and a time interval, during which the vertices closely interact with each other.

2.2.2 Rule Mining in Dynamic Graphs

This part presents several methods for rule mining in dynamic graphs. Given two graphs B and H, we say that $B \rightarrow H$ is a *graph rule*, in which B is the *body* (*antecedent*, *precondition*) and H is the *head* (*consequent*, *postcondition*) of the rule. Graphs B and H can be static graphs extended by timestamp functions defined earlier. Their embeddings are typically subgraphs of snapshots G_i and G_j of a dynamic graph $DG = (G_1, G_2, ..., G_n)$, where $1 \leq i, j \leq n$. In general, there can be arbitrary graphs in the body and the head, and the interpretation

of the rules may differ. For example, if both the antecedent and the consequent come from the same snapshot then the rule is commonly denoted as an *association* one and such rules can be used to examine subgraphs that appear together. On the other hand, if the antecedent's snapshot precedes the consequent's snapshot then the rule can be considered a *predictive* one. Rules of this type may help understand evolution processes in a dynamic graph.

Similarly to frequent subgraph mining, *support* can be defined for rules. Using the above definition of graph rules, one possible way is to count all distinct pairs of snapshots having the body and the head as subgraphs. There are several different definitions of support in literature, mostly adapted to a specific scenario. For example, rules in [13] express only edge additions and thus the body is a subgraph of the head. Support is calculated as the number of occurrences of the head by their algorithm. In order to be able to compute frequent patterns efficiently, support should be again defined in such a way as to be anti-monotonic, i.e. the support of a pattern is at least as large as the support of its arbitrary superpattern. Another useful measure for rules is *confidence*, expressed as the ratio of the rule support to the support of the body. Due to different definitions of support, it is possible to get different definitions of confidence.

GERM. One of the earliest approach for mining rules in dynamic graphs was proposed in [13]. Authors defined Graph Evolution Rules (GER), in which the same subgraph is used for the body and the head but all edges with maximum timestamps are removed from the body. Graph Evolution Rule Miner (GERM), a method for rule extraction, is designed for undirected graphs in which nodes and edges are only added and never deleted. This approach may be extended to cases with edge deletions if an edge appear and disappear at most once. Furthermore, it is assumed that node and edge labels do not change over time and that timestamps are assigned only to edges. New vertices can come only with new edges. GERM is based on frequent subgraph mining algorithm proposed in [18], which in turn is based on the gSpan algorithm [108]. In [17], authors used GERM for link (edge) prediction on real-world networks.

LFR-Miner. The previous algorithm mines rules expressing which edges will appear in an undirected graph in a future snapshot. Simi-

larly, the LFR-Miner algorithm [60] mines rules predicting a new edge between a pair of vertices in a directed graph. The body of a rule is again made up of a subgraph, but now there is a designated pair of vertices, called *start node* and *end node*. The head of a rule contains only a directed edge from the *start node* to the *end node*. Timestamps of all edges in the body have to be smaller than the timestamp of the edge in the head. Furthermore, all other nodes in the body have to be directly connected to both the *start node* and the *edge node*. If the body contains only the designated pair of vertices then there must exist an edge between them. The algorithm assumes that edges are only added and not removed.

Dynamic network motifs. Another method predicting appearance of an edge is by [55]. Specifically, method called *dynamic network motif analysis* is proposed on unlabelled directed dynamic networks. In order to find *dynamic network motifs*, the original graph $G = (V, E)$ is divided according two times $t_0 < t_1$ into two subgraphs $G_0 = (V, E_0)$ and $G_1 = (V, E_1)$, where E_0 and E_1 consist of edges with timestamp before t_0 and t_1, respectively. If an edge appears in the dynamic network multiple times at different times, only the last occurrence is considered. Dynamic network motif candidates are n-node subgraphs with one specific edge from E_1 and the remainder of the subgraph is a weakly connected subgraph of G_0. The method is designed only for 3-node motifs but it is possible to extend it to more nodes.

Link formation patterns. Rules capturing addition of an edge are also mined in [80]. It is assumed that nodes and edges are only added, and that node and edge labels never change. The rules are called *link formation patterns* and each such pattern consists of a connected subgraph b, which is called *base pattern*, and an extra link e that is built after the subgraph b is observed. In other words, b is a condition to form a new edge e. Edge e can connect either two nodes of b, or one node of b and one external node. Link formation patterns are constrained by two parameters. The first says how big the time span of the base patterns can be, i.e. the time difference between the first and the last added edges in a base pattern. The second says how big the time span between the added edge e and the maximal timestamp in the base pattern can be. There may also be several patterns having the same base pattern b differing only in the extra edges. Such patterns are

further analysed and it is examined how many occurrences of the base pattern these patterns share. Patterns sharing a significant amount of occurrences are called *correlation patterns*. The opposite ones are called *contrast patterns*.

Graph evolution DAG. An algorithm for rule mining from undirected labelled dynamic graphs, which is very similar to GERM [13], is presented in [69]. It is also based on gSpan [108] and can be regarded as a modification of GERM. The main difference is that it considers multigraphs. Multiedges are useful for graphs representing interactions as there can be several interactions between a pair of vertices during the time. After finding the rules, the algorithm creates a *graph evolution DAG* (directed acyclic graph) from these rules. Graphs from rule heads and bodies are used as vertices of this DAG, and a directed edge is created between two vertices if corresponding graphs form a rule. Such graph evolution DAG can be further examined. The algorithm also creates so called *abstract graph evolution DAG*. It is based on abstract patterns where timestamps were deleted and edges in rules are divided into two groups. The first group consists of edges with maximum timestamp, i.e. extra edges in the rule head, and the second group consists of other edges. This abstraction helps to unify a lot of patterns and thus to reduce the output to only interesting aspects.

Graph rewriting rules. Rule mining from biological networks in [117] presents *graph rewriting rules* that describe the evolution of two consecutive graphs in a graph sequence. First, the algorithm of the authors discovers the maximum common subgraph for such two consecutive graphs. Maximum common subgraph problem is NP-complete for general graphs [42], however, restriction to biological networks with unique vertex labels allows authors to use an algorithm with quadratic complexity [33]. After the maximum common subgraph is found, the remainder of the first graph is treated as *removal subgraph R* and the remainder of the second graph as *addition subgraph A*. Graph rewriting rule for graphs i and $i+1$ is then $GR_{i,i+1} = \{(R_i, C_{R_i}), (A_{i+1}, C_{A_{i+1}}\}$, where C_{R_i} and $C_{A_{i+1}}$ are sets of connection edges linking removal and addition subgraphs to original graphs, respectively. In other words, if you remove subgraph R from the first graph and add subgraph A, you get the second graph. Rewriting

rules are also used for description of repeated removals and additions in the form of *transformation rules*. They consist of a subgraph which is repeatedly removed and added, and parameters $+t_a$ and $-t_r$. $+t_a$ represents the time interval from the last removal to the current addition and $-t_r$ represents the time interval from the last addition to the current removal.

Tracking of node labels. Rules describing change of node labels are proposed in [25]. Labels are assigned to nodes by a clustering algorithm which considers local properties of the nodes, such as degree and clustering coefficient. Thus, the dynamics of the node labels captures the dynamics of the network structure. Clustering is used independently on each snapshot of the dynamic network. In this way, nodes from different snapshots but with the same characteristics may appear in different groups and thus may be assigned different labels. This undesirable feature is fixed by label unification for similar clusters. After the nodes are labelled, an itemset is created for each node in the form $\{t_{i_1} = j_1, ..., t_{i_n} = j_n\}$, where $t_{i_k} = j_k$ means that the node belonged to cluster j_k at time i_k. Association rule mining is then used and rules are filtered in such a way that the consequents have higher timestamps than antecedents. An example of a rule is $\{t_4 = 2, t_7 = 2\} \Rightarrow \{t_8 = 4\}$ saying that nodes that were in the second group at time 4 and 7 are likely to be in the fourth group at time 8.

Gear. A different view on dynamic graphs and an algorithm called Gear are proposed in [75]. Here, the dynamic graphs are seen as ternary relations and represented by boolean tensors of order 3. The first two dimensions are used for tail and head vertices, and the third one for the time. In other words, it is encoded as a sequence of adjacency matrices. It is assumed that the set of vertices is fixed. The Gear algorithm mines *inter-dimensional rules* in form $X \rightarrow Y$, where X and Y are n-ary and m-ary relations, respectively. Both relations are given on subsets of tensor dimensions, but these two subsets must be disjunctive. Dimensions that are not included in rules are used for computation of support. An example of a rule is $\{d_3\} \times \{a_3, a_4\} \rightarrow \{t_1, t_2\}$, where d_3 is from the dimension of tail vertices, a_3 and a_4 are from the dimension of head vertices, and t_1 and t_2 are from the time dimension. This

example rule expresses that if there are links from vertex 3 to vertices 3 and 4 then this is likely to happen at time 1 and 2.

Pinard. An extension of the previous approach to arbitrary n-ary relations, i.e. boolean tensors, is presented as Pinard algorithm [76]. This extension can also be considered as a generalization of association rule mining. Indeed, in a classical association rule mining setting [5] only two dimensions, typically marked as *Transactions* and *Products*, are considered, which corresponds to a binary relation. In this simple setting, rules contain only items from domain *Products* and domain *Transactions* serves for calculation of support. Unlike the Gear algorithm, Pinard is able to mine *multi-dimensional association rules*, which can contain the same domains in the head and the body. An example of such a rule is $\{d_3\} \times \{a_3, a_4\} \rightarrow \{d_2\}$, where the dimension of tail vertices is used both in the head and the body of the rule. The possibility of using n-ary relations can be beneficial for dynamic graphs. It is already clear from previous description how three dimensions are used. Other dimensions can represent node labels and other information. Further extension is the Pinard++ algorithm [77] combining both Gear and Pinard algorithms. It revises the way the rules are computed, the non-redundancy of rules, and the computation of rule confidence. Gear, Pinard, and Pinard++ are, however, restricted to conjunctive rules. Another extension is given in [78], where disjunctive rules are considered too. More specifically, an algorithm called Cidre is able to express disjunction in heads of rules. Such rules can express more information encoded in the analysed tensor. Nevertheless, all these tensor algorithms extract rules that are generally not *predictive*.

Temporal association rules. Continuing with association rules, another algorithm searching for this type of rules is presented in [71]. The antecedent and consequent graphs of the rule have to share a special node, called *focus node*, and also the timestamps of these graphs have to be bounded by a time interval. In the case of these association rules, the rule's graphs do not have to share the same snapshot but the focus node. A focus node and a time interval bound are part of the user's input as well as a limit on the number of edges in the rule and minimum support and confidence thresholds. The algorithm then searches for rules satisfying these constraints.

2.2.3 Sequence Graph Mining

Next, we consider *sequence graph mining*, where results are typically in the form of sequences, which can be seen as several concatenated rules or, in other words, the sequences of length two as rules. As the title suggests, sequence graph mining combines *sequence mining* [34] and *graph mining* [29]. Given a sequence or sequences of graphs on the input, the following methods are designed to output other, mostly shorter, sequences carrying some pieces of information about the original sequence or sequences. These smaller sequences typically consist of subgraphs or changes occurring in the original larger sequence of sequences.

We will use the following notion of a subsequence. Formally, we say that a sequence $\alpha = \alpha_1\alpha_2...\alpha_n$ is a *subsequence* of another sequence $\beta = \beta_1\beta_2...\beta_m$ and β is a *super-sequence* of α if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq m$ such that $a_1 = b_{j_1}, a_2 = b_{j_2}, \ldots, a_n = b_{j_n}$. This definition allows *gaps*, i.e. sequence α does not have to appear as a contiguous part of sequence β. However, if the gaps are not desired in a specific situation, we put $j_{k+1} = j_k + 1$ for all $k \in [1, n-1]$. In the case of graph mining, elements of sequences are graphs and subgraph isomorphism relation instead of equality relation is used between elements of a subsequence and elements of a given super-sequence, i.e. $a_k \sqsubseteq b_{j_k}$ for all $k \in [1, n]$.

A *frequent subsequence* is a subsequence for which *support* $\geq$ *minimum support*, where *minimum support* $\in [0, 1]$ is given and *support* is the fraction of input sequences in which the considered subsequence occurs. For cases with a single input sequence, *support* is the ratio of the number of occurrences of the subsequence to the length of the input sequence.

GTRACE. In particular, an algorithm for mining frequent sequences of changes in a set of dynamic graphs is GTRACE [50]. Graphs are assumed undirected, however, the principle is applicable to directed graphs as well. Idea of the algorithm is that each consecutive pair of graphs $g^{(j)}$ and $g^{(j+1)}$ in an input sequence of graphs $\langle g^{(1)}, g^{(2)}, ..., g^{(n)}\rangle$, called an *interstate sequence*, can be interpolated by a *intrastate sequence* $s^{(j)} = \langle g^{(j,1)}, g^{(j,2)}, ..., g^{(j,m_j)}\rangle$ and the original sequence can be represented by interpolations as $\langle s^{(1)}, s^{(2)}, ..., s^{(n-1)}\rangle$. The inter-

polation helps to divide large changes into smaller changes, called *transformations*. Specifically, there are six types of *transformation rules* which can be used to describe a change between each consecutive pair in an intrastate sequence: *vertex insertion, vertex deletion, vertex relabelling, edge insertion, edge deletion, edge relabelling*. Thus, each sequence of graphs can be represented compactly by a sequence of transformation rules, called *interstate transformation sequence*. GTRACE compiles the set of input graph sequences into a new set, where each element consists of the interstate transformation sequence and the first graph of the original sequence. Then it finds all *frequent transformation subsequences* (FTSs) from this new set. It also finds all *relevant* FTSs. A sequence of graphs is said to be *relevant* if its *union graph*, which is created from the union of vertices and the union of edges, is connected.

However, GTRACE is appropriate only for dynamic graphs with gradual changes, which means that only a small part of the graph is changed in one step. This is because large changes produce longer transformation sequences and with low minimum support the number of frequent subsequences to be mined increases significantly. To overcome this problem, a new method was designed [52]. This method mines *frequent subgraph subsequences* without gaps as we defined them earlier. Furthermore, each such subgraph is required to be induced and the sequence of those subgraphs to be *relevant* according to the definition in [50]. The method first transforms all input dynamic graphs into union graphs and then employs a frequent subgraph mining algorithm for static graphs on these union graphs. Frequent subgraph subsequences are those whose union graphs correspond to the frequent subgraphs found in the previous step. In comparison with transformation sequences, subgraph subsequences have also an advantage of better interpretability.

On the other hand, transformation rules can be also useful due to their compactness. Thus, GTRACE was extended to GTRACE2 [51] with increased performance. The main difference is that GTRACE2 uses union graphs that include vertex labels when computing relevant FTSs. Union graphs without labels in original GTRACE generated many more pattern combinations. Another enhancement brings GTRACE-RS [49] which mines directly the relevant FTSs and does not need to find the regular FTSs first. This is motivated by experiments, according to which 95% of FTSs mined by GTRACE are irrelevant.

Evolution patterns. Mining of repeating patterns, called *evolution patterns*, in a single sequence of graphs is presented in [87]. Graphs in the input sequence are assumed undirected and the set of nodes fixed. Edges may be either present or absent in each snapshot and this fact is captured by an occurrence sequence for each edge. An occurrence sequence consists of 1s and 0s where 1 denotes presence and 0 absence of the edge. For some edges, there may be an *occurrence rule*, which is a sequence of 1s and 0s that is repeated in the occurrence sequence of the edge. An example of an occurrence rule is 101 in an occurrence sequence 101101101. Finally, an *evolution pattern* is connected subgraph in which all edges share the same occurrence rule. It is necessary for the rule to be repeated at least s times, where s is a given parameter. The algorithm also mines *quasi-patterns*, in which the occurrence rule does not have to start from the beginning of the occurrence sequence.

CorSSS. Mining sequences of subgraphs from a single sequence of graphs is also performed by CorSSS algorithm [81]. Each such sequence of subgraphs is successive in the original sequence, i.e. each adjacent pair of subgraphs is also adjacent in the original sequence. Support of a sequence of subgraphs is the ratio of the number of occurrences of that sequence to the length of the original sequence. To exclude insignificant patterns, CorSSS enumerates only those sequences which are *correlated*, i.e. sequences that have similar consecutive subsequences.

Evolving induced relational states. A rather different type of sequences mined from a single graph sequence is presented in [6]. Here, the sequences to be mined are called *evolving induced relational states* (EIRS) and they are sequences of so called *induced relational states* (IRS). An IRS is an induced subgraph whose set of vertices and set of edges remain the same in at least ϕ consecutive snapshots, where ϕ is a parameter. In other words, this subgraph does not change in those snapshots. Furthermore, ratio of vertices that each pair of adjacent IRSs in an EIRS have in common must be at least β, where β is another parameter. This parameter ensures vertex similarity between adjacent IRSs. EIRSs are useful for capturing persistent states in a dynamic network and also for detecting sudden changes, which are expressed by a transition from one IRS to another.

Recurrent evolutions. We close this subsection with an algorithm for recurrent pattern mining in dynamic attributed graphs [24]. The algorithm looks for *recurrent evolutions*, where an *evolution* is a sequence of sets of vertices and this sequence is a subsequence of the input dynamic graph. An evolution is *recurrent* if there are multiple occurrences of this sequence in the input dynamic graph. Frequency of a pattern, i.e. of an evolution, is given by the number of its recurrences.

2.2.4 Anomaly patterns

Methods based on graph properties. Anomaly detection methods for dynamic graphs can be divided into several groups. One large group is comprised of methods that extract global characteristics, such as diameter, of the graphs as they evolve. The progress of these characteristics can be analysed as time series and anomalous graph snapshots or transitions between snapshots can be found. These methods typically exploit the structure of the graphs, not the labels or different attributes. Overview of these methods can be found in survey [7].

Tensor decomposition. Another large group of methods is based on tensor decomposition. In this case, a tensor is a generalization of an adjacency matrix. Additional dimensions store the information about time and vertex/edge attributes of the dynamic graph. Examples of algorithms from this group are STA [97], TensorSplat [57], ParCube [83], and MalSpot [66], among others. The idea of the decomposition methods is to compute factors of the given tensor and then examine the deviations from the common patterns described by main components. These methods allow us to observe anomalous patterns from the global viewpoint of various dimensions.

Anomalous communities. There are several methods that focus on subgraph structures and thus are more related to this thesis. The idea of these methods is to monitor graph communities or clusters instead of the whole graph. One such method is Com2 [9] that uses tensor decomposition and minimum description length principle for community detection in dynamic graphs. In comparison with the already mentioned decomposition methods, Com2 allows us to work on the level of communities. Thus, we can, for example, observe the sudden changes of interactions inside a community.

Another method for community-based anomaly detection in dynamic networks is presented in [22]. Authors define communities as the maximal cliques in a graph. The algorithm looks at communities that change over time and searches for anomalous communities. There are six types of anomalies, based on what change happened to a community: grown community, shrunken community, merged community, split community, born community, vanished community.

Anomalous subgraphs. Anomalous subgraphs in temporal graphs are also mined in [73]. These graphs are directed and represent computer communication. Each edge is associated with a time series, which expresses connection counts measured every minute. No other attributes are used. The algorithm is designed to discover communication patterns that deviate from historical patterns on a particular set of nodes. Deviating, i.e. anomalous, patterns are in the form of a path or a star and they help discover network intrusion in communication networks.

Anomalous network clusters in the form of connected subgraphs are also mined in [23]. In particular, a nonparametric statistical framework is used to model a social network as a *sensor* network, in which anomalousness levels of the neighbourhood-related attributes are measured for each node. It uses historical baseline distributions of the attributes and compares the current attribute values to those baseline distributions. This process is used to identify anomalous network clusters of nodes. The proposed algorithm works with heterogeneous dynamic networks, in which there are nodes and edges of different types. For example, there are node types such as *user*, *tweet*, and *hashtag* for a social network extracted from Twitter[2].

A Bayesian approach. A different approach is presented in [46]. Specifically, a Bayesian method is designed for detection of anomalous vertices. For each pair of vertices, the method models the communication between the vertices as a counting process. If the relationship has changed at some point in time and this change is statistically significant, the vertex pair is said to be anomalous. The method is able to model edges with labels too.

---

2. `https://twitter.com`.

Graph streams. Anomaly detection problem in graph streams is tackled in several works. Streams differ from regular dynamic graphs mainly in the fact that the history of the graph is typically not available in the case of streams. This property of streams also leads the methods to focus on real-time anomaly detection. Thus, the stream algorithms need to create a model or summaries of data for future use because they cannot return to the past data. One such algorithm is StreamSpot proposed in [65]. This algorithm compares two graph snapshots by a similarity measure that uses local substructrures of the graphs. The similarity measure is used to cluster the network flow and the clustering is dynamically maintained as the graph evolves. A graph snapshot is anomalous if it deviates significantly from the clusters.

Detection of deviations in current stream graph snapshot is also used by PLADS algorithm [38]. PLADS mines subgraph patterns and compares the patterns found in the current graph with the normative patterns found earlier. Patterns whose distance from normative patterns is high enough are marked as anomalous. The key idea of the algorithm is to compute new normative patterns only if the graph significantly changes.

Another outlier detection method for network streams is presented in [2]. This method uses a structural connectivity model in order to define outliers in graph streams. Moreover, it uses a structural reservoir-sampling approach to create partitions of vertices that maintain structural summaries of the underlying large networks. The algorithm focuses on unusual connectivity among different nodes. More specifically, it creates a structural generation model of edges with respect to node partitioning and then examines the edges in future snapshots to find anomalies.

Ensemble methods. An ensemble method for anomaly detection in temporal graphs called SELECT is presented in [89]. This method is a more general one and works also for general multi-dimensional data, i.e. no-graph data. In case of graph anomalies it searches for points in time at which the graph structure notably differs from its past. SELECT does not propose any particular anomaly detection methods but only combines existing methods in an ensemble manner.

2.2.5 Discriminative patterns

Discriminative pattern mining in dynamic graphs is rather a specific topic and, according to our best knowledge, there have not been published many works yet. The methods typically focus on discrimination of global states or attributes of the network snapshots.

An example of an algorithm for discriminative pattern mining in dynamic graphs is TGMiner [122], which mines discriminative subgraphs in a set of positive temporal graphs. It assumes that vertices and edges contain only categorical labels. It uses numerical timestamps, but the patterns must satisfy the same order of the timestamps as in the input graphs. Thus, it allows inexact mining to some degree, but it is restricted by preserving the order of timestamps. Moreover, the labels of the vertices and edges have to match exactly.

Another example is MINDS algorithm [88] for mining discriminative subgraphs from graph snapshots of a time-evolving network. Each snapshot of the network is assigned a global state and the patterns have to discriminate these states. From a technical point of view, the algorithm works with a set of static graphs classified into one of two classes.

Prediction of global numerical network states is also performed by SLR algorithm [30]. It mines a succinct set of subnetworks that are predictive and evolve along with the progression of the states. This means that the subnetworks do not change abruptly, but rather develop smoothly in the original network.

Predictive pattern mining [70] is close to discriminative pattern mining. This approach searches for patterns that are useful for a construction of a good predictive model. The main difference is that it searches for predictive patterns in an existing set of patterns. The approach is a more general one and does not deal with the pattern mining process itself.

2.3 Dynamic Graph Datasets

In this section we present four real-world and two synthetic datasets of dynamic graphs that were used in the subsequent chapters for experiments.

2.3.1 Resolution Proofs in Propositional Logic

The first dataset, shortly denoted as RESOLUTION, was obtained in an introductory course to logic [102]. Via a web-based tool, each of 351 students had to solve at least three resolution proofs randomly chosen from 19 assignments. Each such solution thus represents either a correct or incorrect resolution proof in propositional logic, stored as a dynamic graph. More specifically, the solutions are in the form of binary trees. Each vertex has a set of propositional-logic literals assigned as the label. The edges are directed and have the same label in these graphs, i.e. an empty string. The dynamic graphs capture the process of proof construction by students and they 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, only dynamic graphs of the same assignment share the set of vertex labels.

Among 873 collected solutions, 101 of them were classified as incorrect and 772 as correct. The most serious error in resolution is resolving on two literals. Other common errors in resolution proofs are the following: repetition of the same literal in the clause, incorrect resolution – the literal is missing in the resolved clause, resolving on the same literals (not on one positive and one negative), resolving within one clause, resolved literal is not removed, the clause is incorrectly copied, switching the order of literals in the clause, proof is not finished, intentional negation of literals in a clause. Information about errors that appeared in the proofs is also part of the data. Two examples of solutions are shown in Fig. 2.4.

2.3.2 ENRON

Another dataset, called ENRON, is based on the email correspondence in the Enron company [26]. We used a preprocessed version of this email traffic [86] for our experiments. Specifically, we used the version of data containing information about time, sender, receiver, and LDC topic. We created vertices of a dynamic graph from the set of senders and receivers. These vertices do not change through time. Each email message sent between a sender and a receiver is represented by an addition of a directed edge in the dynamic graph. If the graph already

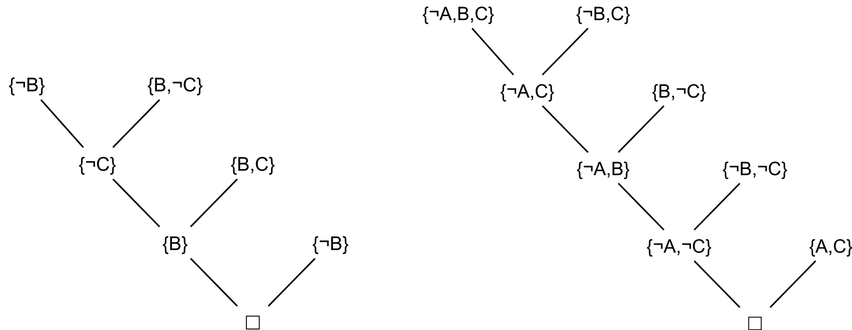


Figure 2.4: An example of a correct (on the left) and an incorrect (on the right) resolution proof.

contains the same edge, we just update its time of addition. Timestamps of the edges are represented as UNIX time, i.e. number of seconds since *epoch*. As there were messages with anomalous dates, we removed all messages sent before year 1998. We used LDC topics [86] 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. It is important to mention that approximately 67% of all edges were labelled as *outlier*. We used rank of employees [86] to label vertices. Vertices with unknown rank were removed from the graph and thus only 130 vertices and 62292 edges remained. Ranks and corresponding labels are available in Table 2.1.

2.3.3 Synthetic Dynamic Graphs

Besides real-world data, we also tested our methods on synthetic datasets. One of this dataset, SYNTH, represents a single dynamic graph and 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

Table 2.1: Ranks of employees in the ENRON dataset and the corresponding vertex labels.

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

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.

2.3.4 DBLP

The experiments were also performed on a undirected dynamic graph representing collaboration between scientists. This graph was created from data acquired from DBLP database[3]. We restricted the data to the following machine learning and data mining conferences: kdd, sigmod, www, vldb, sigir, icde, cikm, icml, nips, cvpr, iccv, pkdd, ecml, ida, pakdd, sdm. The created graph contains 53654 unlabelled vertices representing scientists and 455372 edges representing collaboration between scientists, where labels were given by the conferences and timestamps by the years.

3. http://dblp.uni-trier.de/

2.3.5 Phone Call Network

The last dataset, denoted as TELCO, was obtained from a partnership telecommunication company. A directed dynamic graph was created from phone calls of two consecutive months. From the whole graph we extracted a subgraph of 101718 vertices and 1629540 edges and used this subgraph for the experiments. The vertices represent the customers of the company and other people who called with the customers. Edges contain one numeric attribute – call duration in seconds logarithmically transformed and scaled into 0-1 range. Vertices have no attributes. Timestamps of the edges were represented as UNIX time, i.e. the resolution is in seconds. Due to privacy reasons, we are not allowed to give more details regarding this dataset.

3 Frequent Pattern Mining

The result of patter mining in dynamic graphs does not have to be in the form of general graphs. Indeed, some algorithms may output simpler structures, such as sequences, which can be still useful for various purposes. This chapter presents two methods that mine special kind of patterns and application of these methods on resolution proof graphs. First, we show a method for mining of *generalized patterns* that employs domain knowledge about the resolution proofs. Briefly, the method unifies labels that have the same meaning but are not necessarily identical. We show that these generalized patterns are useful for classification of resolution proofs and outlier detection. Then we show how sequence pattern mining can be used for clustering of resolution proofs.

3.1 Domain-specific Pattern Mining

Labels of vertices and edges are simple and repetitive in common real-world graphs. This is not the case of resolution proof graphs in propositional[1] logic, in which vertices are labelled by clauses represented by lists or sets of literals. An example of such a label is "$\{A, \neg B, \neg C\}$". Classical pattern mining methods test labels on string equality in order to decide whether to create a common pattern from them. Simple string comparison on an enumerated list or set of literals may thus be rather restrictive and the methods may miss some patterns due to discrepancies introduced by different order of literals, different variable names, etc.

In this section we present a method for analysis and evaluation of resolution logic proofs constructed by undergraduate students [102]. The data contains tree structures of the proofs and also temporal information about all actions that students performed, e.g. a node insertion into a proof or its deletion, drawing or deletion of an edge, or a text manipulation. We present a novel method for multi-level generalisation of subgraphs that is useful for characterisation of logic proofs. We

1. We restrict ourselves to analysis of proofs in propositional logic, although a similar approach could be used for predicate logic too.

use this method for feature construction and perform classification and class-based outlier detection on logic proofs represented by these new features. We show that the method helps to find unusual students' solutions and to improve semi-automatic evaluation of the solutions.

Note that a search method can be used in order to find errors in the resolution proofs. However, detection of an error does not necessarily mean that the whole solution is completely incorrect from the perspective of the teacher. Moreover, by a search we can hardly discover patterns, or sequence of patterns, that are typical for wrong solutions.

Up to our knowledge, there is no work on analysis of student solutions of logical proofs by means of graph mining. Definitely, solving logic proofs, especially by means of resolution principle, is one of the basic graph-based models of problem solving in logic. In problem-solving processes, graph mining has been used in [116] for mining concept maps, i.e. structures that model knowledge and behaviour patterns of a student, for finding commonly observed subconcept structures. Combination of multivariate pattern analysis and hidden Markov models for discovery of major phases that students go through in solving complex problems in algebra is introduced in [8]. Markov decision processes for generating hints to students in logic proof tutoring from historical data has been used in [11, 12, 96].

3.1.1 Data

Data used for the experiments contained 873 resolution proofs as described in section 2.3.1. The most common error in these solutions is simultaneous resolution on two literals and because it is referred later in text, we denote this error as *E3*. There were 303 different clauses, i.e. vertex labels, occurring in 7869 vertices in the data, see frequency distribution in Fig. 3.1. Approximately half of the clauses had absolute frequency less than or equal to three.

3.1.2 Generalized Subgraphs

In this subsection we describe a new feature construction method from graph data. Representing a graph only by values of its vertices and edges is insufficient as the structure of the graph also plays a significant role. Common practice is to use substructures of graphs as new

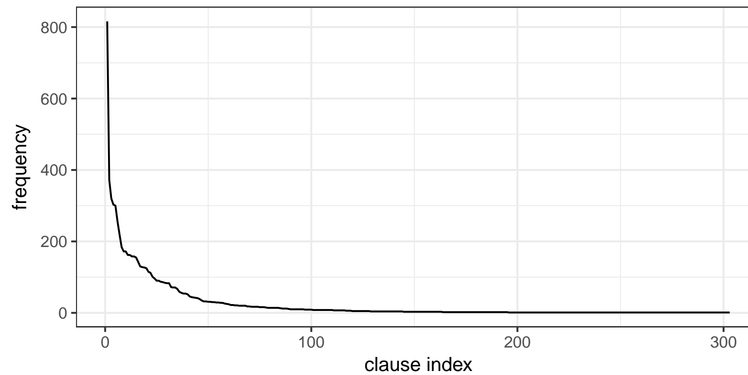


Figure 3.1: Distribution of clause labels ordered by frequency.

features [29]. More specifically, a boolean feature is created for each substructure and the value of the feature depends on whether the corresponding substructure occurs in the given graph instance or not. However, as we stated earlier, the dataset contains 19 different resolution assignments and different assignments can contain different variable names. Moreover, the sets of literals can be written in arbitrary order although technically they are equal. Another problem with classical pattern mining algorithms is that it is often not possible to mine infrequent patterns, usable for anomalies, efficiently. That is because a very low minimum support threshold causes generation of excessive number of frequent patterns.

To overcome the discussed problems, we created a new method for feature construction from our data. The idea of feature construction is to unify subgraphs which carry similar information even though they differ in form. An example of two subgraphs, which differ only in variable letters and ordering of nodes and literals, is shown on the left side of Fig. 3.2. The goal is to process such similar graphs to get one unique graph, as shown in the same figure on the right. In this way, we can better deal with different sets of clauses with different sets of variable letters. To deal with the minimum-support problem, mining of general frequent subgraphs was left out completely and all 3-node subgraphs, which are described later, were looked up.

Unification on Subgraphs. To unify different tasks that may appear in student tests, we defined a unification operator on subgraphs

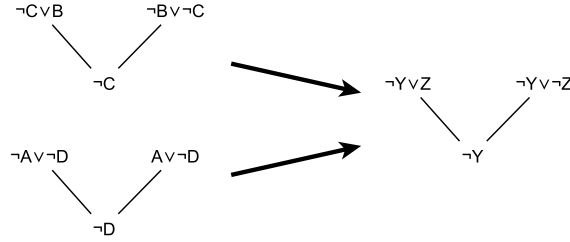


Figure 3.2: An example of pattern unification.

that allows us to find so called *generalized subgraphs*. Briefly saying, a generalized subgraph describes a set of particular subgraphs, e.g. a subgraph with parents $\{A, \neg B\}$ and $\{A, B\}$ and with the child $\{A\}$ (the result of a correct use of the resolution rule), where A, B, C are propositional letters, is an instance of generalized graph $\{Z, \neg Y\}$, $\{Z, Y\} \rightarrow \{Z\}$, where Y and Z are variables. An example of incorrect use of resolution rule $\{A, \neg B\}$, $\{A, B\} \rightarrow \{A, A\}$ matches with the generalized graph $\{Z, \neg Y\}, \{Z, Y\} \rightarrow \{Z, Z\}$. In other words, each subgraph is an instance of one generalized subgraph. We can see that the common set unification rules [35] cannot be used here. In this work we focused on generalized subgraphs that consist of three nodes, two parents and their child. Then each generalized subgraph corresponds to one way—correct or incorrect—of resolution rule application.

Ordering on Nodes. As a resolution proof is, in principal, an unordered tree, there is no order on parents in those three-node graphs. To unify two resolution steps that differ only in order of parents we need to define an ordering on parent nodes[2]. We take a node and for each propositional letter we first count the number of negative and the number of positive occurrences of the letter, e.g. for $\{\neg C, \neg B, A, C\}$ we have these counts: $(0, 1)$ for A, $(1, 0)$ for B, $(1, 1)$ for C. Following the ordering Ω defined as follows: $(X, Y) \leq (U, V)$ if and only if $(X < U \vee (X = U \wedge Y \leq V))$, we have a result for the node $\{C, \neg B, A, \neg C\}$: $\{A, \neg B, C, \neg C\}$ with description $\Delta = ((0, 1), (1, 0), (1, 1))$. We will

2. Ordering on nodes, not on clauses, as a student may write a text that does not correspond to any clause, e.g. $\{A, A\}$.

compute this transformation for both parent nodes. Then we say that a node is smaller if the description Δ is smaller with respect to the Ω ordering applied lexicographically per components. Continuing with our example above, let the second node be $\{B, C, A, \neg A\}$ with $\Delta = ((0,1),(0,1),(1,1))$. Then this second node is smaller than the first node $\{A, \neg B, C, \neg C\}$, since the first components are equal and $(1,0)$ is greater than $(0,1)$ in case of the second components.

Generalization of Subgraphs. Now we can describe how the generalized graphs are built. After the reordering introduced in the previous paragraph, we assign variables $Z, Y, X, W, V, U, \ldots$ to propositional letters. To accomplish this, we initially merge literals from all nodes into one list and order it using the Ω ordering. After that, we assign variable Z to the letter with the smallest value, variable Y to the letter with the second smallest value, etc. If two values are equal, we compare the corresponding letters only within the first parent, alternatively within the second parent or child. For example, let a student's (incorrect) resolution step be $\{C, \neg B, A, \neg C\}, \{B, C, A, \neg A\} \to \{A, C\}$. We order the parents getting the result $\{B, C, A, \neg A\}, \{C, \neg B, A, \neg C\} \to \{A, C\}$. Next we merge all literals into one list, keeping multiple occurrences: $\{B, C, A, \neg A, C, \neg B, A, \neg C, A, C\}$. After reordering, we get $\{B, \neg B, C, C, C, \neg C, A, A, A, \neg A\}$ with $\Delta = ((1,1), (1,3), (1,3))$. This leads to the following renaming of letters: $B \to Z$, $C \to Y$, and $A \to X$. Final generalized subgraph is $\{Z, Y, X, \neg X\}, \{Y, \neg Z, X, \neg Y\} \to \{X, Y\}$. In case that one node contains more propositional letters and the nodes are equal (with respect to the ordering) on the intersection of propositional letters, the longer node is defined as greater. At the end, the literals in each node are lexicographically ordered to prevent from duplicities such as $\{Z, \neg Y\}$ and $\{\neg Y, Z\}$.

Complexity of Graph Pattern Construction. Complexity of pattern generalization depends on the number of patterns and the number of literals within each pattern. Let r be the maximum number of literals within a 3-node pattern. In the first step, ordering of parents must be done, which takes $O(r)$ for counting the number of negative and positive literals, $O(r \log r)$ for sorting and $O(r)$ for comparison of two sorted lists. Letter substitution in the second step consists of counting literals on merged list in $O(r)$, sorting the counts in $O(r \log r)$

and renaming of letters in $O(r)$. Lexicographical reordering is performed in the last step and takes $O(r \log r)$. As construction of advanced generalized patterns is less complex than the construction of patterns mentioned above, we can conclude that the time complexity for whole generalization process on m patterns with duplicity removal is $O(m^2 + m(4r + 3r \log r))$.

Higher-level Generalization. To improve performance of used algorithms, e.g. outlier detection algorithms, we created one more generalization method. This method can be viewed as a higher-level generalization as it generalizes the method described in previous paragraphs. This method uses domain knowledge about general resolution principle. It goes through all literals in a resolvent and deletes those which also appear in at least one parent. Each such literal is also deleted from the corresponding parent or parents in case it appears in both of them. In the next step, remaining literals in parents are merged into a new list *dropped* and remaining literals in the resolvent form another list, *added*. These two lists form a pattern of the higher-level generalization and we will write such patterns in the following format:

$$\underset{\text{(added)}}{[L_{i_1}, L_{i_2}, ..., L_{i_n}]}; \underset{\text{(dropped)}}{[L_{j_1}, L_{j_2}, ..., L_{j_m}]}$$

For example, if we take the generalized subgraph from the right side of Fig. 3.2, there is only one literal in the resolvent, $\neg Y$. We remove it from the resolvent and both parents and we get $dropped = [Z, \neg Z]$, $added = []$.

As a result, there may be patterns which differ only in used letters and order of literals in lists *dropped* and *added*. For this reason, we then apply similar method as in the lower-level generalization. Specifically, we merge lists *dropped* and *added* and compute number of negative and positive literals for each letter in this new list. The letters are then ordered primarily according to the number of occurrences of negative literals and secondly according to the number of occurrences of positive literals. In case of tie we check ordering of affected letters only in *added* list and if needed, then also in *dropped* list. If tie occurs also in these lists, then the order does not matter.

At the end, the old letters are one by one replaced by the new ones according to the ordering and the new lists are sorted lexicographically. For example, let $dropped = [X, \neg X]$, $added = [Y, Z, Z, \neg Z]$. Then $merged = [X, \neg X, Y, Z, Z, \neg Z]$ and the number of occurrences can be listed as $count(X, merged) = (1, 1)$, $count(Y, merged) = (0, 1)$, and $count(Z, merged) = (1, 2)$. Ordering on letters can be expressed as $Y \leq X \leq Z$. Using letters from the end of alphabet we perform following substitution according to created ordering: $Y \rightarrow Z$, $X \rightarrow Y$, $Z \rightarrow X$. Final pattern will have lists $dropped = [\neg Y, Y]$, $added = [\neg X, X, X, Z]$, provided that $\neg$ sign is lexicographically before alphabetic characters. Examples of patterns with absolute support ≥ 10 are shown in Tab. 3.1.

Table 3.1: Higher-level patterns with support ≥ 10.

Pattern ($added; dropped$)	Support
$[]; [\neg Z, Z]$	3345
$[]; [\neg Y, \neg Z, Y, Z]$	59
$[\neg Z]; [\neg Y, Y]$	18
$[]; [\neg Z]$	13
$[]; []$	10

Generalization Example. In this subsection we illustrate the whole generalization process on an example. Assume that the following 3-node subgraph has to be generalized:

$$P1 = \{\neg C, \neg A, \neg C, D, \neg D\}, P2 = \{\neg D, \neg A, D, C\} \rightarrow \{\neg A, A, \neg C\}$$

First, the parents are checked and possibly reordered. For each letter we compute the number of negative and positive literals in either parent. Specifically, $count(A,\ P1) = (1, 0)$, $count(C,\ P1) = (2, 0)$, $count(D,\ P1) = (1, 1)$, $count(A,\ P2) = (1, 0)$, $count(C,\ P2) = (0, 1)$, and $count(D,\ P2) = (1, 1)$. Obtained counts are lexicographically sorted for both parents and both chains are lexicographically compared:

$$((1, 0), (1, 1), (2, 0)) > ((0, 1), (1, 0), (1, 1))$$

In this case, the result was already obtained by comparing the first two pairs, $(1,0)$ and $(0,1)$. Thus, the second parent is smaller and the parents should be switched:

$$P1' = \{\neg D, \neg A, D, C\}, P2' = \{\neg C, \neg A, \neg C, D, \neg D\} \rightarrow \{\neg A, A, \neg C\}$$

Now, all three nodes are merged into one list:

$$S = \{\neg D, \neg A, D, C, \neg C, \neg A, \neg C, D, \neg D, \neg A, A, \neg C\}$$

Once again, the numbers of negative and positive literals are computed: $count(A,\ S) = (3,1)$, $count(C,\ S) = (3,1)$, $count(D,\ S) = (2,2)$. Since $count(A,\ S) = count(C,\ S)$, we also check the counts in the first parent, $P1'$. Because $count(C,\ P1') = count(C,\ P2) < count(A,\ P2) = count(A,\ P1')$, letter C is inserted before A. Finally, the letters are renamed according to the created order: $D \rightarrow Z, C \rightarrow Y, A \rightarrow X$. After the renaming and lexicographical reordering of literals, we get the following generalized pattern:

$$\{\neg X, \neg Z, Y, Z\}, \{\neg X, \neg Y, \neg Y, \neg Z, Z\} \rightarrow \{\neg X, \neg Y, X\}$$

Next, we want to get also the higher-level generalization of that pattern. The procedure goes through all literals in the resolvent and deletes those literals that occur in at least one parent. This step results in a prunned version of the pattern:

$$\{\neg Z, Y, Z\}, \{\neg Y, \neg Z, Z\} \rightarrow \{X\}$$

Parents from the pruned pattern are merged into a new list *dropped* and the resolvent is used in a list *added*. Thus, $added = [X]$ and $dropped = [\neg Z, Y, Z, \neg Y, \neg Z, Z]$. Now it is necessary to rename the letters once again. Lists *added* and *dropped* are merged together and the same subroutine is used as before—now the lists can be seen as two nodes instead of three. In this case, the renaming goes as follows: $X \rightarrow Z, Y \rightarrow Y, Z \rightarrow X$. At the end, literals in both lists are lexicographically sorted and the final higher-level pattern is:

$$\underset{\text{(added)}}{[Z]};\ \underset{\text{(dropped)}}{[\neg X, \neg X, \neg Y, X, X, Y]}$$

3.1.3 Use of Generalized Subgraphs

This subsection puts all the information from previous paragraphs together and describes how generalized patterns are used as new features. Input data in form of nodes and edges are transformed into attributes of two types. Generalized patterns of the lower level can be considered as the first type and the patterns of higher-level generalization as the second type. One boolean attribute is created for each generalized pattern. Value of such an attribute is equal to $TRUE$, if the corresponding pattern occurs in the given resolution proof, and it is equal to $FALSE$ otherwise. This representation allows us to use a lot of existing machine learning algorithms.

3.1.4 Classification of Resolution Proofs

As we stressed in the introduction, this method has not been developed for recognition of correct or incorrect solutions. However, to verify that the feature construction is appropriate, we learned several models for classification into *correct* and *incorrect* classes. We evaluated the models by 10-fold cross validation and we obtained the best result for SMO Support Vector Machines from Weka [45], which had 95.2% accuracy, see Table 3.2. Precision and recall for the class *incorrect* were 0.94 and 0.61, respectively. Minimum support for pattern selection was 0% in this case. To improve performance of classification we used the new level of generalization. Using the same settings, but now with both levels of generalized patterns, we achieved 96.9% accuracy, 0.95 precision and 0.74 recall for the class *incorrect*. Similar results were obtained when only the new level of generalization was used, again with SMO. When ordered according to precision, value 0.98 was achieved by J48, but the accuracy and recall were only 96.1 and 0.68, respectively.

As one of the most common errors in resolution proofs is usage of resolution rule on two pairs of literals at the same time, we repeated the experiment, but now discarding all patterns capturing this specific kind of error. In this scenario the performance slightly dropped but remained still high—J48 achieved 95.4% accuracy, 0.87 precision and 0.72 recall, see the last row in Table 3.2. For the sake of complete-

Table 3.2: Classification results for frequent subgraphs. Precision and recall are stated for the class *incorrect*.

Attributes	Algorithm	Accuracy [%]	Precision	Recall
low-level gen.	SMO	95.2	0.94	0.61
both levels	SMO	96.9	0.95	0.74
both levels	J48	96.1	0.98	0.68
both levels (E3)	J48	95.4	0.87	0.72

ness, F1 score for the class *correct* varied between 0.97 and 0.99 in all of the results above.

3.1.5 Detection of Outlying Resolution Proofs

Mining Class Outliers. In this section we present the main result, obtained from outlier detection. We observed that student creativity is more advanced than ours and that results of the queries for error detection must be used carefully. Detection of anomalous solutions—either abnormal, with picturesque error, or incorrectly classified—helps to improve the tool for automatic evaluation, as will be shown later.

Here we focus only on outliers for classes created from error E3, the resolution on two literals at the same time, as it was the most common error. This means that the data can be divided into two groups, depending whether the instances contain error E3 or not. For other types of errors, the analysis would be similar. We also present results computed only on higher-level generalized patterns. The reason is that they generally achieved much higher outlier scores than lower-level patterns.

The data we processed had been labelled. Unlike in common outlier detection, where we look for outliers that differ from the rest of *normal* data, we needed to exploit information about a class. That is why we used weka-peka [85], the predecessor of RF-OEX [74], that looks for class outliers [48, 82] using Random Forests (RF) [15]. The main idea of weka-peka lies in different computation of proximity matrix

in RF—it also exploits information about a class label [85]. We used the following settings:

```
NumberOfTrees=1000
NumberOfRandomFetaures=7
FeatureRanking=gini
MaxDepthTree=unlimited
Bootstrapping=yes
NumberOfOutliersForEachClass=50
```

Main results of outlier detection process are summarized in Table 3.3. When analysing the strongest outliers that weka-peka found, we can see that there are three groups according to the outlier score. The two most outlying examples, instances numbered 270 and 396, significantly differ from the others. The second cluster consists of four examples with the outlier score between 50 and 100, and the last group is comprised of instances with the lowest score of 15.91.

As weka-peka is based on Random Forest, we can interpret an outlier by analysing trees that classify given instance to a different class than it was labelled. Such trees show which attribute or combination of attributes lead to the resulting class. If we search for repeating patterns in those trees, we can find the most important attributes making the given instance an outlier. Using this method to interpret the instance 270, we found out that high outlier score is caused by not-applying one specific pattern, see Table 3.3. When setting this attribute equal $TRUE$, outlier score decreases to -0,40. Values of attributes of instances 396 and 270 are equal, which means that the interpretation is the same as in the previous case. Similarly, we found that outlierness of instance 236 is given by occurrence of specific pattern in solution and non-occurrence of another pattern. The value of the corresponding attribute is the only difference between instance 236 and 187. Occurrence/non-occurrence of this pattern is therefore the reason why instance numbered 236 achieves higher outlier score than instance 187. See again Table 3.3 for information about particular patterns. We further elaborated this approach of outlier explanation in the following section.

Finding Significant Patterns. As the outlier score is the only output information about the outliers, we created a simple method for finding the attributes with the most unusual values. Let x_{ij} denote the value

Table 3.3: Top outliers for data grouped by error E3.

Instance	Error E3	Outlier score	Significant patterns [(AScore) *added*; *dropped*]	Significant missing patterns [(AScore) *added*; *dropped*]
270	no	131.96	(0.96) *looping*	(-0.99) $[];[\neg Z, Z]$
396	no	131.96	(0.96) *looping*	(-0.99) $[];[\neg Z, Z]$
236	no	73.17	(0.99) $[];[\neg Y, \neg Z, Y]$	
187	no	61.03	(0.99) $[\neg Z];[\neg Y, Y]$ (0.99) $[];[\neg Y, \neg Z, Y]$	
438	yes	54.43	(1.00) $[Z];[\neg X, \neg Y, X, Y]$	(-0.94) $[];[\neg Y, \neg Z, Y, Z]$
389	yes	52.50	(1.00) $[];[\neg Y, \neg Z, Y]$	(-0.94) $[];[\neg Y, \neg Z, Y, Z]$ (-0.81) $[];[\neg Z, Z]$
74	yes	15.91	(0.98) $[\neg Z];[\neg X, \neg Y, X, Y]$ (0.98) $[];[\neg X, \neg Y, \neg Z, X, Y, Z]$	(-0.94) $[];[\neg Y, \neg Z, Y, Z]$
718	yes	15.91	(0.98) $[\neg Z];[\neg X, \neg Y, X, Y]$ (0.98) $[];[\neg X, \neg Y, \neg Z, X, Y, Z]$	(-0.94) $[];[\neg Y, \neg Z, Y, Z]$

of the jth attribute of the ith instance, which is either $TRUE$ or $FALSE$ for the pattern attributes, and $cl(i)$ denote the class of the ith instance. Then for instance i we compute the score of attribute j as:

$$AScore(i,j) = \begin{cases} \frac{|\{k|k \neq i \wedge cl(i)=cl(k) \wedge x_{kj}=FALSE\}|}{|\{k|k \neq i \wedge cl(i)=cl(k)\}|} & \text{if } x_{ij} = TRUE \\ -\frac{|\{k|k \neq i \wedge cl(i)=cl(k) \wedge x_{kj}=TRUE\}|}{|\{k|k \neq i \wedge cl(i)=cl(k)\}|} & \text{if } x_{ij} = FALSE \end{cases}$$

AScore expresses the proportion of other instances from the same class which have different value of the given attribute. If outlier's attribute equals $FALSE$, then the only difference is in the sign of the score. For example, consider our data set of 873 resolution proofs, out of which 53 proofs contain error E3. Assume that one of the 53 proofs is an outlier with an attribute equal to $TRUE$ and from the rest of 52 proofs only two proofs have the same value of this attribute as the outlier. Then the outlier's AScore on this attribute is approximately $50/52 = 0.96$ and it indicates that the value of this attribute is quite unusual.

In general, AScore ranges from -1 to 1. If the outlier resolution graph contains a pattern which is unique for the class of the graph, then AScore of the corresponding attribute is equal to 1. On the other hand, if the outlier misses a pattern and all other graphs contain it, then AScore is equal to -1. AScore equal to 0 means that all other instances are equal to the outlier on the specified attribute.

Interpretation of the Outliers. By using AScore metrics we found the patterns which are interesting for outliers in Table 3.3. Patterns, with AScore > 0.8 are listed in the *significant patterns* column and patterns with AScore < -0.8 in the *significant missing patterns* column.

All outliers from Table 3.3, except for the last one as it is almost identical to the penultimate one, are also displayed in Fig. 3.3. Analysis of individual outliers let us draw several conclusions. Let us remind that higher-level patterns listed in Table 3.3 are derived from lower-level patterns consisting of three nodes, two parents and one resolvent, and that the component *added* simply denotes literals which were added erroneously to the resolvent and the component *dropped* denotes literals from parents which participated in the resolution process. Two most outlying instances, numbered 270 and 396, also contain one specific pattern, *looping*. This pattern represents the ellipsis in a resolution tree, which is used for tree termination if the tree cannot lead

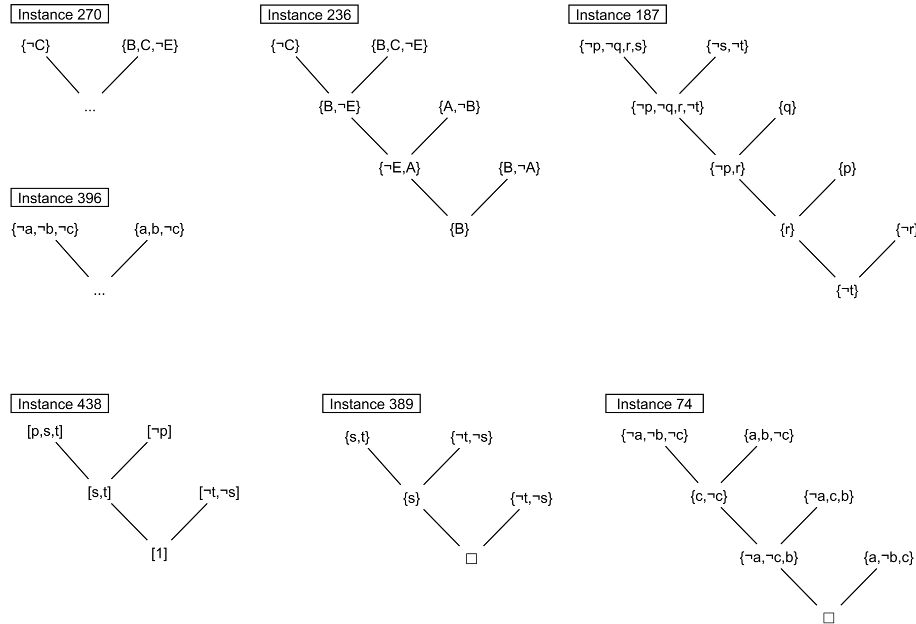


Figure 3.3: Drawings of the outlying instances from Table 3.3.

to a refutation. Both instances contain this pattern, but neither of them contains the pattern of correct usage of the resolution rule, which is listed in the *significant missing patterns* column. The important thing is that these two instances do not contain error E3, but also any other error. In fact, they are created from an assignment which always leads to the *looping* pattern. This shows that it is not sufficient to find all errors and check the termination of proofs, but we should also check whether the student performed at least few steps by using the resolution rule. Otherwise we are not able to evaluate the student's skills. Moreover, there may be situations in which a student only copies the solution.

Instances with the outlier score less than 100 are less different from other instances. In particular, instances number 236 and 187 are more similar to correct proofs than the instances discussed above. Yet, they both contain anomalous patterns such as $[]; [\neg Y, \neg Z, Y]$. This particular error pattern does not indicate error E3, as can be seen in Table 3.3. It is actually not marked as any type of error, which tells us

that it is necessary to extend our list of potential errors in the automatic evaluator.

Continuing with outlier instances we get to those which contain error E3. Two of them exceed the boundary of outlier score 50, which suggests that they are still relatively anomalous. The first outlier, instance number 438, differs from other instances in an extra literal which was added into a resolvent. Specifically, the number 1, which is not even a variable, can be seen at the bottom of the resolution proof in Fig. 3.3. More interesting is the second instance with number 389. Error E3 was detected already in the first step of the resolution, specifically when resolved on parents $\{s, t\}$ and $\{\neg t, \neg s\}$. This would not be a strange thing, if the resolvent was not s. Such a resolvent raises a question whether it is an error of type E3 or just a typing error. The latter is a less serious error.

Last two outliers in the table are almost the same so only the instance number 74 is depicted in Fig. 3.3. These two instances have quite low outlier score and they do not expose any shortcomings of our evaluation tool. Yet, they exhibit some outlying features such as resolving on three literals at the same time.

3.1.6 Discussion

In this section we presented generalization methods for subgraphs of resolution proof trees built by students. Generalized subgraphs created by these special graph mining methods are useful for representation of logic proofs in an attribute-value fashion.

This representation was used for learning classification models, which performed well on the data. In past [100, 101], we experimented also with simple frequent patterns generated by Sleuth algorithm [120], which aims at finding frequent patterns in unordered rooted trees. Unfortunately, there are several drawbacks if the generalization is not used and only regular frequent patterns are used. First, as there are different assignments with differently named propositional variables, the patterns cannot simply cover solutions from different assignments. Next, usage of such an algorithm is quite limited, because by setting the minimum support to a very small value, the algorithm may end up generating excessively many frequent subtrees, which consumes both time and space. The problem is that we wish to include the infrequent

substructures as well because they often represent mistakes in students' solutions and thus may be helpful when searching for anomalies. On the other hand, when we use only generalized patterns, it is enough to extract the 3-node patterns that are then generalized.

We further showed how a class-based outlier detection method can be used on these logic proofs by utilization of the generalized subgraphs. We also discussed how the outlying proofs may be used for performance improvement of our automatic proof evaluator. For as we observed, it is not sufficient to detect only the errors but we need to analyse the context in which an error appeared. Moreover, there are solutions that are erroneous because they do not contain a particular pattern or patterns. Outlier detection helped to find wrong students' solutions that could not be detected by the system of queries even though the set of queries has been carefully built and tested on a test data. We also found a situation when a query did not detected an error although it appeared in the solution. We are convinced that with increasing number of solutions we will be able to further increase performance of wrong solution detection. This method may also be used for other types of data such as tableaux proofs.

3.2 Sequence Mining on Dynamic Graphs

We showed how resolution proofs can be represented by simple subgraphs in the previous section. Now we present a different representation of resolution proofs and its usage for a different task [99]. Specifically, we exploit the dynamic nature of the solution creation process and represent it by sequences of events. As each event, such as addition of a node or an edge, has an exact timestamp, we can use timestamps to order the events into sequences. Having the solutions of resolution proofs in the form of sequences, we employ sequence mining techniques to find frequent patterns and use these patterns to cluster different types of solutions. Such clusters allow us to further understand the solving processes of students and perform more detailed analyses of particular solving styles.

There is also research similar to ours in the literature. For example, authors of [47] analysed students' ordinary handwritten coursework with a digital pen by means of sequence mining techniques to identify

patterns of actions that are more frequently exhibited by either good- or poor-performing students. They used sequence patterns as features for a linear regression model and predicted students' performance by this model. In [67], sequential pattern mining was used for analysis of activity around an interactive tabletop and finding frequent sequences that differentiate high achieving from low achieving groups. These approaches are different from ours as we use sequence mining to find various strategies of solving the problems rather than for students' performance prediction.

3.2.1 Resolution Proofs as Sequences

For processing temporal information, we used two sequence representations of the data. Each resolution proof can be represented as a sequence in either representation. The first one is composed of two types of events: addition of a node, i.e. clause, into a proof and addition of an edge. An example of such a sequence is CCCCCEEEE, where C denotes a node addition and E denotes an edge addition. The second representation uses the same events as the first one, but it also contains events of text modification[3]. An example of a sequence in the second representation is CTCTCTCTCTEEEE, where elements C, T and E denote a node addition, a text modification and an edge addition, respectively. From now on, we will use CE and CET abbreviations for the representations.

As sequences cannot be processed by machine learning algorithms that are typically used for classification, clustering or outlier detection, we need to transform the data. Simple and common practice is to use subsequences as features [34]. This is the same approach as we used in Section 3.1.3 with subgraphs. We considered only subsequences consisting of elements that are consecutive in original sequences, i.e. without *gaps*, because subsequences with gaps are not descriptive in case of our sequences. Recall the definition of subsequences without gaps from Section 2.2.3.

In order to find all potentially useful subsequences, we employed cSpade algorithm [119] for frequent sequence mining. For a given value $min_support \in [0, 1]$, this algorithm finds all subsequences

3. We also tried sequences with node- and edge-deletion events, but these events did not affect the results due to their sparse occurrences.

whose $support \geq min_support$. Support of a subsequence α is a fraction of input sequences which contain α as a subsequence. Specifically, we set $min_support = 0.1$ to get only subsequences that occur in at least 10% of all input sequences. We obtained 121 frequent subsequences from sequences in CE representation, and 242 subsequences in case of CET representation. Each frequent subsequence is used as a new feature with value equal to 1 if the subsequence appears in the given sequence, and 0 otherwise.

3.2.2 Sequence Clustering

Having the resolution proofs represented by features constructed from the two representations of sequences, we performed clustering on features of each representation. For the purpose of clustering, we set the values of features as follows: if a sequence contains a subsequence corresponding to the considered feature, we set the feature value of the sequence to the squared length of the subsequence. Otherwise we set the value to 0. The rationale for this is that long subsequences should be more explanatory so they carry more weight.

Using this representation of data we performed cluster analysis using AGNES (AGglomerative NESting) hierarchical clustering and PAM (Partitioning Around Medoids) algorithms. A description of both algorithms can be found in [56]. In case of AGNES, we used average linkage method and for both algorithms we used Manhattan distance metric.

In order to evaluate different numbers of clusters, i.e. different cuts in AGNES dendrogram and different number of PAM medoids, we utilized two metrics, *Dunn index* [36] and *average silhouette width* [92]. Higher value of either metric indicates better clustering. For each algorithm we performed clustering with different numbers of clusters, specifically we used all integers from the interval $[2, 12]$. To select the most appropriate number of clusters, we ranked the values of the two metrics for each algorithm. The higher the value of a metric, the lower the rank. Then we calculated the total rank by summing over all four ranks. For CE representation, the value of total rank decreased with larger number of clusters, but from 8 clusters onward, the change was not substantial, so we selected 8 clusters as sufficient. Specifically, the values of total rank from 2 to 12 clusters were as

follows: 60, 56, 50, 43, 46, 31, 25, 31, 24, 21, and 21. In case of CET representation, the lowest value of total rank was calculated for 8 clusters. Results for both cases are depicted in Table 3.4, in which clustering is encoded as [*sequence representation*]–[*# of clusters*]. We also included results for CE-2 and CET-2, cases with the smallest number of clusters, for comparison. These two clustering divisions are also considered in statistical tests described later. From Table 3.4 we can see that Dunn index was generally quite low, especially for PAM algorithm.

Table 3.4: Internal evaluation of CE-2, CE-8, CET-2, and CET-8 clusterings by Dunn index (DI) and average silhouette width (SIL).

	AGNES		PAM	
Clustering	DI	SIL	DI	SIL
CE-2	0.14	0.60	0.01	0.60
CE-8	0.35	0.78	0.05	0.76
CET-2	0.09	0.53	0.14	0.57
CET-8	0.16	0.64	0.02	0.61

3.2.3 Analysis of the Clusters

For each cluster we also looked for the most representative sequence. In case of PAM algorithm, it was enough to take the medoids. However, hierarchical clustering algorithms do not use medoids, so we designed and used the following procedure for AGNES algorithm. First, we computed the average value for each feature on the set of sequences from a specific cluster. We used the same features as for clustering. Then we used the same distance metric, Manhattan distance, to find a sequence most similar to the average.

Resulting representative sequences for the above mentioned clusters are shown in Table 3.5. By comparing both algorithms, we can see that they share a lot of similar representatives. Simple division of the proofs can be seen in case of CE-2 clustering for both algorithms, see the first two rows of the table. The first cluster groups proofs solved in a step-by-step fashion, where a step means an application of the resolution rule and relevant edges are added immediately after clauses (nodes) in each step. The second cluster groups proofs solved in such a way that the nodes are added first and all the edges afterwards.

Table 3.5: Cluster representatives of CE-2, CE-8, CET-2, and CET-8 clusterings.

Clustering	AGNES	PAM
CE-2	CCCEECCCCEECEECEE	CCCEECCEECCCCEEEE
	CCCCCCCCCEEEEEEEEE	CCCCCCCCCEEEEEEEEE
CE-8	CCCEECCEECCEECCEE	CCCEECCEECCEECCEE
	CCCCEECEECCCCEEEE	CCCCCEEEECCCCEEEE
	CCCCCCCEEEEEE	CCCCCCCEEEEEE
	CCCEECCEECCEE	CCCEECCEECCEE
	CCCCCCCCCEEEEEEEEE	CCCCCCCCCCCCEEEEEEEEEE
	CCCEECCCECEEE	CCCEECCEECCEECCEECCEE
	CCCCCEEEE	CCCCCEEEE
	CCCEECCEE	CCCCCCCCCEEEEEEEEE
CET-2	CTTCTCTCTCTCTCTCTEEEEEEECTEE	CTCTCTCTEECEETCTCEECTTCTEE
	CCCCCCTTTTTTCTEEEEEE	CTCTCTCTCTCTCTCTCTEEEEEEEE
CET-8	CTCTCTCTCTCTCTCTCTEEEEEEEE	CTCTCTCTCTCTCTCTCTEEEEEEEE
	CCCCCCTTTTTTCCCTTTEEEEEEEE	CCCCCTTTTTCCCTTTCTEEEEEEEE
	CCCCCCCTTTTTTTEEEEEE	CCCCCCCTTTTTTTEEEEEE
	CTCTCTCTCTEEEE	CTCTCTCTCTEEEE
	CTTCTCTEETCTCEETCTCTEECTCTEE	CTTCTCTEECTCEETCTECTE
	CTCTCTEECTCTEE	CTCTTCTCTCTCTCTEEEEEE
	CCCCTTEETTTCCCEETTEETCTCTEET	CTCTCTCTEECEETCTCEECTTCTEE
	CTCTCTEE	CCCCCCCCTTTTTTTTTTCTEEEEEEEE

On the other hand, CET-8 clustering of AGNES, for example, is slightly more difficult to analyse than CE-2. Nevertheless, several distinctive characteristics can be seen from the representatives. In the first half of sequences, all edges are added last, and in the second half, they are added approximately after each step. Similar phenomenon can be observed for the T events with respect to the C events – in some cases, most of the C events are added first, and in some cases, these events are alternating with the T events.

In addition, the first and the fifth clusters from CET-8 of AGNES contained most of the instances, precisely 696 out of 873 instances. This means that for most of the proofs, it should hold that they are not short, node and text addition is alternating, and there is no prevailing way of edge addition. The last cluster in the table, represented by the CTCTCTEE sequence, was also interesting, as its 12 out of 16 instances contained E3 error. Let us remind that E3 denotes the error of simultaneous resolving on two literals and that there were only 53 instances with E3 error in total. It means that almost 1/4 of all erroneous solutions appeared in that cluster. We can exploit the infor-

mation from the representative sequence for detecting potential error and maybe warn a student or offer them a hint even before finishing their solution.

As we want to find whether some behavioural patterns of students are connected with errors in solutions, we also analysed our sequential data with respect to solution errors. From the set of common errors, we considered the most serious error – the error of resolving on two or more literals at the same time, i.e. the E3 error. This error is the most common and also the only one with occurrence rate greater than 5%. We performed Fisher's exact test [40] to compare the occurrence of the E3 error and each of the four sequence clusters, taken for both clustering algorithms. Considering the 5% significance level, we can conclude that the data provides convincing evidence that the occurrence of E3 error is not independent of any of those four clusters of any of the two algorithms. Moreover, by analysing clusters as in the previous section, we can discover more useful patterns, such that for CET-8 there was a cluster with majority of wrong solutions. Such information may help early detect students that are at risk of making a mistake.

3.2.4 Discussion

In this section, we presented a sequence-based representation and clustering approach for logic proof solutions. We showed that by using new sequence features we are able to segment solutions according to the solving strategy and to find clusters of erroneous solutions. We believe that this method can be used even by non-expert in machine learning. There are only few parameters to be set – the minimum support for frequent subsequences and the maximum number of clusters in sequence clustering. Moreover, this method is general and it can be used also for other logic proofs, such as tableaux proofs, and for any other construction tasks that can be represented by sequences of steps.

There is a big potential of the results displayed above in practical education. By using the detection and the analysis of clusters with higher frequency of erroneous solutions a teacher can detect potential reasons of errors and find shortcomings in tutoring. Even in the process of solving the task, it is possible to detect behavioural patterns before completing the proof and warn the student.

4 DGRMiner for Mining Rule Patterns in Dynamic Graphs

Data mining of complex structures in dynamic graphs has been extensively studied in the literature as we shown in Chapter 2.2. Nevertheless, most of these methods for dynamic graphs impose various restrictions, such as the type of the dynamic graphs or the type of changes captured by the patterns. For example, there are algorithms 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 [13], LFR-Miner [60], and the algorithms presented in [55, 69, 80]. These algorithms also pose various restrictions on the form of the rule graphs. Other examples are GREW [16] and the algorithm from [106], which assume that the input dynamic graph has a fixed set of vertices and only edges are inserted or deleted over time. These requirements make them also rather restricted.

In this chapter, we present DGRMiner algorithm [103] for mining frequent patterns that are able to 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 relabelling vertices and edges.

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

In the rest of the chapter we provide necessary definitions, the description of DGRMiner algorithm, and then we present results of experiments performed on two real-world and two synthetic datasets.

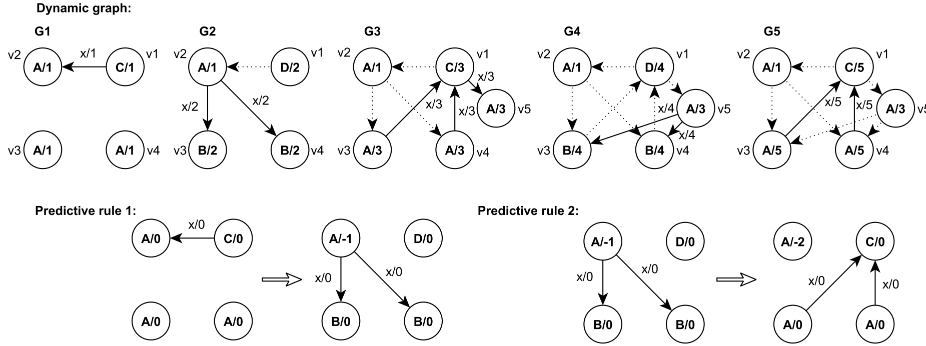


Figure 4.1: An example of a dynamic graph and two predictive graph rules. Numbers after slash symbols represent timestamps and dotted edges represent deleted edges.

4.1 Predictive Graph Rules

In this section, we provide definitions used by DGRMiner algorithm. We use Definition 2 as a definition of a dynamic graph but we add the following conditions. First, each graph snapshot is represented by a static labelled multigraph without loops and with a restriction that no two edges with the same source and target vertices can have the same label, i.e. $\forall e_1, e_2 \in E_G(f(e_1) = f(e_2) \Rightarrow l_E(e_1) \neq l_E(e_2))$. Second, no two adjacent graphs in a dynamic graph are identical as we want to capture only the changes in the dynamic graph. Third, for each $1 \leq i \leq n$, the timestamp functions $t_{G_i,V}$, $t_{G_i,E}$ assign to 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 $t_{G_i,E}(e)$. And lastly, 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. 4.1 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 the exact same information is added there.

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 a timestamp equal to $-t$ will denote a change that happened t snapshots earlier. Now we can define predictive graph rules as follows, see also an informal description below.

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

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

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

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

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 $V_{G_A} \cap V_{G_C} \neq \emptyset$

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

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 the 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 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. 4.1 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 a given dynamic graph. This notion extends subgraph isomorphism defined in Chapter 2.

Definition 4 (Occurrence of an antecedent and consequent graph)**.** *Let G be a graph used in a rule, either the antecedent or the consequent, 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 under φ, written as $G \sqsubseteq_\varphi G_i$, if there exists an injective 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. $\forall e \in E_G(f_G(e) = (u, v) \Rightarrow \exists! e' \in E_{G_i}(f_{G_i}(e') = (\varphi(u), \varphi(v)) \wedge$
$l_{G,E}(e) = l_{G_i,E}(e') \wedge t_{G,E}(e) = t_{G_i,E}(e') - i)$

That means that the mapping of vertices and edges preserves their labels and the timestamps are relative in the rule graph. This definition

is now used to define an *occurrence of a predictive graph rule*, which in turn is used to define support and confidence of a predictive graph rule in a dynamic graph and also in a set of dynamic graphs.

Definition 5 (Occurrence of a predictive graph rule). *Let $G_A \Rightarrow G_C$ be a graph rule, $DG = (G_1, ..., G_i, ..., G_n)$ be a dynamic graph, and $\varphi : V_{G_A \cup G_C} \rightarrow V_{G_i \cup G_{i+1}}$ be a function mapping vertices of both antecedent and consequent to snaphots' vertices. We say that $G_A \Rightarrow G_C$* occurs *in snapshots G_i, G_{i+1}, written as $(G_A \Rightarrow G_C) \sqsubseteq_\varphi (G_i, G_{i+1})$ if $G_A \sqsubseteq_\varphi G_i$ and $G_C \sqsubseteq_\varphi G_{i+1}$.*

Definition 6 (Support and Confidence). *Let $G_A \Rightarrow G_C$ be a predictive graph rule and $DG = (G_1, G_2, ..., G_n)$ 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:*

$$\sigma_{DG}(G_A \Rightarrow G_C) = \frac{|\{i|\exists\varphi : (G_A \Rightarrow G_C) \sqsubseteq_\varphi (G_i, G_{i+1}), 1 \leq i \leq n-1\}|}{n-1}$$

$$\sigma_{DG}(G_A) = \frac{|\{i|\exists\varphi : G_A \sqsubseteq_\varphi G_i, 1 \leq i \leq n-1\}|}{n-1}$$

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

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

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

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

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

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. 4.1 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 $\sigma \geq \sigma_{min}$ and $conf \geq conf_{min}$.

4.2 gSpan Revisited

DGRMiner employs the framework of gSpan algorithm [108]. 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 [108] is an algorithm for mining frequent patterns (subgraphs) from a set of simple undirected static graphs. Recall that *simple* means that it does not contain multiedges. It outputs frequent connected subgraphs.

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 of the form $(i, j, l_i, l_{(i,j)}, l_j)$. 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. A forward edge connects already discovered vertex i with a newly discovered vertex j. On the other hand, a backward edge connects only vertices already discovered.

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 DFS code of the corre-

sponding 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 pattern candidates 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.

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.

4.3 DGRMiner Preliminaries

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

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

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

```
1:  if s ≠ min(s) then
2:      return;
3:  𝕊 ← 𝕊 ∪ {s};
4:  enumerate s in each graph in 𝔻 and count its children;
5:  for each c, c is s′ child do
6:      if σ(c) ≥ σ_min then
7:          s ← c;
8:          Subgraph_Mining(𝔻_s,𝕊,s);
```

of the whole algorithm with a description of the remaining building blocks.

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.

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.

First, let us explain the transformation on the rules[5]. 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 a flag representing addition. For example label A will be changed to $+A$. However, timestamps of vertices and edges that were deleted or relabelled will have timestamps that are opposites of the antecedent timestamps. We know that consequent timestamps of such changes are always equal to 0 so we can easily get the original value of the antecedent back if we need to decompose the graph into a rule again. We take the opposite values because later it will help us recognize current changes simply by taking timestamps greater than or equal to 0. Vertices and edges that were deleted or relabelled 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 relabelling from A to B. As an example, transformed rules from Fig. 4.1 are shown in Fig. 4.2.

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

5. Even though the algorithm does not explicitly transform the rules, only the input dynamic graphs, it is easier to demonstrate the idea on rules.

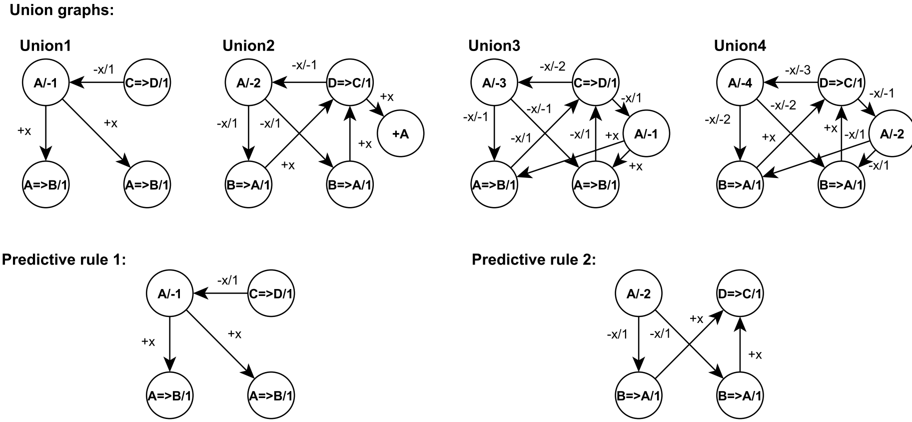


Figure 4.2: The union graph representation of the dynamic graph and the rules from Fig. 4.1.

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. 4.1 are shown in Fig. 4.2. In this case, window size is not set.

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 4.2, we showed that gSpan uses 5-tuples of the 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, and $t_{(i,j)}$ are used for the relative timestamps of vertex i, vertex j, and the edge between i and j, respectively. 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 value 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 DFS code of the Predictive Rule 1 from Fig. 4.2 is as follows: $(0, 1, A, -1, \rightarrow, +x, 0, A => B, 1), (0, 2, A, -1, \rightarrow, +x, 0, A => B, 1), (0, 3, A, -1, \leftarrow, -x, 1, C => D, 1)$.

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 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 DGRMiner Algorithm

This section provides 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 4.3. 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 the previous section, follows next. Then the algorithm removes infrequent vertices and edges but only those that represent changes as the other

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: $\quad p \leftarrow i$-th edge from $\mathbb{S}^1$
8: $\quad p.D \leftarrow$ graphs which contain p;
9: $\quad p.A \leftarrow$ graphs which contain antecedent of p;
10: $\quad \mathbb{E}_{start} \leftarrow$ first i edges from $\mathbb{S}^1$
11: $\quad$ DGR_Subgraph_Mining(p,$p.D$,$p.A$,$\mathbb{E}_{start}$);

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 DGRMiner calls recursively DGR_Subgraph_Mining procedure for each initial edge and this procedure searches for patterns growing from a given initial edge. DGR_Subgraph_Mining, 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

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,$c.D$,$s.A$,$\mathbb{E}_{start}$);

enough to check whether we cannot represent the current pattern by a smaller DFS code which starts by one of the edges in $\mathbb{E}_{start}$. 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 4.3, 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 is 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. Finally, set $c.D$ was created when the pattern s was enumerated and its children were counted.

Table 4.1: Datasets used for experiments.

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

4.5 Frequent Patterns Extracted from Dynamic Graph Datasets

Experiments with DGRMiner were conducted on four graph datasets, which we described in Section 2.3: ENRON, RESOLUTION, SYNTH, and SYNTH20. All the experiments were performed 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.

4.5.1 ENRON

The first dataset used in experiments is the email correspondence network ENRON. We discretized the timestamps to get a dynamic graph with snapshots that corresponds to days. With one extra day for vertex initialization we got 895 snapshots, as can be seen in Table 4.1.

Results on ENRON with $\sigma_{min} = 0.1$ can be found in the first row in Table 4.2. We decided to apply the time abstraction method for vertices because they were 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 dataset by deleting edges which were not updated immediately the next day. This modified dataset is named ENRON DEL in Table 4.2. 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. 4.3.

Table 4.2: 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.

Dataset	**Union vertices**	**Union edges**	σ_{min}	$conf_{min}$	**Time abstraction**		**1-vertex rules**	**All rules**	**Running time (sec)**
					Vertices	**All**			
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

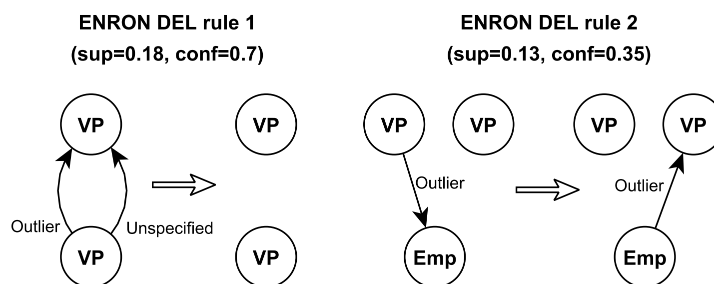


Figure 4.3: Examples of two rules from ENRON DEL. Vertex labels VP and Emp stand for Vice President and Employee, respectively.

4.5.2 Resolution Proofs in Propositional Logic

We used the set of graphs representing resolutions in propositional logic, RESOLUTION, as the second dataset. Recall that there were 19 different assignments in total and graphs from different assignments generally contain labels from different sets. In order to find frequent patterns, we restricted the dataset to only one assignment. Specifically, we took the assignment with the largest number of solutions. This set of graphs contained 103 dynamic graphs with 2911 snapshots in total, see RESOLUTION dataset in Table 4.1. The initial snapshot of each dynamic graph in an empty graph.

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 4.2 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 relabellings without any context and may not be very informative.

4.5.3 Synthetic Datasets

We also tested our method on synthetic datasets SYNTH and SYNTH 20. Each dynamic graph contained one initial snapshot and 100 snapshots with the changes, see Table 4.1.

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, see Table 4.2. This suggests that the support definition for a single dynamic graph is stricter than the one for a set of graphs.

4.6 Discussion

We presented DGRMiner algorithm for frequent predictive graph rule mining from both single dynamic graphs and sets of dynamic graphs. When compared to other existing algorithms, DGRMiner is able to capture various changes in dynamic graphs. Both directed and undirected multiedges are allowed in dynamic graphs. 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.

DGRMiner was also recently extended with a new support and confidence measures in [64]. These new types of measures allows DGRMiner to count multiple non-overlapping occurrences of patterns in a single snapshot and thus to find new patterns that would not have high enough support according to the original measures.

5 DGRMiner for Anomaly Detection and Explanation

The previous two chapters were devoted to frequent patterns in dynamic graphs. Dynamic graphs can also exhibit anomalous behaviour on various levels: from single vertices and edges through subgraphs to whole graphs. Examples include malicious network attacks, frauds in trading networks, opinion spam, and many others [7]. Anomalies, however, do not have to be necessarily negative. For example, novel patterns of behaviour in communication and interaction networks can be considered as an improvement over past patterns.

Most of the existing approaches for anomaly detection in dynamic graphs search for anomalous vertices, edges, or graph snapshots [7]. When searching for anomalies on a local level of the graph, single vertices or edges without the structural context may not provide a satisfactory explanation. Methods based on tensor decomposition [57, 83, 97] are able to find groups of anomalous vertices and edges, but it is hard to capture the evolution on a local level in detail. There are only few methods for subgraph patterns and they typically impose various restrictions on the form of the patterns. For example, the method presented in [46] assumes that the vertices are immutable. Other methods [9, 22] focus on communities, i.e. dense subgraphs, and track changes in these communities.

In this chapter, we present a different method for anomaly detection in labelled dynamic graphs that is able to capture the evolution on the subgraph level [104]. This method was built on DGRMiner algorithm, which we described in Chapter 4. Specifically, the new algorithm searches for patterns that are deviating from the frequent ones and then uses the frequent patterns as an explanation for the anomalous ones. Recall that the DGRMiner 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 relabelling vertices and edges. Moreover, DGRMiner is able to mine patterns from a single dynamic graph and also from a set of dynamic graphs.

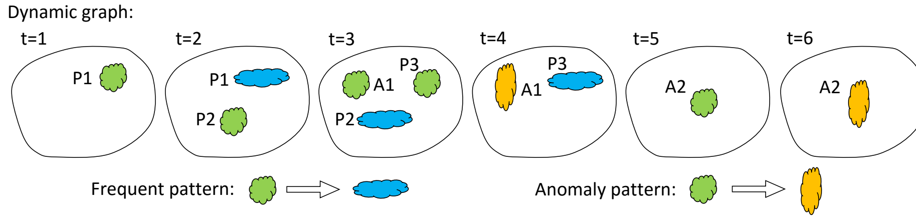


Figure 5.1: An illustration of a dynamic graph with six snapshots, a frequent pattern with occurrences P1, P2, P3, and an anomaly pattern with occurrences A1, A2.

Fig. 5.1 illustrates the idea of anomaly detection by exploiting the frequent predictive rules. Assume that we have found a frequent pattern depicting a transformation of the *green* subgraph into the *blue* one. This pattern has three occurrences in the input dynamic graph: P1, P2 and P3. However, not all occurrences of the *green* subgraph, i.e. the antecedent, are transformed into the *blue* subgraph. There are situations in which it is transformed into the *orange* subgraph: A1 and A2. We mark such a transformation as an anomaly pattern and use the original frequent one as an explanation of it.

The remainder of this chapter is organised as follows. First, we state the problem of anomaly detection. Then we describe the method for single-vertex and multi-vertex anomalies. At the end, we present examples of anomalies found in real-world dynamic graphs.

5.1 Anomaly Pattern Mining

This section describes the extension of DGRMiner for anomaly detection and explanation. It is assumed that the reader is already familiar with the DGRMiner basics described in Chapter 4. A pseudocode of modified DGRMiner is given in Algorithm 5. Lines with ▷ *anomaly* comment mark the extension for anomaly detection and they are described in this section.

Having found the frequent patterns, we are interested in infrequent patterns deviating from the frequent ones, i.e. patterns with the same antecedent but a different consequent. The frequent patterns are then used as an explanation of the anomaly patterns. For each frequent pat-

Algorithm 5 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: output frequent *change* vertices with high enough confidence;
4: find simple anomaly patterns ▷ *anomaly*
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 6 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 $\mathbb{A}$, and enumerate anomaly patterns from antecedent occurrences; ▷ *anomaly*
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:     **for each** anomaly pattern a **do** ▷ *anomaly*
11:         **if** score(a)$\geq$ min_score **then** ▷ *anomaly*
12:             output a ▷ *anomaly*
13: sort remaining children in DFS lexicographic order;
14: **for each** child c **do**
15:     DGR_Subgraph_Mining(c,$c.D$,$s.A$,$\mathbb{E}_{start}$);

tern being processed, we store its occurrences in the dynamic graph. Specifically, we store sets of occupied vertex and edge IDs for each

snapshot. When searching for anomaly patterns, we do not use vertices and edges of these occurrences. This means that occurrences of anomaly patterns and the explanatory frequent patterns are completely disjoint. This ensures that the anomaly patterns are independent of the frequent ones.

In order to decide which deviating patterns are truly anomalies, we use *outlierness* score defined as the opposite value of the confidence, i.e. $out = 1 - conf$. The idea is that the lower the confidence is, the more deviating the pattern is. Given a minimum outlierness score out_{min}, we output patterns for which $out \geq out_{min}$. It is necessary to check the outlierness of all potential anomaly patterns because not all *complementary* patterns of the frequent ones have low enough confidence. For example, the frequent pattern in Fig. 5.1 has $conf = 3/4$, but the anomaly pattern has $conf = 2/4$. Such an anomaly pattern would not be output in the case of $out_{min} = 3/4$.

The confidence of the pattern is computed from its support. We describe how to discover the single-vertex anomalies and how to compute their support in the following subsection. After this simple scenario, we focus on more complex anomaly patterns, whose discovery is a more involved process.

5.1.1 Single-vertex Anomalies

DGRMiner first looks for single-vertex anomaly patterns that are *complementary* to the frequent ones. This is computed at line 4 of Algorithm 5. Single-vertex frequent patterns take one of the following forms: $-A$, $A \Rightarrow B$, $+B$. The antecedent of $-A$ is A, which is also the antecedent of patterns A (no change) and $A \Rightarrow C$ for some C. Thus, anomaly patterns, complementary to this frequent pattern, are of the form A or $A \Rightarrow C$. As for the frequent pattern $A \Rightarrow B$, the possible anomaly patterns can be A (no change), $-A$, and $A \Rightarrow C$ for some $C \neq B$. It is trivial to enumerate such patterns in the input dynamic graph and compute their support because our union-graph representation allows us to obtain the antecedent labels of the vertices.

A different approach is required for frequent patterns of the $+B$ form. The antecedent of these patterns is an empty graph, whose support is the number of snapshots. There is only one anomaly pattern complementary to $+B$ and we mark it by $!B$. The meaning of $!B$ is that

a vertex with label B should have been added. We use it because we need to explicitly express that such an addition did not happen. Thus, support of $!B$ is computed from graph transitions where $+B$ did not occur.

5.1.2 Enumeration of Anomaly Patterns in General

Enumeration of anomaly patterns with regard to larger frequent patterns follows the enumeration of antecedents, as is indicated at line 5 of Algorithm 6. Each such frequent pattern can contain multiple changes. First, suppose there are some vertices and/or edges that do not represent *addition* changes. Such elements are included in the antecedent and their occurrences can be located in the input dynamic graph. As in the case of single-vertex patterns, we can then simply extract unambiguously the consequent part for each such occurrence.

The situation gets more complicated when there are also elements representing *addition* changes in the frequent pattern. Again, each such element is either found in the dynamic graph or we explicitly say that it is missing. In order to capture the parts that are different from the frequent pattern, we search for all maximal common subgraphs of the frequent pattern and the dynamic graph, given the fact that the antecedent part is already mapped to the dynamic graph. Let us illustrate this process on an example depicted in Fig. 5.2. Suppose that we are searching for anomalies with regard to the frequent pattern shown in the figure and we have already found the antecedent in a union graph, also shown in the figure. The antecedent consists of two vertices with label A. It is clear that the frequent pattern does not occur in this union graph. By taking the maximal common subgraphs of those two graphs, we get three anomaly patterns that differ in edge that is missing. The missing edges are depicted by dashed lines with label $!E$.

If we find an occurrence of an anomaly pattern, we use that union graph for support computation. As the frequent patterns and anomaly patterns have to be completely disjoint, there is only one scenario that has to be treated in a different way. If we are searching for anomalies with regard to a frequent pattern whose all elements denote *addition* changes, i.e. the antecedent is an empty graph, care must be taken when the support is computed. Specifically, if a union graph con-

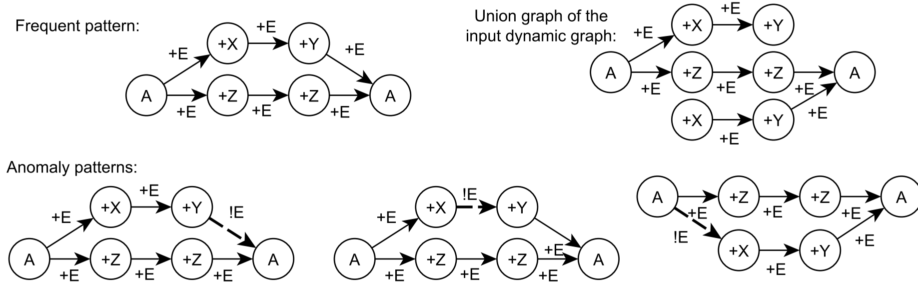


Figure 5.2: An example of anomaly pattern enumeration.

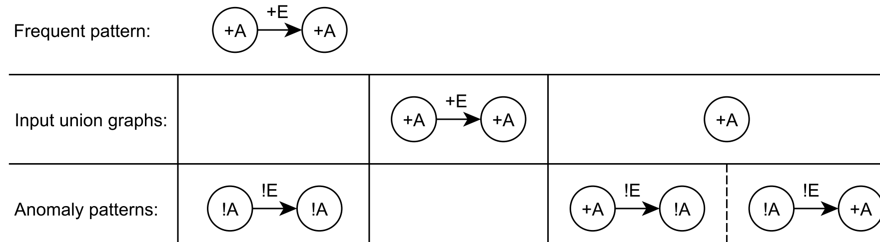


Figure 5.3: An example of anomaly pattern enumeration with regard to a frequent pattern with an empty antecedent.

tains the whole *addition* frequent pattern but not its part, then this union graph cannot contain the anomaly pattern with only explicit *non-additions*. An example with such a scenario is depicted in Fig. 5.3. The first input union graph is an empty graph and the only anomaly pattern with regard to the frequent pattern is the one with only explicit *non-additions*. The second input union graph contains an occurrence of the frequent pattern and it does not contain any part of the pattern besides. In this case, the only possible anomaly pattern could be with only explicit *non-additions*, but this is the special case described earlier and we do not allow such a pattern in such situations. The third input union graph contains a single vertex which is not a part of a frequent pattern occurrence and it can be used as a part of two different anomaly patterns.

5.2 Anomaly Detection Experiments

In this section we present results of experiments on two real-world datasets. Unfortunately, there are not many attributed dynamic graph datasets available that could be used for the experiments. The experiments were conducted by a C++ implementation on a PC equipped with CPU Intel i5-4570, 3.2GHz, 16GB of main memory, and running 64-bit version of Windows 10. For all experiments, we set $conf_{min} = 0.6$ and $out_{mint} = 0.8$.

5.2.1 ENRON

ENRON email correspondence was again used as one of the datasets for our experiments. As in the previous chapter, the dynamic graph had 895 snapshots. However, due to more interesting results in the previous chapter, we implicitly marked all disappeared edges as deleted for these experiments, i.e. as in ENRON DEL dataset in the previous chapter. We also prepared a different dataset, ENRON UNI, where employee IDs were used as the vertex labels. This ensures that all patterns apply to specific employees and not to arbitrary employees of the given ranks.

As for the ENRON dataset, we set minimum support to 0.1 and performed the time abstraction method on vertices. The time abstraction of vertices allows us to ignore time connected to vertices. We have found 44 anomaly patterns, see Table 5.1. Two anomaly patterns are depicted in Fig. 5.4a. In the first example, we can see the following scenario. If an ordinary employee (Emp) sends an *outlier* email to a vice president (VP) and this vice president sends an *outlier* email to another VP, then the employee typically sends such an email to the first VP again the next day. This is captured by the explanation frequent pattern. However, it may occasionally happen that the VP sends the email instead of the employee as is depicted by the anomaly pattern. In the second example, we can see that if a VP sends such three emails, they do not send the emails again the next day. If they do, it is a rare case.

In the case of the second dataset, ENRON UNI, we set minimum support to only 0.02 because patterns connected to specific groups of people are less frequent. 346 anomaly patterns were found and two examples are shown in Fig. 5.4b. The difference from the previous

Table 5.1: Results of experiments. Running time is averaged over five runs, $conf_{min} = 0.6$ and $out_{min} = 0.8$.

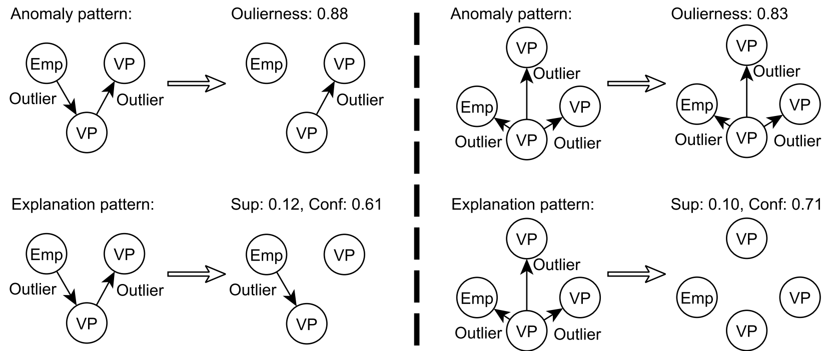
Dataset	σ_{min}	Time abstraction		Anomaly patterns	Running time (sec)
		Vertices	All		
ENRON	0.10	✓	×	44	290.9
ENRON UNI	0.02	✓	×	346	248.9
RESOLUTION	0.05	×	×	76	0.4
RESOLUTION	0.05	✓	✓	198	4.0

cases is that the patterns are related to specific people. For example, the first anomaly pattern applies exactly to people with IDs 58, 63, and 146. Such patterns can be used to discover anomalous communication patterns which may represent changes in the current state. Similarly, anomalous communication patterns in computer networks may represent a security threat.

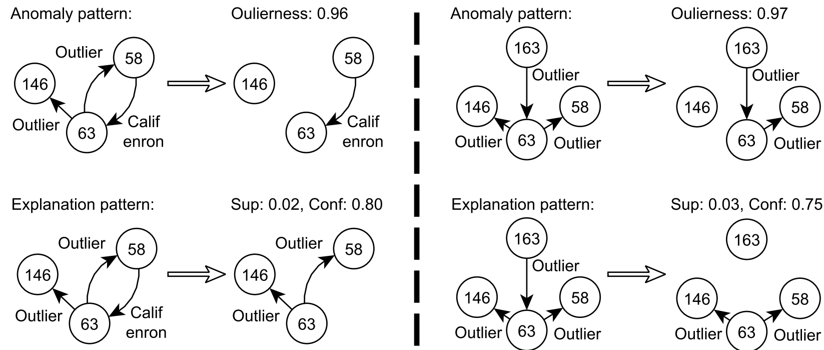
5.2.2 Resolution Proofs in Propositional Logic

We also used RESOLUTION dataset, i.e. a set of graphs representing resolution proofs in propositional logic. The dataset was again restricted to the assignment with the largest number of solutions so there were 103 dynamic graphs with 2911 snapshots in total.

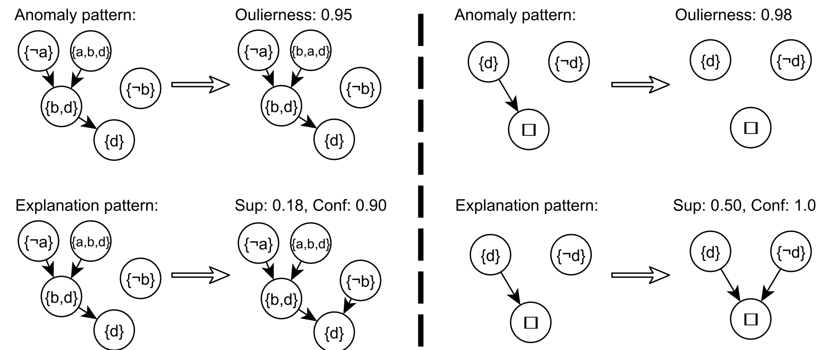
For our experiments on this dataset, we set minimum support to 0.05. First, we did not use time abstraction at all and left the timestamps of vertices and edges as they were. This setting yielded 76 anomaly patterns, out of which 75 were single-vertex patterns. The remaining one described a situation in which an edge should had been added but it was not. Therefore, we repeated the experiment with time abstraction performed both on vertices and edges and got 198 anomaly patterns. Two examples are shown in Fig. 5.4c. The first one captures the case where students did not continue with an addition of an edge between $\{\neg b\}$ and $\{d\}$ but replaced label $\{a, b, d\}$ with $\{b, a, d\}$, which is completely unnecessary. The second example shows an odd situation in which students did not add the final edge pointing to the empty clause (depicted by a square) and they deleted the existing edge instead. Although the frequent pattern in this second example has



(a) Examples of two anomaly patterns in ENRON dataset.



(b) Examples of two anomaly patterns in ENRON UNI dataset.



(c) Examples of two anomaly patterns in RESOLUTION dataset.

Figure 5.4: Examples of anomaly patterns from experiments.

$conf = 1.0$, we discovered this anomaly pattern because both of them can be found in certain solutions.

5.3 Discussion

In this chapter, we presented an extension of DGRMiner algorithm for anomaly detection and explanation in dynamic graphs. The method is able to capture anomaly patterns on the subgraph level in the form of predictive rules. These rules are able to capture various changes, such as addition and deletion of vertices and edges, and relabelling of vertices and edges.

For each anomaly pattern, one or more frequent patterns are output as an explanation. This is because the computation of DGRMiner is driven by frequent pattern mining and anomaly patterns are mined with regard to these frequent patterns. Since multiple frequent patterns can share the same antecedent, it is possible to discover one anomaly pattern several times. Such an anomaly pattern thus can be explained by multiple frequent patterns.

Even though the enumeration of a pattern takes exponential running time in the worst case, DGRMiner can be efficient on real-world dynamic graphs as is shown in Table 5.1. This is mainly caused by diverse vertex and edge labels that significantly reduce the search space when subgraphs are being enumerated. On the contrary, small dense subgraphs with homogeneous labels would require the exponential running time for subgraph enumeration.

6 WalDis for Discriminative Pattern Mining

We have seen in previous chapters that dynamic graphs change in various ways and these changes may be of high importance in many scenarios. There can be additions and deletions of vertices and edges. Moreover, labels or attributes of vertices and edges can change. We have already presented several algorithms for frequent pattern mining. These patterns were made of frequently occurring subgraphs. Now we take a step aside and assess the context of patterns, i.e. what are the scenarios in which particular events happen. More specifically, given two different sets of graph events, we are interested in finding discriminative patterns that appear frequently in the local neighbourhood of events of one set but not the other.

For example, let us consider a phone call network between customers of a telecommunication company. A change of a vertex attribute may represent a change of a rate plan, or worse, a change of the operator. Naturally, such a company is interested in predicting these kinds of events, and possibly in being able to explain them. An example of such a graph is shown in Fig. 6.1. In this example, the edge attributes are denoted by l and they represent normalized durations of the phone calls. Edge timestamps are denoted by t. Furthermore, let us assume that there is a set of positive events representing changes from rate plan B to a more expensive plan A and a set of negative events representing changes to a cheaper plan C. In our example, positive events occur at vertices v_1 and v_2, and negative events at v_3 and v_4. The goal is to find graph patterns occurring in the neighbourhood of the positive events and not in the neighbourhood of the negative events. An example of such a pattern is depicted in the same picture on the right. In order to distinguish the pattern from the input graph, we used pv and pe for IDs of vertices and edges, respectively. This pattern occurs in the neighbourhood of the vertices v_1 and v_2 and matches with edges e_1, e_2, e_3 in case of v_1, and e_4 and e_5 in case of v_2. This is illustrated by dashed edges. As an inexact graph matching is used, the pattern is actually represented by a union graph taken over several similar graphs. This is illustrated by lists of attributes and timestamps at each pattern edge. Furthermore, the timestamps of the pattern edges are depicted as values relative to the timestamps

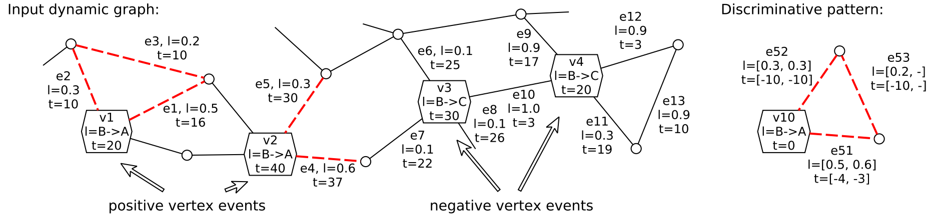


Figure 6.1: An example of a dynamic graph and a discriminative pattern with respect to positive and negative events. Parameter l denotes labels or attributes, and parameter t timestamps.

of the corresponding events in order to highlight their mutual similarity. More specifically, the event at vertex pv_1 has a relative timestamp $t = 0$ and all other timestamps in the pattern are relative to this event. Inexact matching also means that not all edges have to be present in the neighbourhood of the events. This is the case of vertex v_2 and its neighbourhood in our example.

Chapter 2 presented several algorithms for discriminative pattern mining in both static and dynamic graphs. Existing approaches [41, 54, 109, 122] typically focus on patterns that distinguish whole graphs and not local events, i.e. the types of events we described above. Moreover, pattern mining methods usually assume discretized timestamps and the patterns have to match exactly in the input graphs [55, 69, 106].

We present WalDis algorithm [105] for mining discriminative patterns on a local level of dynamic attributed graphs in this chapter. It uses a random walk-based approach to mine patterns matching inexactly from the perspective of attributes and also timestamps. This also means that it does not require any discretization of timestamps to be able to find patterns. Moreover, by utilizing sampling techniques, the algorithm does not have to traverse the whole search space.

6.1 Preliminaries

Before describing WalDis algorithm for mining discriminative patterns, we state necessary definitions. WalDis uses a simpler representation of a dynamic graph than DGRMiner does. We already mentioned

such a representation, consisting only of a single static graph extended by the timestamp functions, in Section 2.2. In order to distinguish this representation from the dynamic graph defined earlier, we will call such a graph a *temporal multigraph* in this chapter.

A *temporal multigraph* is a 6-tuple $G = (V_G, E_G, f_G, l_{G,V}, l_{G,E}, t_G)$, 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 assigning attribute vectors to vertices and edges, respectively, and $t_G : E_G \to \mathbb{R}$ is a function assigning timestamps represented as real numbers to edges. Unless stated otherwise, the term *graph* will denote a temporal multigraph in the following text. Recall the definitions of subgraph isomorphism and label-preserving isomorphism from Section 2.1. In this chapter, the label-preserving isomorphism preserves timestamps as well, i.e. $t_G(e) = t_{G'}(\psi(e))$ for all $e \in E_G$, and we use sets of attributes instead of simple labels. Furthermore, in order to distinguish different functions, we will use subscripts. For example, $\varphi_{G,H}$ will denote function $\varphi : G \to H$.

A *vertex event* in a graph G is a pair (v, t), where $v \in V_G$ and $t \in \mathbb{R}$. An *edge event* in a graph G is a pair (e, t), where $e \in E_G$ and $t \in \mathbb{R}$. For the sake of brevity, the following definitions assume vertex events but they can be analogically stated for edge events. Let P and N be two disjoint sets of vertex events. P denotes a set of *positive* events and N a set of *negative* events. For example, $P = \{(v_1, 20), (v_2, 40)\}$ and $N = \{(v_3, 30), (v_4, 20)\}$ in Fig. 6.1. As the local neighbourhoods of the events may be seen as independent graphs, we call these neighbourhoods *instances*[1].

Let $\mathcal{G} = \{G_1, G_2, ..., G_n\}$ be a set of graphs. A *union graph* of $\mathcal{G}$ is a graph $\mathcal{G}_\cup$ whose vertices and edges correspond to a union of vertices and edges of $\mathcal{G}$, respectively.

A *positive pattern* with respect to a non-empty vertex event set $P = \{(v_1, t_1), (v_2, t_2), ..., (v_n, t_n)\}$ and a graph G is a non-empty set of graphs $\mathcal{G}_P = \{G_1, G_2, ..., G_n\}$ such that its union graph $\mathcal{G}_{P\cup}$ is connected, and for all $1 \leq i, j \leq n$ the following conditions hold: $v_i \in V_{G_i}$ and $G_i \sqsubseteq^* G$ and $\varphi_{G_i,\mathcal{G}_{P\cup}}(v_i) = \varphi_{G_j,\mathcal{G}_{P\cup}}(v_j)$ and $\forall e \in E_{G_i} : t_{G_i}(e) \leq t_i$. In other words, there is a graph for each vertex event that contains the vertex and all event vertices are mapped to the same vertex

1. Technically speaking, each instance is a copy of the input graph.

in the union graph. Timestamps of pattern graph edges are also less than or equal to the timestamp of the corresponding event. For example, the discriminative pattern in Fig. 6.1 is presented by a union graph with edges pe_1, pe_2, and pe_3. The mappings are $\varphi(v_1) = \varphi(v_2) = pv_1$, $\psi(e_1) = \psi(e_4) = pe_1$, etc.

Note that the pattern graphs do not have to be disjunctive, i.e. they may overlap in the original input graph. To deal with this in an implementation, we also use the IDs of the pattern graphs, i.e. instances, in order to fully distinguish the edges.

In order to define discriminative patterns, it is necessary to define a relationship between the pattern and the negative event set N. For a graph G, a non-empty negative event set $N = \{(v_{n+1}, t_{n+1}), (v_{n+2}, t_{n+2}), ..., (v_{n+m}, t_{n+m})\}$, and a pattern $\mathcal{G}_P$, we define *negative pattern* as $\mathcal{G}_N = \{G_{n+1}, G_{n+2}, ..., G_{n+m}\}$, where $v_i \in V_{G_i}$ and $G_i \sqsubseteq^* G$ and $G_i \subseteq \mathcal{G}_{P\cup}$ and $\varphi_{G_1,\mathcal{G}_{P\cup}}(v_1) = \varphi_{G_i,\mathcal{G}_{P\cup}}(v_i)$ and $\forall e \in E_{G_i} : t_{G_i}(e) \leq t_i$ for all $n+1 \leq i \leq n+m$. Thus, a negative pattern is always linked to a positive pattern and all its graphs must be subgraph isomorphic to the union graph of the positive pattern. Moreover, the event vertices must be mapped to the same union graph vertex.

The quality of a pattern is assessed by a *fitness* function F. Before defining F, we need to define several other functions. First, we employ Kronecker delta function in the calculations. To simplify notation, we put $\delta_{e_i,e_j} = 1$ if $\psi_{G_i,\mathcal{G}_{P\cup}}(e_i) = \psi_{G_j,\mathcal{G}_{P\cup}}(e_j)$ and 0 otherwise. Thus, this function checks whether two edges are mapped to the same edge in the union graph of the pattern. In Fig. 6.1, $\delta_{e_1,e_4} = 1$, $\delta_{e_1,e_5} = 0$.

Next, we need to define function $r(e, t_k)$ that influences the likelihood of selecting edge e in a random-walk starting from the vertex of event (v_k, t_k). It is computed as $r(e, t_k) = e^{-x/\sigma_1}$ for $x \geq 0$ and 0 otherwise, where σ_1 is a *time unit* parameter and $x = t_k - t_G(e)$. Thus, $r(e, t_k) = 1$ for edges having the same timestamp as the related event. For the older edges, the similarity drops exponentially. For the newer edges, the probability is zero.

Another function that is needed for the fitness is the *similarity* function of two edges from instances i and j and we define it as follows:

$$s(e_i, e_j) = \frac{2 * att_sim(e_i, e_j) * time_sim(e_i, e_j)}{att_sim(e_i, e_j) + time_sim(e_i, e_j)} \tag{6.1}$$

Function $s(e_j, e_i)$ expresses the similarity of selecting edges e_j and e_i together, i.e. mapping them to the same pattern edge. It is defined as the harmonic average of *attribute similarity* and *timestamp similarity* of those edges. Harmonic mean is used because its value is high only if both components are high enough. Attribute similarity *att_sim* is computed from the mixed Euclidean distance measure on attribute vectors, in which the difference between categorical values is 0 or 1 depending on whether they are equal or not. Attribute similarity is also bounded from below by value 0.1 so that different edges may be selected together by inexact matching. Time similarity is defined as $time_sim(e_i, e_j) = e^{-y/\sigma_2}$, where σ_2 is another time unit parameter and $y = |(t_i - t_G(e_i)) - (t_j - t_G(e_j))|$. Thus, $time_sim(e_i, e_j) = 1$ for edges having the same time distance from the corresponding events and this similarity exponentially drops for edges having different time distance from the events. It is assumed that attribute values are normalized so that *att_sim* and *time_sim* are on the same scale. From the definitions, we can see that $s(e_j, e_i) = s(e_i, e_j)$. Continuing our example from Fig. 6.1, let us set $\sigma_1 = 5$ and $\sigma_2 = 2$. Then $r(e_1, t_1) = e^{-(20-16)/5} \approx 0.45$, $r(e_4, t_2) = e^{-(40-37)/5} \approx 0.55$, $att_sim(e_1, e_4) = 0.9$, $time_sim(e_1, e_4) = e^{-|4-3|/2} \approx 0.61$, $s(e_1, e_4) = s(e_4, e_1) \approx 0.73$.

Finally, let us define *fitness* F by using two functions F^+ and F^-:

$$F^+(\mathcal{G}_P) = \frac{1}{\frac{1}{2}n(n-1)|E_{\mathcal{G}_{P\cup}}|} \sum_{1 \le i < j \le n} \sum_{e_i \in E_i, e_j \in E_j} \delta_{e_i, e_j} s(e_i, e_j) \tag{6.2}$$

$$F^-(\mathcal{G}_P, \mathcal{G}_N) = \frac{1}{nm|E_{\mathcal{G}_{P\cup}}|} \sum_{1 \le i \le n < j \le n+m} \sum_{e_i \in E_i, e_j \in E_j} \delta_{e_i, e_j} s(e_i, e_j) \tag{6.3}$$

$$F(\mathcal{G}_P) = F^+(\mathcal{G}_P) - \max_{\mathcal{G}_N} F^-(\mathcal{G}_P, \mathcal{G}_N) \tag{6.4}$$

Function F^+ sums the similarities of all pairs of edges from positive patterns that are mapped to the same edge in the union graph. Function F^- is different in the fact that one edge of the pair must be from a positive pattern and the second one from a negative pattern. The higher the value of F^+, the more similar the graphs of $\mathcal{G}_P$ are. The higher the value of F^-, the more similar the negative pattern

graphs to positive pattern graphs are. Both of these functions are normalized by the number of possible edge pairs. The overall *fitness* F of pattern $\mathcal{G}_P$ is given by F^+ and penalized by F^-. For a given pattern $\mathcal{G}_P$, F selects $\mathcal{G}_N$ that maximizes the negative fitness F^-. Thus, a high value of F ensures that the pattern does not occur in the neighbourhood of the negative events. We say that a positive pattern $\mathcal{G}_P$ is a *discriminative pattern* if $F(\mathcal{G}_P) > 0$.

6.2 WalDis Algorithm

This section describes WalDis algorithm used for mining discriminative patterns. The algorithm assumes a set of positive and a set of negative events on the input. It consists of two main phases as stated in Algorithm 7. In the first step, the algorithm explores the local neighbourhoods of the events by using a random walk technique and computes similarities of edges from these neighbourhoods. These random walks serve as a sampling method so that the algorithm does not have to compute similarities for all edge pairs. In the second step, it utilizes the computed similarities to extract a discriminative pattern by a greedy method. By running the algorithm repeatedly on different samples of positive and negative events, it is possible to find multiple patterns.

The first phase utilizes random walks in order to compute the edge similarities. The walks run from all events in a parallel fashion, although not entirely independently. Briefly, each step of a random walk is performed in one instance first and then in all other instances according to that first step. These dependent steps go through edges that are similar to the edge walked first. Thus, the algorithm aims to walk similar edges simultaneously. Moreover, each edge can be traversed by one walk at most once in a given instance. A pseudocode of this process is given in Algorithm 8.

The second phase of the algorithm uses similarities computed in the previous phase in order to extract a pattern from the temporal graph. The idea is to create a pattern from edges that appear in positive instances and not in the negative instances. The edges are selected by a greedy method stated as Algorithm 9.

The following two subsections describe both phases in detail.

Algorithm 7 WalDis(G, P, N, size_of_pattern_graph)

```
1: similarities ← compute_similarities(G, P, N)
2: positive_instances ← prepare_positive_instances(G, P)
3: negative_instances ← prepare_negative_instances(G, N)
4: pattern ← construct_pattern_greedy(G, similarities,
                                      positive_instances,
                                      negative_instances,
                                      size_of_pattern_graph)
5: return pattern
```

6.2.1 Computing Statistics by Random Walks

The first step of WalDis computes similarities according to Algorithm 8. For a given number of random walks, see line 1, it performs the following steps to obtain the similarities.

At the beginning, see line 2 and 3, it initializes *live_instances* set with indices of used instances and randomly[2] selects event vertices that are used as starting points for random walks. This is necessary for edge events as there are two vertices that can be used as starting points. The algorithm proceeds with the while loop on line 4, in which one step of a walk is performed in each instance. To assign a higher score to edges closer to events, the algorithm uses random-walk restarts with default probability 0.1, i.e. there is a 10% chance at each step that the algorithm will start a new random walk from the events. This is controlled by *should_continue* function, which also checks that there are at least two positive instances in *live_instances*.

Each step across all instances is driven by an edge e_1, called *primary edge*, which is selected from a randomly chosen positive instance on lines 5 and 6. This choice is limited to positive instances because we want edges that are in the positive instances and possibly not in the negative ones. More precisely, a live positive instance is randomly selected first and then the algorithm selects an edge adjacent to the instance's current vertex by using function $r(e,t)$, i.e. the probabilities of edge selection are proportional to values of exponential function $r(e,t)$. Time unit parameter σ_1 in $r(e,t)$ helps the algorithm

2. If not specified, uniform distribution is used for random selection.

Algorithm 8 compute_similarities(G, P, N)

```
1:  for i ← 1 to number_of_random_walks do
2:      live_instances ← {1, 2, ..., n + m} ;
3:      current_vertices ← randomly select starting vertices from P
                                        and N;
4:      while should_continue(live_instances, restart_prob=0.1) do
5:          i1 ← randomly select one element from live_instances,
                                        but only from values 1..n;
6:          e1 ← select_prim_edge(G, current_vertices[i1])
7:          if e1 is NULL then
8:              break;
9:          for i2 in live_instances \{i1} do
10:             e2 ← select_sec_edge(G, e1, current_vertices[i2])
11:             if e2 is NULL then
12:                 live_instances ← live_instances \{i2};
13:             else
14:                 s_{e1,e2} ← s_{e1,e2} + s(e1, e2);
            update the current_vertices in live_instances;
15: return all s_{ei,ej} values
```

handle edges appearing in minutes, days, etc. The exponential function was chosen as it decreases rapidly for older edges but does not equal to zero.

Furthermore, the algorithm can choose to abort the selection process with a small probability. This allows the algorithm to avoid the selection if there are only edges having small probabilities to be chosen.

The algorithm continues by selecting a *secondary edge* e_2 in each of the remaining instances, see lines 9 and 10. The probability of selecting edge candidate e_k as the secondary edge is proportional to similarity $s(e_1, e_k)$ according to Eq. 6.1. Time unit parameter σ_2 in $s(e_1, e_k)$ also serves as a multiplication factor that helps us deal with different time units on edges. The algorithm again adds a small probability to abort the selection so it can avoid edges dissimilar from the primary one.

If a secondary edge was selected successfully, we add the similarity of the couple, represented by $\{(i_1, e_1), (i_2, e_2)\}$, to the aggregated statis-

tics on line 14. We sum the similarities so that edges closer to events have higher chance of being selected into the pattern. Instance IDs are necessary as one edge can appear in several overlapping instances. It is enough to store only s_{e_1,e_2} as s_{e_2,e_1} is identical.

Random walks technique described above allows us to explore only a smaller set of edge combinations, i.e. combinations of edges that are rather similar to each other. If we were to consider all combinations of edges by using an exhaustive search, we would have to deal with an exponential number of combinations.

6.2.2 Pattern Construction by a Greedy Approach

The second phase of the algorithm uses the aggregated similarities computed in the first phase in order to extract a pattern from the temporal graph. The idea is to create a pattern from edges that appear in positive instances and not in the negative instances. Each pattern edge is represented by a set of original edges, but there is at most one edge from each positive instance for such a pattern edge. Edges that are similar to each other and were walked simultaneously in the first phase are put together into a set by the algorithm. This process, stated as Algorithm 9, is performed in a greedy fashion as follows.

Until the desired number of pattern edges is extracted, see line 3, repeat the following procedure for finding one pattern edge. First, extract a pair of edges from positive instances with the highest score, i.e. aggregated similarity, such that none of these two edges has already been chosen and assign this score to both edges. This happens on line 4 of the algorithm. Now, by running lines 5 and 6, select the next edge in such a way that it has the highest score with one of the already selected edges from *pattern_edge* set and add this score to this newly added edge. The restriction is that this edge has not been selected in the whole pattern construction process yet and also this instance has not been used for this pattern edge. This continues until we occupy all instances or run out of usable edges.

The second part of the for loop, starting on line 7, searches for edges in negative instances. As the main goal of WalDis is to find the patterns occurring in the positive instances and not in the negative ones, we have to ensure that there are no edges in the negative instances similar to the ones taken from the positive instances. Therefore, the idea is

Algorithm 9 construct_pattern_greedy(G, similarities, positive_instances, negative_instances, size_of_pattern_graph)

```
1:  pattern_edges ← ∅
2:  neg_pattern_edges ← ∅
3:  for i ← 1 to size_of_pattern_graph do
4:      pattern_edge ← select_nth_best_pair_of_edges(G,
                    similarities, positive_instances,
                    pattern_edge);
5:      while |pattern_edge| < |positive_instances| do
6:          pattern_edge ← pattern_edge ∪
                    select_best_edge(G, similarities,
                    positive_instances, pattern_edge);
7:      neg_pattern_edge ← ∅
8:      while |neg_pattern_edge| < |negative_instances| do
9:          neg_pattern_edge ← neg_pattern_edge ∪
                      select_best_edge(G, similarities,
                      negative_instances,
                      pattern_edge);
10:     pattern_edges ← pattern_edges ∪ pattern_edge
11:     neg_pattern_edges ← neg_pattern_edges ∪
                              neg_pattern_edge
12: pattern_edges ← assess_pattern_edges(pattern_edges,
                              neg_pattern_edges)
13: pattern_edges ← reduce_pattern_edges(pattern_edges)
14: return pattern_edges
```

to find the most similar edges from negative instances, and if their similarities are low, we have an evidence that the selected edges from positive instances are appropriate. Practically, the pattern edge is assessed by the sum of the scores from positive instances from which the sum of the scores from negative instances is subtracted. The algorithm again selects at most one edge from each negative instance by running lines 8 and 9, and it selects the highest-scoring edges, again, with respect to the already selected positive edges. That means that it does not consider already selected negative edges.

At the end of the for loop, the pattern edge is tied with a set of edges from positive instances and a set of edges from negative instances. This process repeats until the desired number of pattern edges is created or until the algorithm runs out of usable edges. Then, *assess_pattern_edges* function is used on line 12 to assess each pattern edge. For each pattern edge, it subtracts the average negative score from each positive score. If we compare the above description to definitions of F^+ and F^- used for definition of F, see Eq. 6.2, 6.3, and 6.4, we can see that our computation is simpler. The algorithm greedily selects only a subset of similarities into the final score in order not to perform quadratic number of summations.

In the above description of the algorithm, we stated that the algorithm may not find an edge in each positive instance for each pattern edge. If we represent the selected edges as a matrix with one row for each pattern edge and one column for each instance, there may be some empty cells in such a matrix. In order to get a compact pattern, i.e all pattern edges cover the same set of instances, we remove some rows or columns so that there are no empty cells any more. For this problem, we use a greedy algorithm for weighted set cover problem [118] to remove rows and columns in such a way that the final pattern score is as high as possible. After this reduction on line 13, all instances that remain *embedded* in the pattern contain the same edges, deviating only in the attribute values and timestamps. The reduced pattern is then returned as a result.

6.3 Patterns Extracted by WalDis

WalDis was experimentally evaluated by a Python implementation on three real-world graph datasets from Chapter 2.3: DBLP, TELCO, and ENRON. For each dataset, we experimented with a set of 5 different parameter settings. The parameter settings and the results can be found in Table 6.1. *Set size* denotes the number of positive and negative events. Probability of edge non-selection was set to 0.05 in all experiments. For each setting, we executed 10 runs on randomly selected sets of events and the table shows average results over those 10 runs. Each experiment run was performed in the following way. A set of N positive and a set of N negative events were chosen as a train-

ing dataset, from which we extracted a discriminative pattern. Then different N positive and N negative events were chosen as a test set on which we assessed the extracted pattern.

The assessment on a test dataset proceeds as follows for a given pattern and a given test event. Select one instance that is embedded in the pattern. Perform several random *walks* (10 in our experiments) without restarts on this pattern instance while walking simultaneously in the tested instance. We allow to continue from any vertex that was already visited and thus the result is technically not a walk, but a subgraph. When compared to the first phase of WalDis, the pattern graph now acts as the primary instance and the tested instance as the secondary one. The score of this pattern instance is given by the sum of similarities of the simultaneously-walked edges divided by the number of all random-walk steps. If the random walk cannot continue in the tested instance, it continues only in the pattern instance and keeps counting the walks without similarities. This enables the algorithm to lower the score for cases when there are no edges in the tested instance that could be matched with the pattern edges. Pattern instance gets then the highest score across all random walks. Such a score is computed for each pattern instance and the average is returned as the final matching score. By averaging the pattern scores on the set of positive events, we get a final *positive score*. In the same way, we compute a *negative score* by assessing the patterns on the set of negative events. Table 6.1 shows averages over 10 positive (resp. negative) score values obtained from 10 independent runs. The higher the positive score with regard to the negative score is, the more discriminative the pattern is.

Individual datasets and corresponding results are described in the following paragraphs. Notice that we used rather small sets of input events. One reason is that in some cases there are at most hundreds of events in total. Another reason is that graph patterns in cases like this are not very frequent, i.e. their support is rather low.

6.3.1 DBLP

The first graph used for experiments was graph created from DBLP dataset. In the experiment, we randomly selected 30 edges representing collaboration on an *icml* paper as positive events and 30 edges

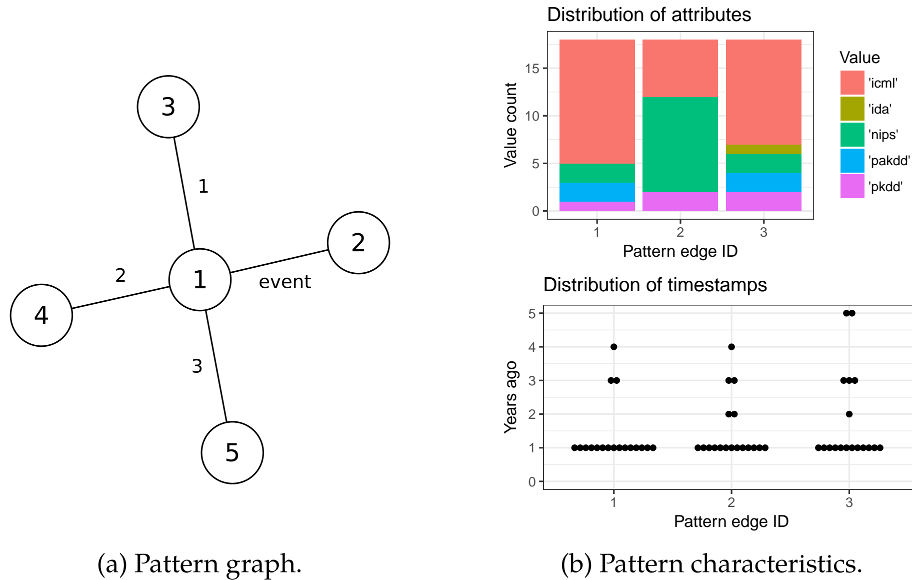
Table 6.1: Experiment results of WalDis for different parameter settings. 10 independent runs were executed for each setting and score on positive and negative event sets were averaged over these 10 runs.

	Parameters					Results	
Dataset	**Set size**	**Time unit** σ_1	**Time unit** σ_2	**Restart probability**	**Random walks**	**Positive score**	**Negative score**
DBLP	30	2 [years]	1 [years]	0.1	1000	0.28	0.13
	30	1 [years]	1 [years]	0.1	1000	0.30	0.14
	30	1 [years]	0.5 [years]	0.1	1000	0.25	0.15
	30	1 [years]	0.5 [years]	0.1	5000	0.27	0.13
	30	1 [years]	0.5 [years]	0.3	5000	0.26	0.12
TELCO	10	20 [days]	2 [days]	0.1	1000	0.24	0.21
	10	10 [days]	2 [days]	0.1	1000	0.24	0.22
	10	10 [days]	1 [days]	0.1	1000	0.25	0.22
	10	10 [days]	1 [days]	0.1	5000	0.24	0.19
	10	10 [days]	1 [days]	0.3	5000	0.25	0.21
ENRON	20	4 [days]	2 [days]	0.1	1000	0.52	0.35
	20	2 [days]	2 [days]	0.1	1000	0.54	0.38
	20	2 [days]	0.5 [days]	0.1	1000	0.39	0.23
	20	2 [days]	0.5 [days]	0.1	5000	0.37	0.24
	20	2 [days]	0.5 [days]	0.3	5000	0.39	0.27

representing *kdd* collaboration as negative events. Time unit parameter for primary edge selection, i.e. σ_1, was either one or two years and for secondary edge selection, i.e. σ_2, half a year or one year. One of the extracted patterns is depicted in Fig. 6.2. Specifically, Fig. 6.2a visualizes the pattern graph with the event edge, which has label *icml* in the dataset. The numbers on edges and vertices represent IDs. The characteristics of the pattern edges are shown in Fig. 6.2b. The first graph shows the distributions of labels across all pattern edges. For example, pattern edge with ID 1 has an *icml* label in 13 positive instances, *nips* as well as *pakdd* in 2 instances and *pkdd* in 1 instance. Similarly for the other two pattern edges. The pattern edges mostly represent collaboration on *icml* or *nips* papers. Obviously, the pattern does not cover the remaining 12 instances from the positive set. The second graph shows the distributions of the relative timestamps with respect to the the timestamps of the events. We can see that most of the 18 instance edges of all three pattern edges have a timestamp older by one year in comparison to the timestamps of the event edges. Thus, the *icml* and *nips* collaborations typically happen one year before the event, but sometimes also two or more years. Table 6.1 shows that patterns found in this dataset have typically positive score twice as high as the negative score and are clearly discriminative.

6.3.2 Phone Call Network

The second experiment was performed on phone call network dataset TELCO. Recall that the graph was created from phone calls of two consecutive months. In this experiment, we randomly picked 10 vertices representing customers who *churned*, i.e. left the operator, in the latter month and the same number of customers who did not churn. The vertices representing the non-churning customers were used as positive events with randomly assigned timestamps from the second month. The negative events were created from customers who churned with timestamps of the churn dates. Time unit for primary edge selection was 10 or 20 days and for the secondary edge selection 1 or 2 days. One of the patterns is depicted in Fig. 6.3. The interpretation of the graphs is the same as in the case of DBLP dataset, with the exception of the *duration* attribute distributions that are visualized by a dot plot as it is a numeric attribute. An evaluation on a test set of the same



(a) Pattern graph.

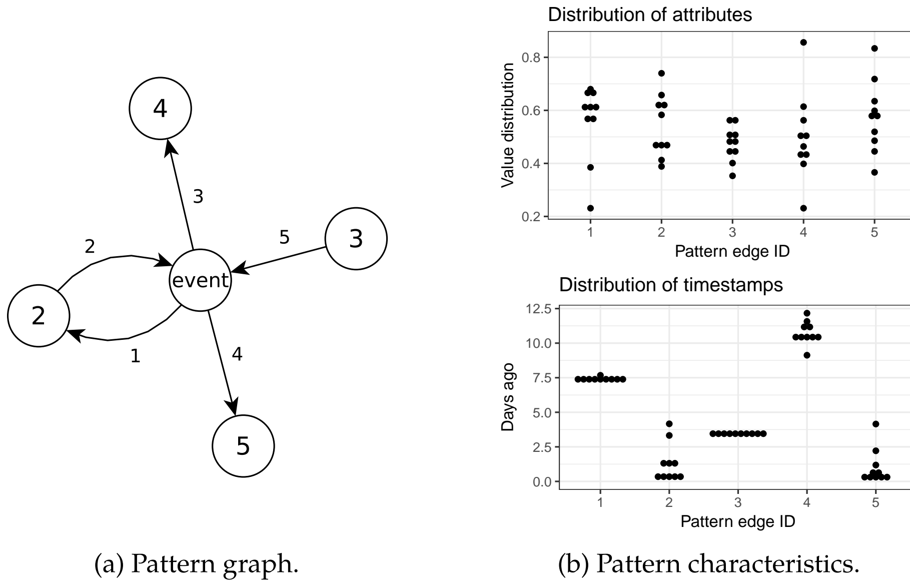
(b) Pattern characteristics.

Figure 6.2: A pattern found by WalDis in DBLP network dataset.

size shows that the pattern match slightly better in positive instances than in negative instances. Further investigation revealed that due to high density of edges it is probably not easy to find patterns that are in the neighbourhood of the positive but not negative events.

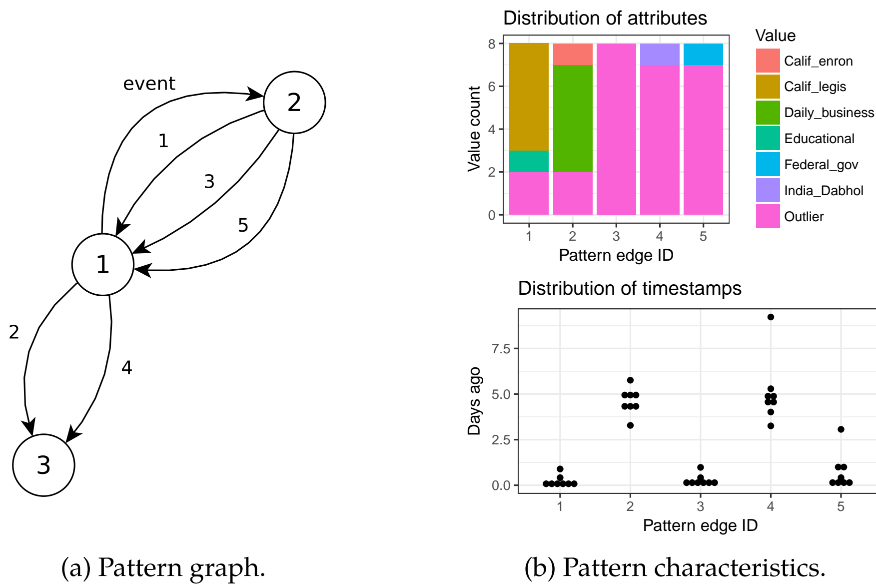
6.3.3 ENRON

Email correspondence of ENRON dataset was used as the last graph for experiments. 20 edges labelled as "California bankruptcy" were chosen as positive edge events and 20 edges labelled as "California business" as negative events. Time unit for primary edge selection was 2 or 4 days and for secondary edge selection 12 or 48 hours. Again, one of the pattern is shown in Fig. 6.4. The first pattern edge typically covers emails about California legislature (*Calif_legis* label) and the second one emails about daily business issues such as meetings (*Daily_business* label). Unfortunately, a lot of edges covered by the pattern are labelled as *Outlier* and this is probably inevitable as there are approximately 67% of all edges labelled as *Outlier*. Table 6.1 shows that



(a) Pattern graph. (b) Pattern characteristics.

Figure 6.3: A pattern found by WalDis in TELCO network dataset.



(a) Pattern graph. (b) Pattern characteristics.

Figure 6.4: A pattern found by WalDis in ENRON network dataset.

the positive scores are significantly higher than the negative scores, although not as much as for DBLP dataset.

6.4 WalDis Based on a Genetic Algorithm

We showed a discriminative pattern mining algorithm using a greedy approach in previous paragraphs. However, the greedy approach chooses edges from positive instances without considering the negative instances. We thus created another version of WalDis that is based on a genetic algorithm, which controls the selection process with respect to negative instances. This new version, called EWalDis, has not been published elsewhere yet and it is described in the rest of this section. Results of experiments are provided in the following section.

EWalDis has the same general structure as WalDis, see Algorithm 7, and the first phase is very similar to the one described in Algorithm 8. However, there some differences because of the genetic algorithm used in the second phase. First, the random selection of the primary edge is based on a uniform distribution and it is not driven by function $r(e,t)$. We use this simpler selection process because the full responsibility for pattern construction is left to the pattern construction phase. Second, instead of adding similarity $s(e_1,e_2)$ to s_{e_1,e_2}, see line 14 in Algorithm 8, we assign the value of $\frac{1}{2}(r(e_1,t_{i_1})s(e_1,e_2)+r(e_2,t_{i_2})s(e_2,e_1))$ into s_{e_1,e_2}. Thus, the final similarity s_{e_1,e_2} is computed by weighting the original similarity $s(e_1,e_2)$ by $r(e_1,t_{i_1})$ and $r(e_2,t_{i_2})$. This ensures that value s_{e_1,e_2} is high if edges e_1 and e_2 are similar to each other and their timestamps are close to the timestamps of corresponding events. Values s_{e_i,e_j} are used to compute fitness in the genetic algorithm, which we describe next.

The second step of EWalDis, stated as Algorithm 10, completely replaces the second step used by WalDis. It extracts a pattern by using a genetic algorithm and the similarities s_{e_i,e_j} computed in the first step. Throughout this section we use examples from Fig. 6.5 to explain particular concepts. In this figure, there are two positive and two negative instances, and also two edges were already selected for the pattern: e_{11} and e_{21}.

Algorithm 10 construct_pattern_genetic(G, similarities, positive_instances, negative_instances, size_of_pattern_graph)

```
 1: pattern_edges ← ∅
 2: for i ← 1 to size_of_pattern_graph do
 3:     pos_p ← init_edges(G, positive_instances, pattern_edges);
 4:     for j ← 1 to number_of_epochs do
 5:         pos_p ← pos_p ∪ crossover(pos_p);
 6:         pos_p ← pos_p ∪ mutation(pos_p);
 7:         neg_p ← init_neg_edges(G, negative_instances, pos_p);
 8:         for k ← 1 to number_of_subepochs do
 9:             neg_p ← neg_p ∪ crossover(neg_p);
10:             neg_p ← neg_p ∪ mutation(neg_p);
11:             neg_p ← evaluate_and_select_negative(pos_p,
                                  neg_p, similarities);
12:         pos_p ← evaluate_and_select_positive(pos_p,
                                  neg_p, similarities);
13:     pattern_edges ← pattern_edges ∪ best_individual(pos_p)
14: return pattern_edges
```

We defined the positive pattern as a set of graphs in Section 6.1. However, the pattern can be seen as lists of edges, where each list contains edges from positive instances that are mapped to the same edge of the pattern's union graph. In each list there is *at most* one edge from each positive instance, which allows us to perform inexact matching if there are no appropriate edges in some instances. Algorithm 10 creates one such edge list (pattern edge) in each cycle of the for loop beginning at line 2. Such an edge list is called a *positive individual* and these individuals are grouped into *positive populations*. The selection of the most suitable positive individual to the pattern starts at line 3 of the algorithm, where positive populations *pos_p* are initialized randomly. In Fig. 6.5 there is an example of two positive populations and the first one consists of three individuals (e_{12}, e_{21}), (e_{13}, e_{21}), and (e_{14}, e_{23}). The individuals are divided into several populations so that we do not mix edges that would break isomorphism with regard to already selected edges. For example, edges e_{16} and e_{23} cannot form

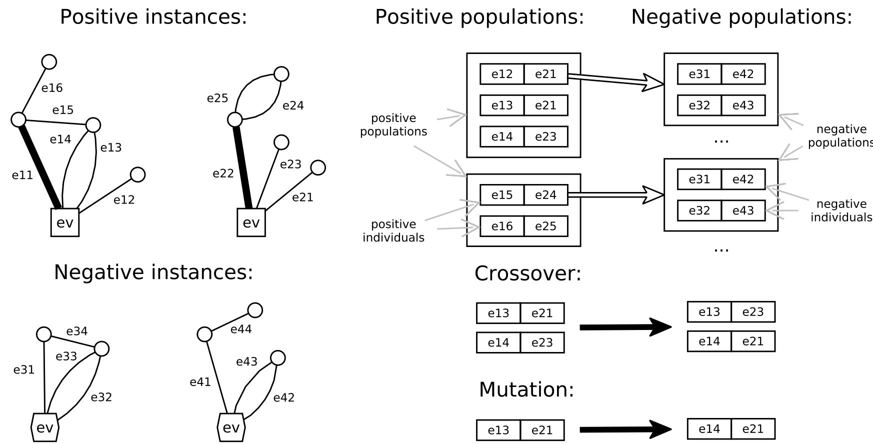


Figure 6.5: An illustrative example of notions used in the genetic algorithm. Highlighted edges e11 and e21 are assumed to be already selected.

an individual within a population with regard to already selected edges e_{11} and e_{21}.

The algorithm continues on lines 5 and 6, where new positive individuals are generated in each positive population by using crossover and mutation operators. For crossover, the implementation allows the user to choose either uniform or single-point crossover method. Mutation operator simply changes one edge randomly in each individual. Example usages of these operators are illustrated in Fig. 6.5. Both operators are executed separately within each positive population so that isomorphically inconsistent edges are not put together.

For all positive individuals in positive populations we need to pick the edges from negative instances that are most similar to these individuals in order to compute their fitness. This is performed inside another genetic algorithm, which is nested in the original one, on lines 7 to 11 of the algorithm. For each positive individual, one negative population is created on line 7. Such a negative population consists of negative individuals formed by edges from negative instances that appear in *similarities* statistics together with the edges from the considered positive individual. For the given number of *subepochs*, it creates new

negative individuals by the same crossover and mutation operators and then selects the best negative individuals in compliance[3] with fitness F^- defined in Eq. 6.3. It uses *Stochastic universal sampling* [10] technique for selection. After running the nested genetic algorithm, the algorithm continues on line 12 and computes the fitness F of positive individuals in compliance[4] with Eq. 6.2 and Eq. 6.4 by using F^- of the best negative individuals selected earlier. The same selection technique is used as before. The algorithm then continues on line 4 and repeats the for loop for the given number of *epochs* in order to refine the positive populations.

Finally, the algorithm selects a single individual with the highest fitness F across all positive populations and adds it to the selected pattern edges. After the desired number of individuals, given by $size_of_pattern_graph$ parameter, is selected, the algorithm outputs all selected edges as a final pattern. The algorithm can finish earlier, if the best selected individual has fitness < 0.1.

6.5 Patterns Extracted by the Genetic Version of WalDis

A C++ implementation of EWalDis was experimentally evaluated on DBLP and ENRON network datasets. We did not use TELCO dataset because of the reasons mentioned in Section 6.3.

To assess EWalDis, we extended the method used for WalDis and trained and evaluated a classification model on a data created from discriminative patterns. We used k-NN classifier with $k = 3$ as our model.

Given a train and a test set of events, both of size N, we first discovered patterns by the following procedure. We repeatedly selected n events from the train set at random and ran EWalDis on these events to get several patterns. Then we assessed the existence of such patterns on both the training and the test set by the same method as in Section 6.3. By using this procedure, we computed the matching score

3. Random walks, as a sampling method, do not compute similarities for all pairs of edges and thus F^- is normalized here by the number of obtained pairs, not by nm constant.
4. Once again, only the number of obtained pairs is used for normalization of F^+.

Table 6.2: Experiment results of EWalDis with accuracy computed for both training and test datasets.

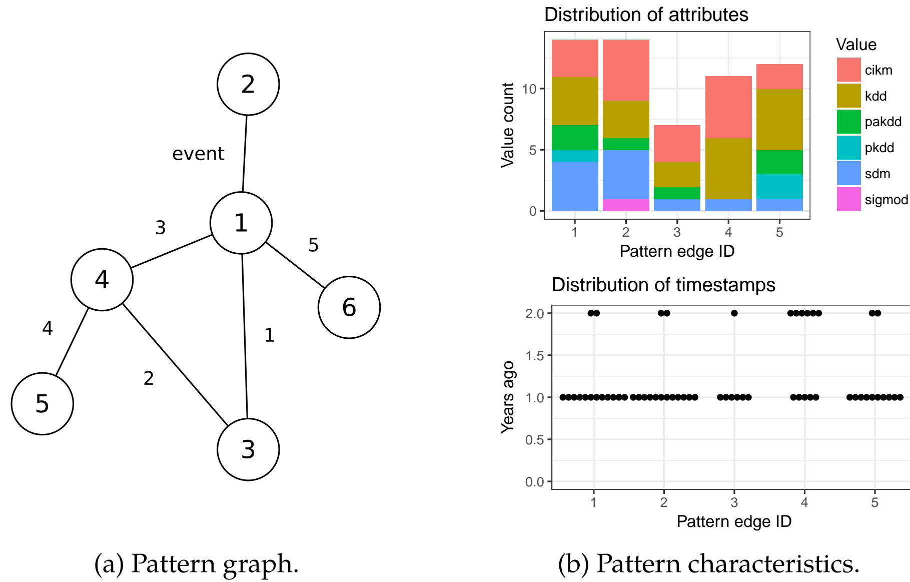
Settings			Resulting accuracy			
Dataset	Positive event	Negative event	Baseline		EWalDis	
			Train	Test	Train	Test
DBLP	KDD	ICML	73.5	67.5	80.5	74.5
DBLP	NIPS	KDD	68.0	54.5	78.5	82.0
DBLP	NIPS	ICML	69.5	52.5	79.0	70.0
ENRON	C. bankruptcy	C. business	87.5	67.5	90.0	90.0

for both positive and negative events from both training and test set[5]. Then we used these matching scores from training data as new features and learned k-NN model on these data. The model was then evaluated on a test set created from matching scores obtained on the original test set.

We also created a simple baseline method for comparison with EWalDis. The same classification algorithm was used, but the dataset features were prepared differently. Specifically, we created features from the edges adjacent to events. Each feature denotes an edge encoded by a pair (*label*, *relative_timestamp*). For each event, the features had value either 0 or 1 depending on whether there was such an edge adjacent to the event.

Summary results are shown in Table 6.2, where accuracy for each model is given, assessed both on training and test set. Results for each dataset are further described in the following paragraphs. In all experiments, we used uniform crossover technique in the genetic algorithm. We also tried single-point variant and the final accuracy was typically lower by 1%. We set the number of epochs and subepochs to 25 and the maximum number of edges per pattern to 5. For the same parameters, we also tried to increase the number of epochs and subepochs, but the increase of final accuracy was insignificant so 25 epochs and subepochs was enough for our datasets.

5. We called it either *positive score* or *negative score* in Section 6.3, depending on which instance was used.



(a) Pattern graph.

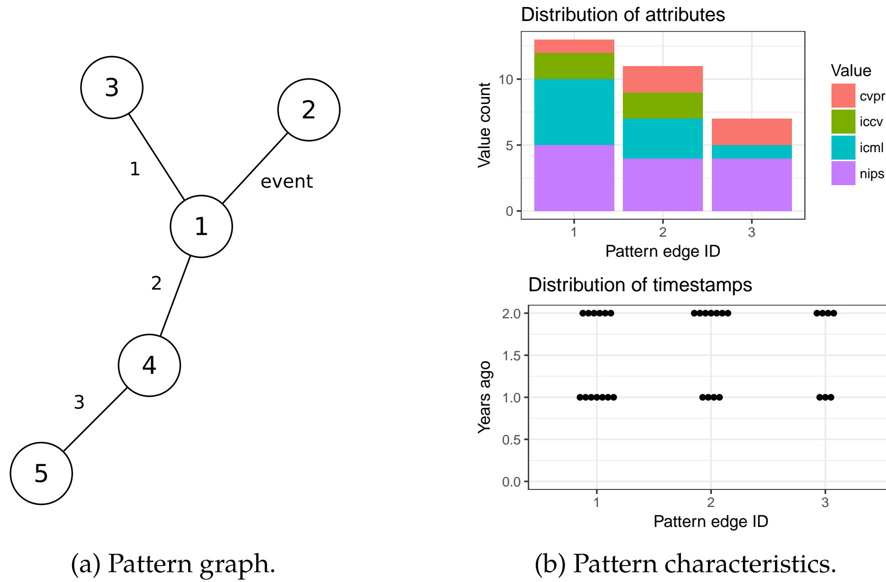
(b) Pattern characteristics.

Figure 6.6: A pattern found by EWalDis in DBLP network dataset with *kdd* publication as a positive event and *icml* as a negative event.

6.5.1 DBLP

In the experiment with DBLP dataset, we tried various types of positive and negative events. First, we randomly selected 100 edges representing collaboration on *kdd* paper as positive events and 100 edges representing *icml* collaboration as negative events, thus $N = 200$. We mined 20 patterns from samples of size $n = 60$. Furthermore, number of random walks was 1000. Time unit for primary edge selection was one year and for secondary edge selection half a year. We also tried *nips* vs. *kdd* and *nips* vs. *icml*. For all scenarios, the results of the classifier are shown in Table 6.2. As we can see, features created from patterns found by EWalDis are more useful for prediction than simple features created by the baseline method.

One of the patterns extracted during the first scenario, i.e. *kdd* as positive events and *icml* as negative ones, is depicted in Fig. 6.6. As with WalDis, Fig. 6.6a visualizes the pattern graph with the event edge and the characteristics of the pattern edges are shown in Fig. 6.6b. For example, pattern edge with ID 1 has an *sdm* label in 4 positive



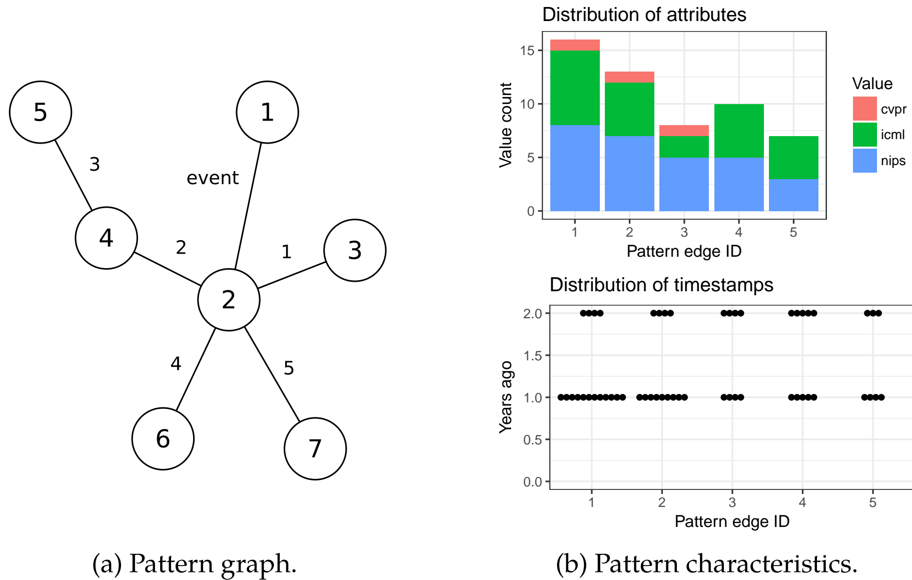
(a) Pattern graph.

(b) Pattern characteristics.

Figure 6.7: A pattern found by EWalDis in DBLP network dataset with *nips* publication as a positive event and *kdd* as a negative event.

instances, *pkdd* in 1, *pakdd* in 2, *kdd* in 4, and *cikm* in 3 instances. Similarly for the other four pattern edges. The pattern edges mostly represent historical collaboration on *kdd*, *cikm* or *sdm* papers. The second graph again shows the distributions of the relative timestamps with respect to the timestamps of the events. Thus, the collaborations typically happened one or two years before the event.

Examples of patterns from other two experiments on DBLP dataset can be found in Fig. 6.7 and 6.8. From Fig. 6.7 we can see that for *nips* as positive events and *kdd* as negative events, there are mostly edges representing collaboration on papers from conferences oriented more on machine learning, such as *nips*, *icml*, *iccv*, or *cvpr*. When *icml* is used for negative events, see Fig. 6.8, then there are mostly collaborations on *nips* or *icml* papers. In this case, we can see that these conferences are not very dissimilar.



(a) Pattern graph.

(b) Pattern characteristics.

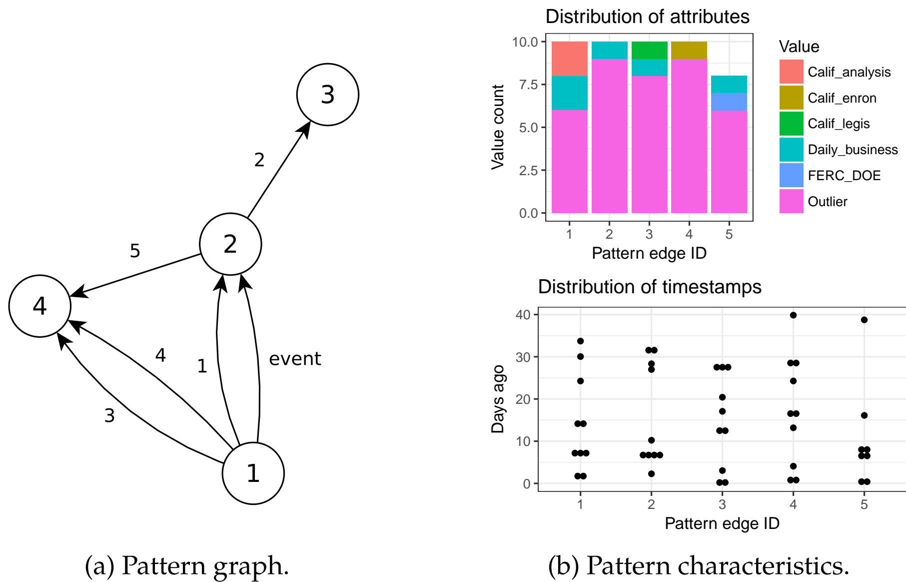
Figure 6.8: A pattern found by EWalDis in DBLP network dataset with *nips* publication as a positive event and *icml* as a negative event.

6.5.2 ENRON

As for ENRON dataset, we chose $N = 40$ as the size of training and test datasets, $n = 20$ as the size of 10 samples used for extraction of 10 discriminative patterns. Positive events were again represented by edges labelled as "California bankruptcy" and negative events by edges labelled as "California business". Time unit for primary edge selection was 20 days and for secondary edge selection 4 days, number of random walks was 3000.

An example of a pattern is shown in Fig. 6.9. The extracted edges covered similar topics as in case of WalDis, i.e. a lot of edges were labelled as *Outlier* and there were also topics about California analysis (*Calif_analysis* label) and daily business issues (*Daily_business* label).

For this experiment, we also show the feature statistics of the training and test datasets created from the EWalDis patterns in Fig. 6.10. There is a distribution of values for each pattern for both positive and negative class. As we can see, the values are much higher for positive



(a) Pattern graph. (b) Pattern characteristics.

Figure 6.9: A pattern found by EWalDis in ENRON network dataset with *California bankruptcy* email topic as a positive event and *California business* as a negative event.

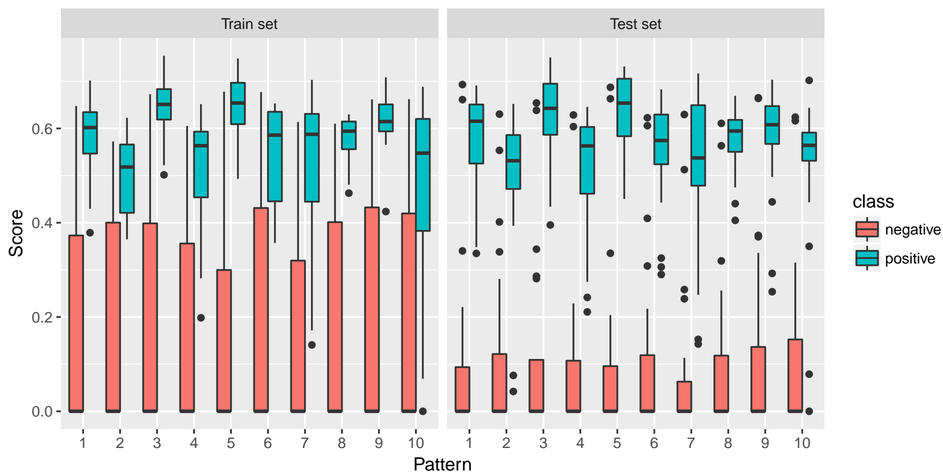


Figure 6.10: Matching score distributions of discriminative patterns found by EWalDis on ENRON dataset.

class, which confirms that the patterns are really discriminative and have higher matching score in positive instances.

6.6 Time Complexity

The time complexity of the algorithm is given by the parameters. The average-case time complexity of the first phase can be roughly estimated as follows. For the sake of simplicity, let us assume that the probability of edge non-selection is 0. Let p be the probability of random walk restart, d the average vertex degree, m number of random walks, n number of positive and negative events, and a the number of attributes used for edge similarity calculation. The expected number of steps in one walk is 1/p according to the geometric distribution. Then the average-case time complexity of the first phase is $\mathcal{O}(m * 1/p * d * n * d * a)$.

Regarding the second phase for pattern extraction, let k be the maximum number of pattern edges to be extracted. For the sake of brevity, let us omit minor operations such as checking that the newly extracted edges do not break injective mappings between pattern vertices across instances. Maximum possible number of edge pairs in similarity statistics can be generated if each pair of edges is walked simultaneously at most once. Then the expected value of this maximum number is $m * 1/p$. For each of the k pattern edges, the greedy-version of algorithm will inspect all the recorded edge pairs in the worst case. Thus, the average-case time complexity can be bound by $\mathcal{O}(k * m * 1/p)$ in case of the greedy-approach.

The genetic-algorithm approach is a bit more complex to analyse, mostly because of the procedure for population preparation, which have to take into consideration already selected edges and it cannot put together edges that would break isomorphism if they were selected together for a pattern edge. However, the main parameters influencing the complexity are the maximum number of pattern edges to be extracted k, the number of epochs and the number of subepochs, as can be seen from Algorithm 10. For each epoch, the algorithm has to perform crossover, mutation, selection, and the inner part the genetic algorithm, which in turn also performs crossover, mutation, and selection. Crossover operator have to create two new individuals and thus

its complexity is given by the number of graph instances, i.e. length of the individuals, times the number of individuals to be created. Mutation operator also creates new individuals so its complexity is given by the same factors. More complex are the functions computing the fitness of individuals because they use Eq. 6.2 and 6.3 and the complexity is $\mathcal{O}(n^2)$ for one individual. Simplifying this complexity is one of the possible future directions.

6.7 Discussion

In this chapter, we presented WalDis, an algorithm for discriminative pattern mining in temporal graphs, and its modification EWalDis. Both algorithms work on a local level of graphs and they are able to distinguish vertex or edge events. Furthermore, they work with attributed graphs and do not need discretization of timestamps.

We presented research related to discriminative pattern mining in graphs in Chapter 2. From the point of events on the level of vertices and edges, there are also methods close to this topic, even though they do not mine discriminative patterns. For example, a large group of methods is designed for link prediction in graphs [72, 107, 121]. There are also methods for vertex [111] or edge [4] classification. Similar area, prediction of vertex labels in graphs, is presented in [115].

There are also works dealing with *event detection* in graphs, such as [63, 93]. The first work focuses on spatial events and uses social media in order to discover them. Such a method can be useful for detection of disease outbreaks or civil unrests. The second work defines an event as a subset of nodes that are close to each other and have high activity levels, i.e. high values of a numeric attribute. It can be employed for discovering activity regions in sensor networks, for example. However, as we can see, these methods solve completely different tasks than WalDis or EWalDis.

7 Conclusion and Future Work

This thesis focuses on pattern mining in dynamic graphs. We presented several algorithms for mining various types of patterns in such graphs.

First, we showed in Chapter 3 how simple graph patterns, which can be extracted efficiently, may be used as features to represent resolution proofs created by students. However, to deal with more complex vertex labels, a generalization method based on domain knowledge was introduced. The usefulness of pattern-based representation was verified in a classification task. Moreover, by employing an outlier detection method on this representation, we were able to detect suspicious students' solutions that were not detected by a contemporary evaluator. This chapter also covers sequence mining approach for pattern mining in dynamic graphs. Specifically, we presented a different representation of resolution proof solutions. Each solution was represented by a sequence of actions a student performed when building the solution. Although the graph structure is not taken into account, the dynamics of temporal graphs is still captured by elements of such sequences. We showed that such a representation of dynamic graphs may be useful for analysis too. Subsequences found frequently in the input sequences were used as features and consequently for clustering of resolution proofs.

Mining of general graph rule patterns in dynamic graphs by DGRMiner was presented in Chapter 4. The advantage of this algorithm is that it is able to process graphs with various types of changes. Discovered rules can be used as pattern features for graph representation or as predictive patterns because they express how specific subgraphs change into different subgraphs. DGRMiner is useful both for directed and undirected graphs. An extension of DGRMiner for anomaly detection and explanation was provided in Chapter 5. Anomaly patterns mined by DGRMiner are in the form of rules too and use frequent patterns as explanatory normative patterns, from which they deviate. Such patterns can be employed for detection of suspicious behaviour or novel, not necessarily negative, patterns of behaviour.

Last but not least, Chapter 6 discussed algorithm WalDis and its modification EWalDis for discriminative pattern mining in dynamic

graphs. Both WalDis and EWalDis allow us to discover subgraph patterns that appear in the local neighbourhood of positive and not negative events. Thus, the patterns enable us to understand the context of positive events and how it differs from the context of negative events. Events may be given by a set of vertices or edges. The algorithms differ in approach they use. Specifically, WalDis uses a greedy approach and EWalDis uses a genetic algorithm to extract patterns. Both of them utilize a random-walk method to perform sampling and inexact matching to deal with the variability of attributes and timestamps in dynamic graphs.

As future work, it would be interesting to further investigate time abstraction methods of DGRMiner because time discretization determines the results quite significantly. We have seen that inexact matching used in WalDis and EWalDis allows us to mine patterns that are not completely identical. Time abstraction methods can help DGRMiner deal with such patterns, even though the algorithm uses exact matching approach. Another extension of DGRMiner could be the ability to mine frequent closed patterns, which should help reduce the number of generated patterns.

WalDis and EWalDis can be easily extended in order to be able to handle vertex additions and deletions, attribute changes, or larger events consisting of several vertices and edges. As far as these larger events contain the same number of vertices so that a bijective mapping can be established between vertices of any pair of events, the algorithms do not need substantial upgrades. Another possible direction of extending WalDis and EWalDis is usage of existing techniques in order to identify the initial events automatically. For example, mining of anomalous patterns by DGRMiner could be used for this task. First, the method would find several small patterns and then WalDis or EWalDis could use these small patterns as initial events, whose explanation is to be found. As we already discussed in the previous chapter, it would be also useful to improve the time complexity of the genetic algorithm used in EWalDis.

Bibliography

[1] A. Adiga, A. K. S. Vullikanti, and D. Wiggins. Subgraph Enumeration in Dynamic Graphs. In *13th IEEE International Conference on Data Mining* (ICDM '13). IEEE Computer Society, Washington, DC, USA, pp. 11–20, 2013.

[2] C. C. Aggarwal, Y. Zhao, and P. S. Yu. Outlier detection in graph streams. In *2011 IEEE 27th International Conference on Data Engineering* (ICDE '11), pp. 399–409, 2011.

[3] C. C. Aggarwal. *Mining Text Data*. New York: Springer, 2012.

[4] C. C. Aggarwal, et al. On Edge Classification in Networks with Structure and Content. In *2017 IEEE 33rd International Conference on Data Engineering*, (ICDE '17), pp. 187–190, 2017.

[5] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data* (SIGMOD '93). ACM, New York, NY, USA, pp. 207–216, 1993.

[6] R. Ahmed, and G. Karypis. Algorithms for Mining the Evolution of Conserved Relational States in Dynamic Networks. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining* (ICDM '11). IEEE Computer Society, Washington, DC, USA, pp. 1–10, 2011.

[7] L. Akoglu, H. Tong, and D. Koutra. Graph-based anomaly detection and description: A survey. In *Data Mining and Knowledge Discovery*, Vol. 29, no. 3, pp. 626–688, 2015.

[8] J. R. Anderson, H. S. Lee, and J. M. Fincham. Discovering the structure of mathematical problem solving. In *NeuroImage*, Vol. 97, pp. 163–177, 2014.

[9] M. Araujo, et al. Com2: Fast automatic discovery of temporal (comet) communities. In *Advances in Knowledge Discovery and Data Mining* (PAKDD), Springer, pp. 271–283, 2014.

[10] J. E. Baker. Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, Hillsdale, New Jersey, 1987.

[11] T. Barnes, and J. Stamper. Toward automatic hint generation for logic proof tutoring using historical student data. In *Proceedings of the 9th International Conference on Intelligent Tutoring Systems*, pp. 373–382, 2008.

[12] T. Barnes, and J. Stamper. Automatic hint generation for logic proof tutoring using historical data. In *Educational Technology and Society*, Vol. 13, no. 1, pp. 3–12, 2010.

[13] M. Berlingerio, F. Bonchi, B. Bringmann, and A. Gionis. Mining graph evolution rules. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases: Part I* (ECML PKDD '09). Springer-Verlag, Berlin, Heidelberg, pp. 115–130, 2009.

[14] F. Bi, et al. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (SIGMOD '16), ACM, San Francisco, California, USA, pp. 1199–1214, 2016.

[15] L. Breiman. Random Forests. In *Machine Learning*, Vol. 45, no. 1, pp. 5-–32, 2001.

[16] K. M. Borgwardt, H.-P. Kriegel, and P. Wackersreuther. Pattern Mining in Frequent Dynamic Subgraphs. In *Proceedings of the Sixth International Conference on Data Mining* (ICDM '06). IEEE Computer Society, Washington, DC, USA, pp. 818–822, 2006.

[17] B. Bringmann, M. Berlingerio, F. Bonchi, and A. Gionis. Learning and predicting the evolution of social networks. In *Proceedings of IEEE Intelligent Systems*. Vol. 25, no. 4, pp. 26–35, 2010.

[18] B. Bringmann, and S. Nijssen. What is frequent in a single graph? In *Proceedings of the 12th Pacific-Asia conference on Advances in knowledge discovery and data mining* (PAKDD '08), Takashi Washio, Akihiro Inokuchi, Einoshin Suzuki, and Kai Ming Ting (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 858–863, 2008.

[19] T. Calders, J. Ramon, and D. V. Dyck. All normalized anti-monotonic overlap graph measures are bounded. In *Data Mining and Knowledge Discovery*, Vol. 23, no. 3, pp. 503–548, 2011.

[20] L. Cerf, J. Besson, C. Robardet, and J.-F. Boulicaut. Closed patterns meet n-ary relations. In *ACM Transactions on Knowledge Discovery from Data* (TKDD). Vol. 3, no. 1, Article 3, 36 pages, March, 2009.

[21] J. Y. Chen, and S. Lonardi. *Biological Data Mining*. Boca Raton, FL: Chapman & Hall/CRC, 2010.

[22] Z. Chen, W. Hendrix, and N. F. Samatova. Community-based anomaly detection in evolutionary networks. In *Journal of Intelligent Information Systems*, Vol. 39, no. 1, pp. 59–85, 2012.

[23] F. Chen, and D. B. Neill. Non-parametric Scan Statistics for Event Detection and Forecasting in Heterogeneous Social Media Graphs. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, (KDD '14), pp. 1166–1175, 2014.

[24] Z. Cheng, F. Flouvat, and N. Selmaoui-Folcher. Mining Recurrent Patterns in a Dynamic Attributed Graph. In *Advances in Knowledge Discovery and Data Mining* (PAKDD), Springer, pp. 631–643, 2017.

[25] S. Choobdar, F. Silva, and P. Ribeiro. Network node label acquisition and tracking. In *Proceedings of the 15th Portugese conference on Progress in artificial intelligence* (EPIA '11), Luis Antunes and H. Sofia Pinto (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 418–430, 2011.

[26] W. W. Cohen. "Enron Email Dataset." Web. Accessed 2 June 2015. `https://www.cs.cmu.edu/~./enron/`.

[27] D. J. Cook, and L. B. Holder. Substructure Discovery Using Minimum Description Length and Background Knowledge. In *Proceedings of the 12th National Conference on Artificial Intelligence*, Seattle, WA, USA, Vol. 2, 1994.

[28] D. J. Cook, and L. B. Holder. Graph-Based Data Mining. In *IEEE Intelligent Systems*, Vol. 15, no. 2, pp. 32–41, 2000.

[29] D. J. Cook, and L. B. Holder. *Mining Graph Data*. John Wiley & Sons, 2006.

[30] X.-H. Dang, et al. Subnetwork Mining with Spatial and Temporal Smoothness. In *Proceedings of the 2017 SIAM International Conference on Data Mining* (SDM '17), pp. 354–362, 2017.

[31] M. Davis, W. Liu, P. Miller, and G. Redpath. Detecting anomalies in graphs with numeric labels. In *Proceedings of the 21st ACM Conference on Information and Knowledge Management* (CIKM '11), Glasgow, Scotland, pp. 1197—1202. ACM, 2011.

[32] E. Desmier, M. Plantevit, C. Robardet, and J.-F. Boulicaut. Trend Mining in Dynamic Attributed Graphs. In *Machine Learning and Knowledge Discovery in Databases*. Springer Berlin Heidelberg, pp. 654–669, 2013.

[33] P. Dickinson, H. Bunke, A. Dadej, and M. Kraetzl. On graphs with unique node labels. In *Proceedings of the 4th IAPR international conference on Graph based representations in pattern recognition* (GbRPR'03), Edwin Hancock and Mario Vento (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 13–23, 2003.

[34] G. Dong, and J. Pei. *Sequence data mining*. Springer, New York, 2007.

[35] A. Dovier, E. Pontelli, and G. Rossi. Set Unification. In *CoRR*, cs.LO/0110023, 2001.

[36] J. C. Dunn. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. In *Journal of Cybernetics 3* (3): 3257, 1973.

[37] W. Eberle, and L. B. Holder. Discovering structural anomalies in graph-based data. In *Proceedings of the International Workshop on Mining Graphs and Complex Structures at the 7th IEEE International Conference on Data Mining* (ICDM '07), Omaha, NE, pp. 393—398. IEEE Computer Society, 2007.

[38] W. Eberle, and L. Holder. Identifying Anomalies in Graph Streams Using Change Detection. In *KDD Workshop on Mining and Learning in Graphs* (MLG '16), August, 2016.

[39] M. Fiedler, and C. Borgelt. Support Computation for Mining Frequent Subgraphs in a Single Graph. In *Proceedings of the Fifth Workshop on Mining and Learning with Graphs* (MLG '07), 2007.

[40] R. A. Fisher. *Statistical Methods for Research Workers*. Oliver & Boyd, 1970.

[41] A. Fuksova, O. Kuzelka, and A. Szaboova. A method for mining discriminative graph patterns. In *NIPS Workshop on Machine Learning in Computational Biology*, 2013.

[42] M. Garey, and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, 1979.

[43] L. Getoor, and B. Taskar. *Introduction to statistical relational learning*. Cambridge, MA: The MIT Press, 2007.

[44] S. Gurukar, S. Ranu, and B. Ravindran. COMMIT: A Scalable Approach to Mining Communication Motifs from Dynamic Networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (SIGMOD '15), pp. 475–489, 2015.

[45] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. SIGKDD *Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[46] N. A. Heard, et al. Bayesian anomaly detection methods for social networks. In *Annals of Applied Statistics*, 4: pp. 645–662, January, 2010.

[47] J. Herold, A. Zundal, T. F. Stahovich. Mining Meaningful Patterns from Students' Handwritten Coursework. In *Proceedings of the 6th International Conference on Educational Data Mining* (EDM '13), 2013.

[48] N. Hewahi, and M. Saad. Class outliers mining: Distance-based approach. In *International Journal of Intelligent Technology*, Vol. 2, no. 1, pp. 55—68, 2007.

[49] A. Inokuchi, H. Ikuta, and T. Washio. Efficient Graph Sequence Mining using Reverse Search. In *IEICE Transactions on Information and Systems*, Volume E95.D, no. 7, pp. 1947–1958, 2012.

[50] A. Inokuchi, and T. Washio. A Fast Method to Mine Frequent Subsequences from Graph Sequence Data. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining* (ICDM '08). IEEE Computer Society, Washington, DC, USA, pp. 303–312, 2008.

[51] A. Inokuchi, and T. Washio. GTRACE2: Improving Performance Using Labeled Union Graphs. In *Proceedings of the 14th Pacific-Asia conference on Advances in Knowledge Discovery and Data Mining - Volume Part II* (PAKDD'10), Mohammed J. Zaki, Jeffrey Xu Yu, B. Ravindran, and Vikram Pudi (Eds.), Vol. Part II. Springer-Verlag, Berlin, Heidelberg, pp. 178–188, 2010.

[52] A. Inokuchi, and T. Washio. Mining frequent graph sequence patterns induced by vertices. In *Proc. of the Tenth SIAM International Conference on Data Mining* (SDM '10), 2010.

[53] R. Jin, S. Mccallen, and E. Almaas. Trend Motif: A Graph Mining Approach for Analysis of Dynamic Complex Networks. In *Proceedings of the 2007 Seventh IEEE International Conference on Data Mining* (ICDM '07). IEEE Computer Society, Washington, DC, USA, pp. 541–546, 2007.

[54] N. Jin, and W. Wang. LTS: Discriminative Subgraph Mining by Learning from Search History. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering* (ICDE '11), pp. 207–218, 2011.

[55] Y. Kabutoya, K. Nishida, K. Fujimura. Dynamic Network Motifs: Evolutionary Patterns of Substructures in Complex Networks. In *Proceedings of the 13th Asia-Pacific web conference on Web technologies and applications* (APWeb'11), Xiaoyong Du, Wenfei Fan, Jianmin Wang, Zhiyong Peng, and Mohamed A. Sharaf (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 321–326, 2011.

[56] L. Kaufman, and P. J. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. Hoboken, N.J.: Wiley, 2005.

[57] D. Koutra, E. Papalexakis, and C. Faloutsos. Tensorsplat: Spotting latent anomalies in time. In *16th Panhellenic Conference on Informatics (PCI)*, 2012.

[58] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks. In *Journal of Statistical Mechanics: Theory and Experiment*, November, 2011.

[59] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining* (KDD '05). ACM, New York, NY, USA, pp. 177–187, 2005.

[60] C. W.-K. Leung, E.-P. Lim, D. Lo, and J. Weng. Mining interesting link formation rules in social networks. In *Proceedings of the 19th ACM international conference on Information and knowledge management* (CIKM '10), ACM, New York, NY, USA, pp. 209–218, 2010.

[61] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. 2.nd ed. Berlin: Springer, 2011.

[62] Z. Liu, J. X. Yu, Y. Ke, X. Lin, and L. Chen. Spotting significant changing subgraphs in evolving graphs. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining* (ICDM '08). IEEE Computer Society, Washington, DC, USA, pp. 917–922, 2008.

[63] Y. Liu, et al. Graph Topic Scan Statistic for Spatial Event Detection. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (CIKM '16), ACM, New York, NY, USA, pp. 489–498, 2016.

[64] M. Macko. *Support in graph mining* (Bachelor's thesis). Masaryk University, Faculty of Informatics, Brno, 2018. `https://is.muni.cz/th/tgzml/`.

[65] E. A. Manzoor, S. M. Milajerdi, and L. Akoglu. Fast Memory-efficient Anomaly Detection in Streaming Heterogeneous Graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, USA, August 13-17, pp. 1035–1044, 2016.

[66] H.-H. Mao, et al. MalSpot: Multi2 Malicious Network Behavior Patterns Analysis. In *Advances in Knowledge Discovery and Data Mining* (PAKDD '14), Springer, pp. 1–14, 2014.

[67] R. Martinez, et al. Analysing frequent sequential patterns of collaborative learning activity around an interactive tabletop. In *Proceedings of the 4th International Conference on Educational Data Mining* (EDM '11), 2011.

[68] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network Motifs: Simple Building Blocks of Complex Networks. In *Science*, Vol. 298, no. 5594, pp. 824–827, 25 October, 2002.

[69] Y. Miyoshi, T. Ozaki, and T. Ohkawa. Mining Interesting Patterns and Rules in a Time-evolving Graph. In *Proceedings of the International MultiConference of Engineers and Computer Scientists* (IMECS '11), Vol. I, March 16–18, 2011.

[70] K. Nakagawa, et al. Safe Pattern Pruning: An Efficient Approach for Predictive Pattern Mining. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '16), San Francisco, California, USA, pp. 1785–1794, 2016.

[71] M. H. Namaki, et al. Discovering Graph Temporal Association Rules. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management* (CIKM '17), ACM, New York, NY, USA, pp. 1697–1706, 2017.

[72] S. Negi, and S. Chaudhury. Link Prediction in Heterogeneous Social Networks. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (CIKM '16), ACM, New York, NY, USA, pp. 609–617, 2016.

[73] J. Neil, et al. Scan Statistics for the Online Detection of Locally Anomalous Subgraphs. In *Technometrics*, Vol. 55, no. 4, pp. 403–414, 2013.

[74] L. Nezvalová, L. Popelínský, L. Torgo, and K. Vaculík. Class-Based Outlier Detection: Staying Zombies or Awaiting for Resurrection?

In *Advances in Intelligent Data Analysis XIV - 14th International Symposium* (IDA '15), pp. 193–204, 2015.

[75] K-N. T. Nguyen, L. Cerf, M. Plantevit, and J-F. Boulicaut. Discovering Inter-Dimensional Rules in Dynamic Graphs. In *Workshop on Dynamic Networks and Knowledge Discovery DyNaK'10 co-located with ECML PKDD 2010*, Barcelona, Spain. CEUR Workshop Proceedings, 2010.

[76] K-N. T. Nguyen, L. Cerf, M. Plantevit, and J-F. Boulicaut. Multidimensional association rules in boolean tensors. In *Proceedings of the 2011 SIAM International Conference on Data Mining* (SDM), 2011.

[77] K-N. T. Nguyen, L. Cerf, M. Plantevit, and J-F. Boulicaut. Discovering descriptive rules in relational dynamic graphs. In *Intelligent Data Analysis - Dynamic Networks and Knowledge Discovery*, Vol. 17, no. 1, pp. 49–69, January, 2013.

[78] K-N. T. Nguyen, M. Plantevit, and J-F. Boulicaut. Mining Disjunctive Rules in Dynamic Graphs. In *Proc. 9th IEEE RIVF Int. Conf. on Computing and Communication Technologies, Research, Innovation, and Vision for the Future*, Ho Chi Minh City, Vietnam, pp. 74–79, 2012.

[79] C. C. Noble, and D. J. Cook. Graph-based anomaly detection. In *Proceedings of the 9th ACM International Conference on Knowledge Discovery and Data Mining* (SIGKDD '03), Washington, DC, pages 631–636, 2003.

[80] T. Ozaki, and M. Etoh. Correlation and Contrast Link Formation Patterns in a Time Evolving Graph. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining Workshops* (ICDMW '11). IEEE Computer Society, Washington, DC, USA, pp. 1147–1154, 2011.

[81] T. Ozaki, and T. Ohkawa. Discovery of Correlated Sequential Subgraphs from a Sequence of Graphs. In *Proceedings of the 5th International Conference on Advanced Data Mining and Applications* (ADMA '09), Ronghuai Huang, Qiang Yang, Jian Pei, João Gama,

Xiaofeng Meng, and Xue Li (Eds.). Springer-Verlag, Berlin, Heidelberg, pp. 265–276, 2009.

[82] S. Papadimitriou, and C. Faloutsos. Cross-outlier detection. In *Advances in Spatial and Temporal Databases*, Springer Berlin Heidelberg, pp. 199–213, 2003.

[83] E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *ECLM PKDD*, Bristol, UK, pp. 521–536, 2012.

[84] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in Temporal Networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pp. 601–610, 2017.

[85] Z. Pekarčíková. Supervised outlier detection. Master's thesis, Masaryk University, 2013. `http://is.muni.cz/th/207719/fi_m/diplomova_praca_pekarcikova.pdf`.

[86] C. E. Priebe, J. M. Conroy, D. J. Marchette, and Y. Park. "Scan Statistics on Enron Graphs." Web. Accessed 8 June 2015. `http://www.cis.jhu.edu/~parky/Enron`.

[87] G. Qin, L. Gao, J. Yang, and J. Li. Evolution pattern discovery in dynamic networks. In *Proc. IEEE Int. Conf. on Signal Processing, Communications and Computing* (ICSPCC), 2011.

[88] S. Ranu, et al. Mining discriminative subgraphs from global-state networks. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '13), pp. 509–517, 2013.

[89] S. Rayana, and L. Akoglu. Less is More: Building Selective Anomaly Ensembles. In *ACM Trans. Knowl. Discov. Data*, Vol. 10, no. 4, pp. 42:1–42:33, 2016.

[90] U. Redmond, M. Harrigan, and P. Cunningham. Identifying Time-Respecting Subgraphs in Temporal Networks. In *3rd International Workshop on Mining Ubiquitous and Social Environments* (MUSE), 2012.

[91] J. Rissanen. *Stochastic Complexity in Statistical Inquiry Theory*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1989.

[92] P. J. Rousseeuw. Silhouettes: a Graphical Aid to the Interpretation and Validation of Cluster Analysis. In *Computational and Applied Mathematics 20*: 5365, 1987.

[93] P. Rozenshtein, et al. Event Detection in Activity Networks. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '14), ACM, New York, NY, USA, pp. 1176–1185, 2014.

[94] K. Semertzidis, and E. Pitoura. Durable graph pattern queries on historical graphs. In *Proceedings of the 32nd IEEE International Conference on Data Engineering* (ICDE '16), pp. 541–552, 2016.

[95] K. Semertzidis, and E. Pitoura. Top-k Durable Graph Pattern Queries on Temporal Graphs. In *IEEE Transactions on Knowledge and Data Engineering* (TKDE '18), 2018.

[96] J. C. Stamper, M. Eagle, T. Barnes, and M. J. Croy. Experimental evaluation of automatic hint generation for a logic tutor. In *International Journal of Artificial Intelligence in Education*, Vol. 22, no. 1-2, pp. 3–17, 2013.

[97] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *SIGKDD*, Philadelphia, PA, pp. 374–383, 2006.

[98] K. Takabayashi, et al. Extracting Discriminative Patterns from Graph Structured Data Using Constrained Search. In *Advances in Knowledge Acquisition and Management*, Springer Berlin Heidelberg, pp. 64–74, 2006.

[99] K. Vaculík, L. Nezvalová, and L. Popelínský. Educational data mining for analysis of students' solutions. In *Artificial Intelligence: Methodology, Systems, and Applications* (AIMSA '14), London, Springer, pp. 150–161, 2014.

[100] K. Vaculík, L. Popelínský. Graph mining for automatic classification of logical proofs. In *Proceedings of the 6th International Conference on Computer Supported Education (CSEDU '14)*, 2014.

[101] K. Vaculik, L. Popelinsky, E. Mrakova, and J. Jurco. Tutoring and automatic evaluation of logic proofs. In *Proceedings of the 12th European Conference on e-Learning (ECEL '13)*, pp. 495-–502, 2013.

[102] K. Vaculík, L. Nezvalová, and L. Popelínský. Graph mining and outlier detection meet logic proof tutoring. In *Proceedings of EDM 2014 Ws Graph-based Educational Data Mining (G-EDM '14)*, CEUR-WS.org, pp. 43–50, ISSN 1613-0073, 2014.

[103] K. Vaculík. A Versatile Algorithm for Predictive Graph Rule Mining. In *Proceedings ITAT 2015: Information Technologies - Applications and Theory*, Prague, CEUR-WS.org, pp. 51–58, 2015.

[104] K. Vaculík, and L. Popelínský. DGRMiner: Anomaly Detection and Explanation in Dynamic Graphs. In *Advances in Intelligent Data Analysis XV - 15th International Symposium* (IDA '16), Stockholm, Sweden, pp. 308–319, 2016.

[105] K. Vaculík, and L. Popelínský. WalDis: Mining Discriminative Patterns within Dynamic Graphs. In *Proceedings of the 22nd International Database Engineering & Applications Symposium* (IDEAS '18), ACM, New York, NY, USA, pp. 95–102, 2018.

[106] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm, and K. M. Borgwardt. Frequent subgraph discovery in dynamic networks. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs* (MLG '10). ACM, New York, NY, USA, pp. 155–162, 2010.

[107] Z. Wang, C. Chen, and W. Li. Predictive Network Representation Learning for Link Prediction. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval* (SIGIR '17), ACM, New York, NY, USA, pp. 969–972, 2017.

[108] X. Yan, and J. Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference*

on Data Mining (ICDM '02). IEEE Computer Society, Washington, DC, USA, 2002.

[109] X. Yan, et al. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (SIGMOD '08), pp. 433–444, 2008.

[110] C. Yang, et al. Making pattern queries bounded in big graphs. In *Proceedings of the 2015 IEEE 31st International Conference on Data Engineering* (ICDE '15), pp. 161–172, 2015.

[111] P. Yang, and P. Zhao. A Min-Max Optimization Framework For Online Graph Classification. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management* (CIKM '15), ACM, New York, NY, USA, pp. 643–652, 2015.

[112] Y. Yang, et al. Diversified Temporal Subgraph Pattern Mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, USA, August 13-17, pp. 1965–1974, 2016.

[113] S. Yang, et al. Fast top-k search in knowledge graphs. In *Proceedings of the 2016 IEEE 32nd International Conference on Data Engineering* (ICDE '16), pp. 990–1001, 2016.

[114] Z. Yang, et. al. Diversified Top-k Subgraph Querying in a Large Graph. In *Proceedings of the 2016 International Conference on Management of Data* (SIGMOD '16), pp. 1167–1182, 2016.

[115] W. Ye, et. al. Learning from Labeled and Unlabeled Vertices in Networks. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (KDD '17), ACM, New York, NY, USA, pp. 1265–1274, 2017.

[116] J. S. Yoo, and M. H. Cho. Mining Concept Maps to Understand University Students' Learning. In *Proceedings of the 5th International Conference on Educational Data Mining* (EDM), 2012.

[117] C.-H. You, L.B. Holder, and D.J. Cook. Learning patterns in the dynamics of biological networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and*

data mining (KDD ’09). ACM, New York, NY, USA, pp. 977–986, 2009.

[118] N. E. Young. Greedy Set-Cover Algorithms. In *Encyclopedia of Algorithms*, Springer US, Boston, pp. 1–4, 2008.

[119] M. J. Zaki. Sequences Mining in Categorical Domains: Incorporating Constraints. In *Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM)*. pp. 422–429, .2000.

[120] M. J. Zaki. Efficiently mining frequent embedded unordered trees. In *Fundamenta Informaticae*, 66(1-2):33-52. Mar/Apr, 2005.

[121] P. Zhao, C. Aggarwal, and G. He. Link prediction in graph streams. In *2016 IEEE 32nd International Conference on Data Engineering* (ICDE ’16), pp. 553–564, 2016.

[122] B. Zong, et al. Behavior Query Discovery in System-Generated Temporal Graphs. In *Proceedings of the VLDB Endowment*, Vol. 9, no. 4, pp. 240–251, 2015.

A Author's Contribution

A.1 Publications

- K. Vaculík, and L. Popelínský. WalDis: Mining Discriminative Patterns within Dynamic Graphs. In *Proceedings of the 22nd International Database Engineering & Applications Symposium* (IDEAS '18), ACM, New York, NY, USA, pp. 95–102, 2018.
  Author's contribution: 75%

- K. Vaculík, and L. Popelínský. WalDis: Mining Discriminative Patterns within Dynamic Graphs. In *Proceedings of the 12th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science* (MEMICS '17), 2017.
  Author's contribution: 75%

- K. Vaculík, and L. Popelínský. DGRMiner: Anomaly Detection and Explanation in Dynamic Graphs. In *Advances in Intelligent Data Analysis XV - 15th International Symposium* (IDA '16), Stockholm, Sweden, pp. 308–319, 2016.
  Author's contribution: 75%

- K. Vaculík. A Versatile Algorithm for Predictive Graph Rule Mining. In *Proceedings ITAT 2015: Information Technologies - Applications and Theory*, Prague, CEUR-WS.org, pp. 51–58, 2015.
  Author's contribution: 100%

- L. Nezvalová, L. Popelínský, L. Torgo, and K. Vaculík. Class-Based Outlier Detection: Staying Zombies or Awaiting for Resurrection? In *Advances in Intelligent Data Analysis XIV - 14th International Symposium* (IDA '15), pp. 193–204, 2015.
  Author's contribution: 15%

- K. Vaculík, L. Nezvalová, and L. Popelínský. Educational data mining for analysis of students' solutions. In *Znalosti*, Praha, pp. 32–35, 2014.
  Author's contribution: 70%

- K. Vaculík, L. Nezvalová, and L. Popelínský. Educational data mining for analysis of students' solutions. In *Artificial Intelligence: Methodology, Systems, and Applications* (AIMSA '14), London, Springer, pp. 150–161, 2014.
  Author's contribution: 70%
- K. Vaculík, L. Nezvalová, and L. Popelínský. Graph mining and outlier detection meet logic proof tutoring. In *Proceedings of EDM 2014 Ws Graph-based Educational Data Mining (G-EDM '14)*, CEUR-WS.org, pp. 43–50, ISSN 1613-0073, 2014.
  Author's contribution: 70%
- K. Vaculík, and L. Popelínský. Graph Mining for Automatic Classification of Logical Proofs. In *Proceedings of the 6th International Conference on Computer Supported Education* (CSEDU '14), Portugal, SCITEPRESS – Science and Technology Publications, pp. 268–275, 2014.
  Author's contribution: 75%
- K. Vaculík, and L. Popelínský. Graph Mining for Automatic Classification of Logical Proofs. In *Datakon a Znalosti*, Ostrava, pp. 133–138, 2013.
  Author's contribution: 75%
- K. Vaculík, and L. Popelínský, E. Mráková, and J. Jurčo. Tutoring and Automatic Evaluation of Logic Proofs. In *Proceedings of the 12th European Conference on e-Learning* (ECEL '13), Sophia Antipolis, France, pp. 495–502, 2013.
  Author's contribution: 35%

A.2 Invited Talks

- K. Vaculík. GDPR & ePrivacy rule as an EU gift to non-EU technological competitors. Machine Learning Prague, Praha, 2018.
- K. Vaculík. Advanced data analysis on Hadoop clusters. Machine Learning Prague workshop, Praha, 2017.
- K. Vaculík. Graph Mining: Applications. WIKT & Data a Znalosti, Smolenice, 2016.

A.3 Supervising Works

- M. Macko. *Support in graph mining* (Bachelor's thesis). Masaryk University, Faculty of Informatics, Brno, 2018. https://is.muni.cz/th/tgzml/.