Graph Mining and Outlier Detection Meet Logic Proof Tutoring

Karel Vaculík
Knowledge Discovery Lab
Faculty of Informatics
Masaryk University
Brno, Czech Republic
xvaculi4@fi.muni.cz

Leona Nezvalová
Knowledge Discovery Lab
Faculty of Informatics
Masaryk University
Brno, Czech Republic
324852@mail.muni.cz

Luboš Popelínský
Knowledge Discovery Lab
Faculty of Informatics
Masaryk University
Brno, Czech Republic
popel@fi.muni.cz

ABSTRACT

We introduce a new method for analysis and evaluation of logic proofs constructed by undergraduate students, e.g. resolution or tableaux proofs. This method employs graph mining and outlier detection. The data has been obtained from a web-based system for input of logic proofs built at FI MU. The data contains a tree structure of the proof and also temporal information about all actions that a student performed, e.g. a node insertion into a proof, or its deletion, drawing or deletion of an edge, or text manipulations. We introduce a new method for multi-level generalization of subgraphs that is useful for characterization of logic proofs. We use this method for feature construction and perform class-based outlier detection on logic proofs represented by these new features. We show that this method helps to find unusual students' solutions and to improve semi-automatic evaluation of the solutions.

Keywords

logic proofs, resolution, educational data mining, graph mining, outlier detection

1. INTRODUCTION

Resolution method is, together with tableaux proof method, one of the advanced methods taught in undergraduate courses of logic. To evaluate a student solution properly, a teacher needs not only to check the result of a solution (the set of clauses is or is not contradictory) but also to analyse the sequence of steps that a student performed—with respect to correctness of each step and with respect to correctness of that sequence. We need to take into account all of that when we aim at building a tool for analysis of students' solutions. It has to be said that for an error detection (e.g. resolution on two propositional letters, which is the most serious one) we can use a search method. However, detection of an error does not necessarily mean that the solution was completely incorrect. Moreover, by a search we can hardly discover patterns, or sequence of patterns, that are typical for wrong solutions.

To find typical patterns in wrong solutions, we developed a new method for analysis of students' solutions of resolution proofs [13, 14] and showed its good performance. Solutions were manually rewritten into GraphML and then analysed. First, the frequent patterns were found by Sleuth [16], which was suitable for this type of data—unordered rooted trees. This algorithm finds all frequent subtrees from a set of trees for a given minimum support value. Such frequent subgraphs were generalized and these generalizations used as new attributes.

The main drawback of a frequent subgraph mining algorithm itself is its strong dependence on a particular task, i.e. on the input set of clauses that has to be proved, or unproved, as contradictory. Moreover, a 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.

In this paper we propose a novel way of subgraph generalization that solves the problems mentioned above and is independent on the input set of clauses. We show that by means of graph mining and class outlier detection, we are able to find outlying students' solutions and use them for the evaluation improvement.

The structure of this paper is following. Section 2 brings related work. In Section 3 we introduce the source data. In Section 4 we introduce the improved method for construction of generalized resolution graphs. In Section 5 we bring the main result—detection of anomalous student solutions. Discussion and conclusion are in Sections 6 and 7, respectively.

2. RELATED WORK

Overview of graph mining methods can be found in [5]. 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 [15] 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 [1]. Markov decision processes for generating hints to students in logic proof tutoring from historical data has been solved in [2, 3, 12].

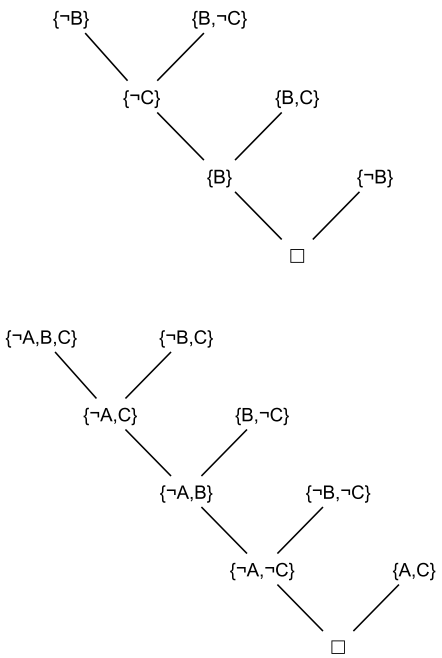

Figure 1: A correct and an incorrect resolution proof.

3. DATA

By means of a web-based tool, each of 351 students solved at least three tasks randomly chosen from 19 exercises. All solutions were stored in a PostgreSQL database. The data set contained 873 different students' solutions of resolution proofs in propositional calculus, 101 of them being incorrect and 772 correct. Two examples of solutions are shown in Fig. 1.

Common errors in resolution proofs are the following: repetition of the same literal in the clause, resolving on two literals at the same time, 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, resolving the clause and the negation of the second one (instead of the positive clause). For each kind of error we defined a query that detects the error. For automatic evaluation we used only four of them, see Table `ERRORS` described in appendix A. As the error of resolving on two literals at the same time is very common and referred later in text, we denote this error as E3.

All actions that a student performed, like adding/deleting a node, drawing/removing an edge, writing/deleting a text into a node, were saved into a database together with time stamps. More details on this database and its tables can be found in appendix A.

In the data there were 303 different clauses occurring in 7869 vertices, see frequency distribution in Fig. 2. Approximately half of the clauses had absolute frequency less than or equal to three.

4. GENERALIZED SUBGRAPHS

In this section we describe feature construction from graph data. Representing a graph 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 features [5]. More specifically, boolean features are used and the value of a feature depends on whether the corresponding substructure occurs in the given instance or not.

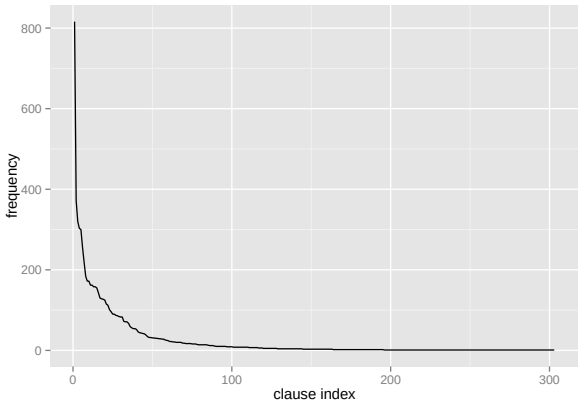

Figure 2: Distribution of clause labels ordered by frequency.

As we showed earlier, a frequent subgraph mining algorihm is inappropriate. 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 but 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. 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, the algorithm for frequent subgraphs was left out completely and all 3-node subgraphs, which are described later, were looked up.

4.1 Unification on Subgraphs

To unify different tasks that may appear in student tests, we defined a unification operator on subgraphs that allows finding of 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 a resolution rule), where A, B, C are propositional letters, is an instance of generalized graph $\{Z, \neg Y\}$, $\{Z, Y\} \rightarrow \{Z\}$, where Y, Z are variables (of type *proposition*). 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 [6] 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.

4.2 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

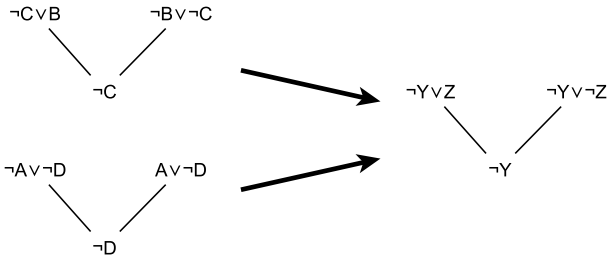

Figure 3: An example of pattern unification.

ordering on parent nodes[1]. 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, and (1,1) for C. Following the ordering Ω defined as follows: $(X,Y) \leq (U,V)$ iff $(X < U \vee (X = U \wedge Y \leq V))$, we have a result for the node $\{C, \neg B, A, \neg C\}$: $\{A, \neg B, C, \neg C\}$ with description Δ = ((0,1), (1,0), (1,1)). We will 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 Δ = ((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 second components.

4.3 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,... 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 $\{C, \neg B, A, \neg C\}, \{B, C, A, \neg A\} \rightarrow \{A, C\}$. We order the parents getting the result $\{B, C, A, \neg A\}, \{C, \neg B, A, \neg C\} \rightarrow \{A, C\}$. 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 Δ = ((1,1), (1,3), (1,3)). This leads to the following renaming of letters: $B \rightarrow Z$, $C \rightarrow Y$, and $A \rightarrow X$. Final generalized subgraph is $\{Z, Y, X, \neg X\}$, $\{Y, \neg Z, X, \neg Y\} \rightarrow \{X, Y\}$. 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 $\{\neg Y, Z\}$.

4.4 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 $O(m^2 + m(4r + 3r \log r))$.

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

4.5 Higher-level Generalization

To improve performance of used algorithms, e.g. outlier detection algorithms, we created a new 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, 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 number of occurrences of negative literals and secondly according to 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 number of occurrences can be listed as count(X, merged) = (1, 1), count(Y, merged) = (0, 1), 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. 1.

4.6 Generalization Example

In this section we illustrate the whole generalization process by an example. Assume that the following 3-node subgraph has to be generalized:

Table 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

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

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), 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'. As 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:

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

4.7 Use of Generalized Subgraphs

This section puts all the information from previous sections 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 attribute is equal to *TRUE*, if the corresponding pattern occurs in the given resolution proof, and it is equal to *FALSE* otherwise. Thus following this procedure, the resolution proofs can be transformed into an attribute-value representation as shown in Table 2. Such representation allows us to use a lot of existing machine learning algorithms.

Table 2: Attribute-value representation of resolution proofs

Instance	$Pattern_1$	$Pattern_2$	...	$Pattern_m$
1	TRUE	FALSE	...	FALSE
...	...	...	...	...
n	FALSE	FALSE	...	TRUE

5. OUTLIER DETECTION

5.1 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 only results computed 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 labeled. 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 [9] that looks for class outliers [8, 10] using Random Forests (RF) [4]. The main idea of weka-peka lies in different computation of proximity matrix in RF—it also exploits information about a class label [9]. We used the following settings:

```
NumberOfTrees=1000
NumberOfRandomFetaures=7
FeatureRanking=gini
```

Table 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) $\{\};\{\neg Y, \neg Z, Y, Z\}$

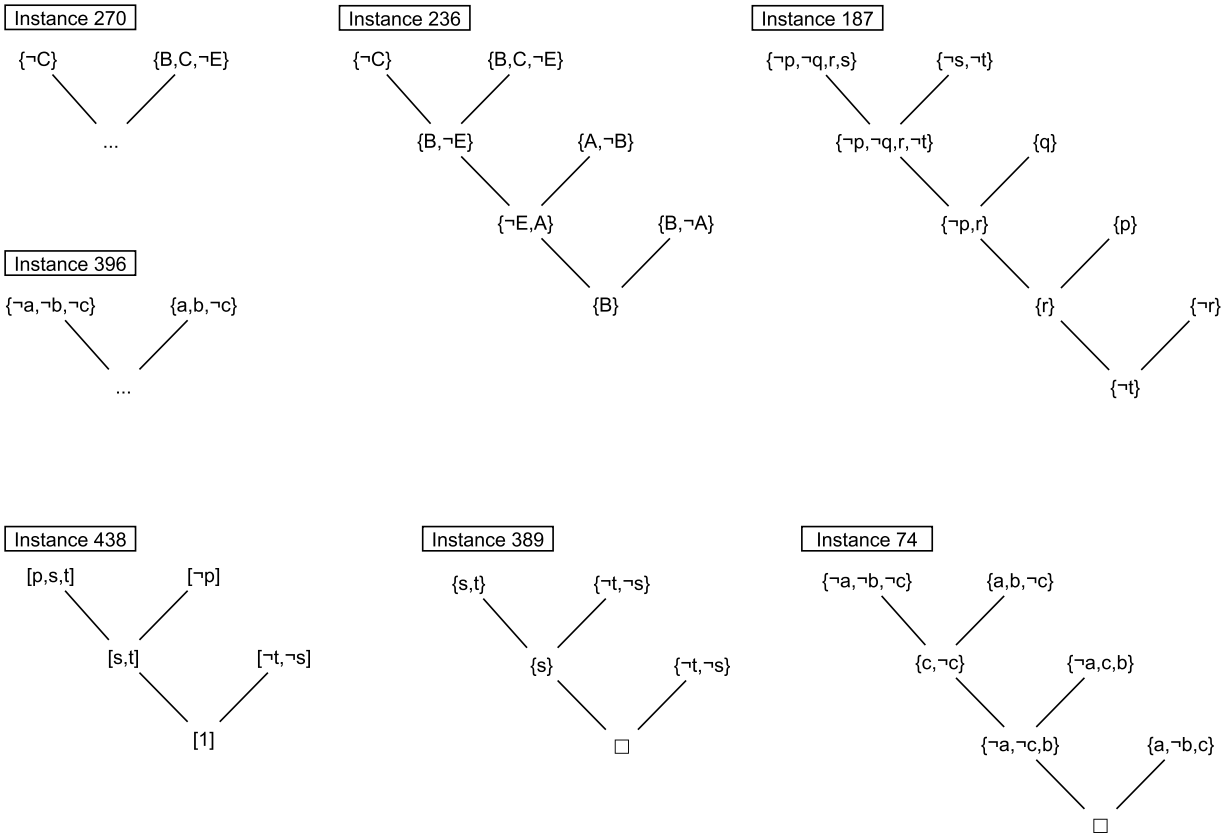

Figure 4: Drawings of the outlying instances from Table 3.

Table 4: Classification results for frequent subgraphs

Used attributes	Algorithm	Accuracy [%]	Precision for incorrect proofs	Recall
low-level generalization	SVM (SMO)	*95.2	0.94	0.61
both levels of generalization	SVM (SMO)	*96.9	0.95	0.74
both levels of generalization	J48	96.1	*0.98	0.68
both levels of generalization E3	J48	*95.4	0.87	0.72

```
MaxDepthTree=unlimited
Bootstrapping=yes
NumberOfOutliersForEachClass=50
```

Main results of outlier detection process are summarized in Table 3. When analyzing 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 analyzing trees that classify given instance to a different class than it was labeled. 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). When setting this attribute equal TRUE, outlier score decreases to -0,40. Values of attributes of instances 396 and 270 are equal, it means that also interpretation is the same as in previous case. Similary, we found that outlierness of instance 236 is given by occurence of specific pattern in solution and non-occurence of another pattern. The value of the corresponding attribute is the only difference between instance 236 and 187. Occurence/non-occurence of this pattern is therefore the reason why instance numbered 236 achieves higher outlier score than instance 187. See again Table 3 for information about particular patterns. We further elaborated this approach of outlier explanation in the following section.

5.2 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 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, the AScore ranges from -1 to 1. If the outlier resolution graph contains a pattern which is unique for the class of the graph, then the 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 the AScore is equal to -1. An AScore equal to 0 means that all other instances are equal to the outlier on the specified attribute.

5.3 Interpretation of the Outliers

Using the AScore metrics we found the patterns which are interesting for outliers in Table 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, except for the last one as it is almost identical to the penultimate one, are also displayed in Fig. 4. Analysis of individual outliers let us draw several conclusions. Let us remind that higher-level patterns listed in Table 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 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 resolution 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. 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, differ 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. 4. More interesting is the second instance with number 389. Error E3 was detected already in the first step of 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. 4. 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.

6. DISCUSSION

As we observed it is not sufficient to detect only the errors but we need to analyze a 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 the 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.

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 also learned various classifiers of that kind. In previous work we used only generalized patterns as attributes for classification with allerrors class attribute. However, these patterns were not sufficient for our current data. Repeating the same experiments we got the best result for SMO Support Vector Machines from Weka [7], which had 95.2% accuracy, see Table 4. 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. For the sake of completeness, F1 score for the class "correct" varied between 0.97 and 0.99 in all the results above.

We also checked whether inductive logic programming (ILP) can help to improve the performance under the same conditions. To ensure it, we did not use any domain knowledge predicates that would bring extra knowledge. For that reason, the domain knowledge contained only predicates common for the domain of graphs, like node/3, edge/3, resolutionStep/3 and path/2. We used Aleph system [11]. The results were comparable with the method described above.

7. CONCLUSION AND FUTURE WORK

In this paper we introduced a new level of generalization method for subgraphs of resolution proof trees built by students. Generalized subgraphs created by this special graph mining method are useful for representation of logic proofs in an attribute-value fashion. We 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. This method may also be used for other types of data such as tableaux proofs.

As a future work we are going to analyse the temporal information, which was saved together with the structural information of logic proofs.

ACKNOWLEDGEMENTS

This work has been supported by Faculty of Informatics, Masaryk University and the grant CZ.1.07/2.2.00/28.0209 Computer-aided-teaching for computational and constructional exercises.

8. REFERENCES

[1] J. R. Anderson. Discovering the structure of mathematical problem solving. In *Proceedings of EDM*, 2013.

[2] 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*, pages 373–382, 2008.

[3] T. Barnes and J. Stamper. Automatic hint generation for logic proof tutoring using historical data. *Educational Technology and Society*, 13(1):3–12, 2010.

[4] L. Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, Oct. 2001.

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

[6] A. Dovier, E. Pontelli, and G. Rossi. Set unification. *CoRR*, cs.LO/0110023, 2001.

[7] 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.

[8] N. Hewahi and M. Saad. Class outliers mining: Distance-based approach. *International Journal of Intelligent Technology*, 2.

[9] Z. Pekarcikova. Supervised outlier detection, 2013. `http://is.muni.cz/th/207719/fi_m/diplomova_praca_pekarcikova.pdf`.

[10] P. Spiros and F. Christos. Cross-outlier detection. In *Proceedings of SSTD*, pages 199–213, 2003.

[11] A. Srinivasan. The Aleph Manual, 2001. `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/` [Accessed: 2014-01-09].

[12] J. C. Stamper, M. Eagle, T. Barnes, and M. J. Croy. Experimental evaluation of automatic hint generation for a logic tutor. *I. J. Artificial Intelligence in Education*, 22(1-2):3–17, 2013.

[13] K. Vaculik and L. Popelinsky. Graph mining for automatic classification of logical proofs. In *Proceedings of the 6th International Conference on Computer Supported Education CSEDU 2014*, 2014.

[14] 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 2013*, pages 495–502, 2013.

[15] J. S. Yoo and M. H. Cho. Mining concept maps to understand university students' learning. In *Proceedings of EDM*, 2012.

[16] M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundam. Inf.*, 66(1-2):33–52, Jan. 2005.

APPENDIX
A. DESCRIPTION OF DATA

CLAUSE – list of nodes from all graphs
. idclause – ID of the node
. coordinatex – x position in drawing
. coordinatey – y position in drawing
. timeofcreation – when the node was created
. timeofdeletion – when the node was deleted (if not deleted, value is "NA")
. idgraph – in which graph the node appears
. text – text label

EDGE – list of (directed) edges from all graphs
. idedge – ID of the edge
. starting – ID of the node from which this edge goes
. ending – ID of the node to which this edge goes
. timeofcreation
. timeofdeletion
. idgraph

ERRORS – errors found in resolution graphs (found by means of SQL queries)
. idgraph – ID of the graph
. error3 – resolving on two literals at the same time (1 = error occurred, 0 = not occurred)
. error4 – repetition of the same literal in a set
. error5 – resolving on identical literals
. error8 – no resolution performed, only union of two sets
. allerrors – any of the previously listed errors occurred / not occured

GRAPH – list of graphs
. idgraph – ID of the graph
. logintime – start of graph creation
. clausetype – either set or ordered list
. resolutiontype – type of resolution, encoded by numbers (see table RESOLUTIONTYPES)
. assignment – textual assignment of task
. endtime – end of graph creation

MOVEMENT – list of coordinate changes of nodes
. idmovement – ID of the change
. idclause – ID of the node whose coordinates were changed
. coordinatex – new x coordinate
. coordinatey – new y coordinate
. time – time of the change

RESOLUTIONTYPES – encoding of resolution types
. typeid – ID (numeric encoding)
. typetext – textual value

TEXT – list of text (label) changes of nodes.
. idtext – ID of the change
. idclause – ID of the node whose text label was changed
. time – time of the change
. text – new text (label) value

TYPES – list of resolution type and clause type changes
. idtypes – ID of the change
. resolutiontype – new value of resolution type for specific graph
. clasetype – new value of clause type for specific graph
. timeofchange – time of the change
. idgraph – ID of the graph whose values were changed