Learning Tree Pattern Transformations

Daniel Neider
TU Dortmund University, Germany
Center for Trustworthy Data Science and Security, UA Ruhr, Germany

Leif Sabellek
CONTACT Research, Germany

Johannes Schmidt
Jönköping University, Sweden

Fabian Vehlken
Ruhr University Bochum, Germany

Thomas Zeume
Ruhr University Bochum, Germany

Abstract

Explaining why and how a tree t structurally differs from another tree $t^\star$ is a question that is encountered throughout computer science, including in understanding tree-structured data such as XML or JSON data. In this article, we explore how to learn explanations for structural differences between pairs of trees from sample data: suppose we are given a set $\{(t_1, t_1^\star), \ldots, (t_n, t_n^\star)\}$ of pairs of labelled, ordered trees; is there a small set of rules that explains the structural differences between all pairs $(t_i, t_i^\star)$? This raises two research questions: (i) what is a good notion of "rule" in this context?; and (ii) how can sets of rules explaining a data set be learned algorithmically?

We explore these questions from the perspective of database theory by (1) introducing a pattern-based specification language for tree transformations; (2) exploring the computational complexity of variants of the above algorithmic problem, e.g. showing NP-hardness for very restricted variants; and (3) discussing how to solve the problem for data from CS education research using SAT solvers.

2012 ACM Subject Classification Theory of computation → Complexity theory and logic; Computing methodologies → Supervised learning; Theory of computation → Database theory

Keywords and phrases Tree pattern transformations, learning from positive examples, computational complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2025.24

Related Version *Long Version*: `https://arxiv.org/abs/2410.07708` [11]

Funding *Daniel Neider*: Supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grant 434592664.
Johannes Schmidt: Partially supported by the Swedish Research Council (VR), grant 2022-03214.
Fabian Vehlken: Supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grants 448468041 and 532727578.
Thomas Zeume: Supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation), grants 448468041 and 532727578.

Acknowledgements We are grateful to Florin Manea and Markus Schmid for insightful discussions.

1 Introduction

Explaining why and how a tree t structurally differs from another tree $t^\star$ is a question that is encountered throughout computer science, including in understanding tree-structured data such as XML or JSON data, analysis of formal syntax trees, or code optimization.

In this article, we explore how to learn explanations for structural differences between pairs of trees from sample data: suppose we are given a set $\{(t_1, t_1^\star), \ldots, (t_n, t_n^\star)\}$ of pairs of labelled, ordered trees; is there a small set Γ of rules that explains the structural differences between all pairs $(t_i, t_i^\star)$? This raises two research questions:

(i) What is a good notion of "rule" in this context?

(ii) How can sets of rules explaining a data set be learned algorithmically?

We explore these questions from the perspective of database theory.

Our initial motivation for addressing these questions comes from CS education research on understanding common mistakes and misconceptions of students in introductory courses on *Logic in Computer Science*. A common educational task for students in such courses is to model scenarios described in natural language by logical formulas. A teacher provides the description and a solution formula φ, and students try to model the description with a formula $\varphi^\star$. The student attempt is correct if it is equivalent to φ. Empirically, most students try to write a formula that resembles the "logical structure of the provided description" very closely. If the teacher provides solution formulas with that aspect in mind, e.g. by providing a formula with an implication for "if ..., then ..." statements rather than other equivalent formulas, a natural approach for understanding what kind of mistakes students make in modelling is to find explanations for structural differences between the syntax trees of the solution formula φ and $\varphi^\star$ provided by students. Besides modelling, another type of exercise is having to transform a formula into some normal form. Here, individual transformation steps by students may incorrectly transform a formula φ into a formula $\varphi^\star$. To identify the types of mistakes students make, it is also helpful to look for explanations for structural differences between such two formulas.

Within the educational support system Iltis [15], a large set of sample data from students has been collected, e.g. > 500 pairs $(\varphi, \varphi^\star)$ of propositional formulas for many modelling exercises. In a preliminary analysis of this data with a custom approach, common types of mistakes in propositional logic have been identified by Schmellenkamp et al. [14]. For scaling this analysis and to ensure that a broad set of explanations are taken into account, a more systematic approach is needed. This work takes a first step in this direction.

While our motivation comes from CS education research, we believe that the raised conceptual questions are relevant and interesting for tree-shaped data in general. We therefore state them in the language of database theory and address them with techniques from this field.

Contributions

Towards a notion of rules for explaining structural differences between pairs of trees, we introduce a tree pattern based transformation language for labelled, ordered trees (see Section 2). For us, a tree pattern essentially is a tree whose nodes are labelled with labels, node variables, and tree variables. Such a tree pattern can be matched into a tree, meaning that nodes of the pattern are injectively[1] mapped to nodes of the tree respecting the tree structure and labels. A tree pattern transformation is of the form $\sigma \rightsquigarrow \sigma^\star$ for two tree patterns σ and $\sigma^\star$. It is applied to a tree t by matching σ into t and constructing a tree $t^\star$ according to the shape of $\sigma^\star$ and according to how variables are assigned (see Figure 1 for an illustration).

[1] While injective semantics are less common for tree patterns, they are more natural in the context of tree transformations, see Section 2.

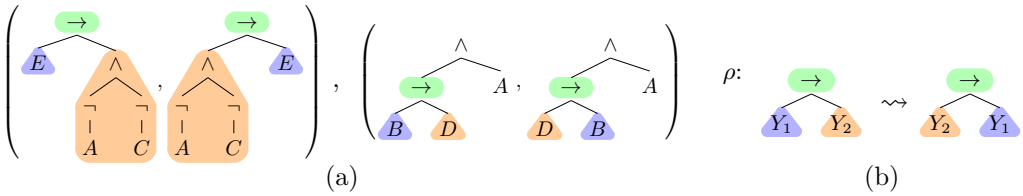


Figure 1 (a) Two pairs of syntax trees for the pairs $(E \to (\neg A \wedge \neg C), (\neg A \wedge \neg C) \to E)$ and $((B \to D) \wedge A, (D \to B) \wedge A)$ of propositional formulas. (b) A tree pattern transformation explaining the structural differences of both pairs. The transformation selects a $\to$-labelled node and swaps its subtrees Y_1 and Y_2.

We then study algorithmic properties of tree pattern transformations (see Section 3). We are particularly interested in variants of the following algorithmic problem:

Problem: LearningTreeTransformations
Input: Pairs $(t_1, t_1^\star), \ldots, (t_n, t_n^\star)$ of labelled, ordered trees, and $s, r \in \mathbb{N}$.
Find: A set Γ of at most r tree pattern transformations such that t_i can betransformed into $t_i^\star$ with transformations from Γ in at most s steps, for all i.

It turns out that this algorithmic problem is already NP-hard, even for very restricted inputs. For instance, we show that it is hard when fixing $s = 1$ as well as when fixing $s = 3$ and $r = 2$. Due to the hardness, we discuss how to attack the problem by encoding it into SAT and employing a SAT solver (see Section 3.2). We apply this approach exemplarily to educational data (see Section A).

Related work

Tree patterns have been studied extensively for tree-shaped data by the database theory community. Among other things, the computational complexity of algorithmic problems for tree patterns, such as containment and minimisation, has received much attention (see e.g. [6], also for an extensive list of references). Our notion of tree patterns slightly differs from the standard notion, as we require matches to be injective and use tree variables. We deviate to accommodate our motivation to learn structural differences, as discussed in the next section.

Manipulation of tree-shaped data has been studied in the context of tree transductions [9, 4], especially in the context of XML [16, 8]. Edit distances and similar measures have also been studied [3, 17]. We introduce another formalism for manipulating trees, tree pattern transformations, as it is unclear how to extract high-level explanations for structural differences between trees from existing formalisms.

Learning from examples has been studied before in the context of tree-shaped objects. Tree patterns representing queries on graphs are learned in [5]. The complexity of finding string patterns common to a set of strings is studied in [1]. Syntax trees of LTL and PSL formulas have been learned by employing SAT solvers in [10] and [12]. Our setting conceptually differs from these standard approaches as we aim at learning tree pattern transformations explaining differences between pairs of trees (instead of learning trees only). In the area of automata learning, learning from only positive examples has been studied [2, 13]. In contrast to all these approaches, typically we are interested in learning multiple transformations, whereas in the above learning problems only a single "object" is learned (e.g. a string pattern, an LTL formula, an automaton, and so on).

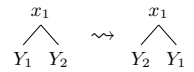
Outline

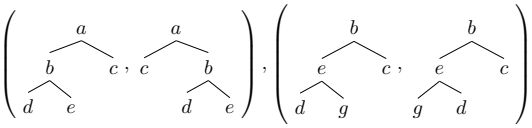
We introduce tree patterns and tree transformations based on these in Section 2. In Section 3, we study algorithmic properties of the problem LearningTreeTransformations and sketch a SAT-based approach to solve it in practice before presenting an extension of tree transformations in Section 4. Detailed proofs and additional explanations can be found in the long version of this paper [11].

2 Pattern-Based Tree Transformations

The goal of our tree transformation language is to explain structural differences between trees. Before formalizing our language, we discuss some of our design choices and fix notations.

Towards explaining structural differences between trees. In many contexts, structural differences between trees t and $t^\star$ are mostly local in the following sense: t can be transformed into $t^\star$ by "reshuffling" a small subtree t_v rooted at some node v of t and keeping the rest of t as is. This basic idea is already illustrated in Figure 1. Formally, a tree pattern transformation will be of the form $\sigma \rightsquigarrow \sigma^\star$ for two tree patterns σ and $\sigma^\star$ called *body* and *head*, respectively. A transformation is applied to a tree t by matching its body into t and manipulating the "covered" subtree as indicated by the head. To this end, body and head may use explicit labels, node variables to "copy" labels of single nodes, and tree variables to "copy" subtrees.

▶ **Example 1.** Consider the tree pattern transformation ρ:  with node variable x_1 and tree variables Y_1 and Y_2. The structural differences between the two pairs

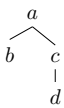
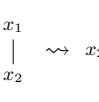


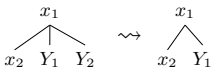
of trees are explained by ρ by applying it to the root in the first pair, and to the left child of the root in the second pair. The node variable x_1 then copies the labels a and e, respectively, and the tree variables Y_1 and Y_2 copy suitable subtrees. ⌟

Our goal to use tree pattern transformations to explain structural differences between trees requires to exactly capture the structure of trees. This influences our choice of semantics for tree patterns as follows. We require that

1. tree patterns can be matched anywhere in trees (and not just root nodes); and
2. tree pattern nodes with k children must be matched to tree nodes with exactly k children.

Requirement 1 allows for more flexible explanations as is already illustrated in Example 1. Requirement 2, although non-standard, is desirable in our context because we use tree patterns as a building block for tree transformations.

▶ **Example 2.** Consider the tree t:  and transformation ρ:  with node variables x_1 and x_2. Here, the tree pattern node labelled with x_1 has fewer children than the node labelled a in t. If matching ρ to t by mapping x_1 to the a-labelled node and x_2 to the b-labelled node was allowed, the result would be a single node labelled b. The c-labelled node and its d-labelled child would be removed without a "good explanation".

Now consider a slightly different transformation ρ':  in which the body's root node has more children than the a-labelled node in t. Matching the body of ρ' into t would require the embedding to be non-injective, for example, the tree variables Y_1 and Y_2 could both capture the subtree rooted at c. But then it is not clear what the result of applying ρ' to t would be, because the subtree rooted at c is supposed to be copied (due to Y_1) as well as removed (due to Y_2). To avoid this, we require embeddings to be injective. ⌟

We highlight that other design choices are possible, whose study we leave for future work.

Trees and tree patterns. A Σ-*labelled (ordered) tree* t is a prefix-closed subset $V \subseteq \mathbb{N}^*$ together with a labelling function $\ell\colon V \to \Sigma$. We also write trees t as (V, E, ℓ) where E is the set of edges induced by V, i.e. $(u, v) \in E$ if $vj = u$ for some $j \in \mathbb{N}$. The children of a node $v \in V$ are ordered according to the natural linear order on $\mathbb{N}$, i.e. for $u_1 \stackrel{\text{def}}{=} vj_1$ and $u_2 \stackrel{\text{def}}{=} vj_2$ we have $u_1 < u_2$ if $j_1 < j_2$. For a tree t and a node v, we write t_v for the subtree of t rooted at v. A *context* context(t, h) is a tree $t = (V, E, \ell)$ with a *hole* $h \in V$. A tree t' can be *plugged* into a context context(t, h) by replacing the node h by t' in the tree t.

A *tree pattern* σ is a $\Sigma \uplus \mathcal{N} \uplus \mathcal{T}$-labelled tree, where $\mathcal{T}$-labelled nodes must be leaves. The elements of Σ are called *labels*, the elements of $\mathcal{N}$ are called *node variables*, and the elements of $\mathcal{T}$ are called *tree variables*. A *match* μ of a tree pattern $\sigma = (V_\sigma, E_\sigma, \ell_\sigma)$ in a tree $t = (V_t, E_t, \ell_t)$ is an injective mapping $\mu\colon V_\sigma \to V_t$ from nodes of σ to nodes of t such that

1. labels are mapped consistently, i.e. $\ell_\sigma(v) = \ell_t(\mu(v))$ for all Σ-labelled nodes in V_σ;
2. node variables are mapped consistently, i.e. if $u, v \in V_\sigma$ are labelled with the same node variable, then the nodes $\mu(u)$ and $\mu(v)$ must have the same label;
3. tree variables are mapped consistently, i.e. if $u, v \in V_\sigma$ are labelled with the same tree variable, then the subtrees of t rooted at $\mu(u)$ and $\mu(v)$ are isomorphic; and
4. the number and order of children is preserved, i.e. if $v \in V_\sigma$ has the children $u_1 < \ldots < u_k$ then $\mu(v)$ has exactly the children $\mu(u_1) < \ldots < \mu(u_k)$.

Condition 4 is non-standard, but essential for an intuitive notion of tree pattern transformations (see discussion above).

Tree pattern transformations. The idea of tree pattern transformations is to select a subtree t_r of a tree t rooted at a node ν by matching a tree pattern σ, and then to construct a new tree $t^\star$ by plugging a re-combination $t^\star_\nu$ of t_ν according to a tree pattern $\sigma^\star$ into the context(t, ν). The patterns σ and $\sigma^\star$ may use the same node and tree variables, which allows for rearranging whole subtrees. For an example of a tree pattern transformation and its application we refer to Figure 1.

A *tree pattern transformation* $\rho\colon \sigma \rightsquigarrow \sigma^\star$ consists of two tree patterns σ and $\sigma^\star$ such that all variables occurring in $\sigma^\star$ also occur in σ. The pattern σ is called *body*, $\sigma^\star$ is called *head*, and ρ is called *name* of the transformation.

The semantics of tree pattern transformations is defined as follows. A tree pattern transformation $\rho\colon \sigma \rightsquigarrow \sigma^\star$ *transforms* a tree t into a tree $t^\star$, written as $t^\star \in \rho(t)$ or $t \rightsquigarrow_\rho t^\star$, if there is a match $\mu_{\sigma,t}$ of σ into t with root ν and a match $\mu_{\sigma^\star,t^\star}$ of $\sigma^\star$ into $t^\star$ with root $\nu^\star$ such that

1. t and $t^\star$ only differ on the subtrees t_ν and $t^\star_{\nu^\star}$, i.e. the contexts context(t, ν) and context$(t^\star, \nu^\star)$ are isomorphic (in particular, ν and $\nu^\star$ have the same positions in t and $t^\star$);

2. the subtree t_ν is rearranged into the subtree $t^\star_{\nu^\star}$, i.e.
   (i) if nodes u in σ and $u^\star$ in $\sigma^\star$ are labelled with a node variable x, then $\mu_{\sigma,t}(u)$ and $\mu_{\sigma^\star,t^\star}(u^\star)$ have the same label; and
   (ii) if nodes u in σ and $u^\star$ in $\sigma^\star$ are labelled with a tree variable X, then the subtrees of t and $t^\star$ rooted at $\mu_{\sigma,t}(u)$ and $\mu_{\sigma^\star,t^\star}(u^\star)$ are isomorphic.

When $t \rightsquigarrow_\rho t^\star$, we also say that the tree pattern transformation ρ *explains* the pair $(t, t^\star)$, as ρ can be seen as an explanation for the structural differences of t and $t^\star$. Testing whether ρ explains $(t, t^\star)$ is simple.

▶ **Proposition 3.** *Given a tree pattern transformation $\rho\colon \sigma \rightsquigarrow \sigma^\star$ and a pair $(t, t^\star)$ of trees, it can be tested in polynomial time whether ρ can transform t into $t^\star$, i.e. whether $t \rightsquigarrow_\rho t^\star$.*

Proof sketch. Try all pairs $(\nu, \nu^\star)$ of nodes of t and $t^\star$ as roots for matches of σ and $\sigma^\star$. ◀

For a set Γ of tree pattern transformations and a pair $(t, t^\star)$ of trees, we also write $t \rightsquigarrow^s_\Gamma t^\star$ if t can be transformed into $t^\star$ with transformations from Γ in (at most) s steps. We also say that Γ explains $(t, t^\star)$ in s steps. Note that we do not need an explicit wildcard label which is common for tree patterns, because it can be simulated by variables that only occur in the body of a transformation.

3 Learning Tree Pattern Transformations

In this section we study algorithmic properties of the learning problem for tree pattern transformations. Formally, we consider the following algorithmic problem:

Problem: LearningTreeTransformations
Input: Pairs $(t_1, t_1^\star), \ldots, (t_n, t_n^\star)$ of ordered, labelled trees, and $s, r \in \mathbb{N}$.
Find: A set Γ of at most r tree pattern transformations that explains all pairs $(t_i, t_i^\star)$ in at most s steps, i.e. a set Γ such that $t_i \rightsquigarrow^s_\Gamma t_i^\star$, for all i.

For a pair $(t_i, t_i^\star)$, we sometimes call t_i source tree and $t_i^\star$ target tree. Observe that the problem is trivial if $r \geq n$, because in this case, one transformation $\rho_i\colon t_i \rightsquigarrow t_i^\star$ for each pair of trees can be chosen.

3.1 Learning Tree Pattern Transformations Is Hard

It is easy to see that LearningTreeTransformations is NP-hard in general.[2] For this reason, we take a closer look at fragments obtained by restricting s and r. In practical applications, such as for the data from CS education research sketched in the introduction, s and r can be chosen to be very small. For instance, solving the above problem even when fixing the number of steps to $s = 1$ would help to identify common types of mistakes. Unfortunately, the learning problem remains NP-hard even for small numbers s of steps and small numbers r of transformations to learn.

▶ **Theorem 4.** LearningTreeTransformations is NP-complete when
1. fixing $s \stackrel{\text{def}}{=} 1$; or
2. fixing both $s \stackrel{\text{def}}{=} 3$ and $r \stackrel{\text{def}}{=} 2$.

[2] Formally, hardness is for the decision version of LearningTreeTransformations, i.e. "Is there such a set Γ?"

Membership in NP is straightforward for these cases. We note that membership in NP for the general problem, i.e. for arbitrary s and r, is less clear because intermediate trees can become exponentially large. Since s and r are small in our applications, we leave the general question for future work.

We prove the hardness of the two cases of Theorem 4 in Propositions 5 and 7.

▶ **Proposition 5.** LearningTreeTransformations is NP-hard, even when all examples are binary trees with labels from a binary alphabet and when $s \stackrel{\text{def}}{=} 1$ is fixed.

Proof sketch. We start by proving hardness for a large label alphabet Σ. It can be adapted to binary alphabets by encoding single labelled nodes as trees whose sequence of leaves represents the original label.

We reduce from the NP-complete problem VertexCover where, given a graph G with nodes $V = \{v_1, \dots, v_n\}$ and edges $E = \{e_1, \dots, e_m\}$ as well as a natural number k, one asks for a set of at most k nodes from V such that each edge from E is incident to at least one of these nodes (cf. [7]). Such a set of k nodes is called a *k-vertex cover*.

Construction. The idea for our reduction f is to construct from a VertexCover instance (G, k), an instance of LearningTreeTransformations with $s \stackrel{\text{def}}{=} 1$, $r \stackrel{\text{def}}{=} k$, and one example pair $(t_{u,v}, t^\star_{u,v})$ of trees for each edge $e = (u, v)$. The goal is that each k-vertex cover S of G corresponds to a solution set Γ_S of transformations of size k that allows to transform each $t_{u,v}$ into $t^\star_{u,v}$ in one step.

Let ℓ be minimal such that $n \leq 2^\ell$. For each edge $e = (u, v)$, construct the pair $(t_{u,v}, t^\star_{u,v})$ such that $t_{u,v}$ is the full binary tree of depth ℓ and $t^\star_{u,v}$ is a single node. We associate the first n leaves (from left to right) of $t_{u,v}$ with the nodes $v_1, \dots, v_n$ of the graph. In $t_{u,v}$, the leaves corresponding to u and v are labelled with $\ell_{u,v}$ and all other nodes are labelled with b. In $t^\star_{u,v}$, the single node is labelled with $\ell_{u,v}$. This construction is demonstrated in Example 6.

The reduction f is clearly computable in polynomial time.

Correctness. Suppose first that (G, k) is a positive VertexCover instance with vertex cover $S = \{v_{c_1}, \dots, v_{c_k}\}$. We construct a solution set Γ_S of transformations for $f(G, k)$. For each v_{c_i}, the set Γ_S contains one transformation $\rho_i \colon \sigma \rightsquigarrow \sigma^\star_i$ where

- σ is the full binary tree of depth ℓ, whose first n leaves are labelled with node variables $x_1, \dots, x_n$ and all other nodes are labelled with b; and
- $\sigma^\star_i$ consists of a single node labelled with the node variable x_{c_i}.

The set Γ_S of transformations is of size k and allows to transform each $t_{u,v}$ into $t^\star_{u,v}$ in one step. The latter can be seen as follows. The leaves of $t_{u,v}$ corresponding to u and v are labelled with $\ell_{u,v}$, while all other nodes have label b. Since S is a vertex cover of G, we have that $u \in S$ or $v \in S$, i.e. there exists i such that $u = v_{c_i}$ or $v = v_{c_i}$. Hence, applying the transformation ρ_i to $t_{u,v}$ yields a single node labelled $\ell_{u,v}$, that is, $t^\star_{u,v}$.

Suppose now that $f(G, k)$ is a positive LearningTreeTransformations instance with solution set $\Gamma = \{\rho_1, \dots, \rho_k\}$ of k transformations. We construct a k-vertex cover S_Γ from Γ. To this end we show that if a set Γ of size k solves $f(G, k)$, then its transformations must be of a certain form. From transformations of this form we then extract a k-vertex cover of G.

We observe that due to the structural differences between source and target trees in all pairs $(t, t^\star)$, a transformation $\rho \colon \sigma \rightsquigarrow \sigma^\star$ can transform t into $t^\star$ only, if

(a) the root of σ is matched to the root of t (because the whole tree t must be replaced by the single node tree $t^\star$); and

(b) $\sigma^\star$ is a single node (because $t^\star$ only has one node) and one of the following cases applies
- **(i)** $\sigma^\star = \ell_{u,v}$ for some edge (u, v);
- **(ii)** $\sigma^\star = x$ for some node variable x which occurs in σ at the end of a path of length ℓ; or
- **(iii)** $\sigma^\star = Y$ for some tree variable Y which occurs in σ at the end of a path of length ℓ.

The condition on σ in Cases (b)(ii) and (b)(iii) comes from the shape of our examples: all second components $t^\star$ of pairs are single nodes labelled with some $\ell_{u,v}$. Nodes with such a label occur only at leaves of trees t in first components. We note that in these cases, the leftmost path to x or Y in σ uniquely determines a leaf to which x is matched when σ is matched to the full binary tree of depth ℓ; denote the node of G corresponding to this leaf by $v_{\sigma,x}$ or $v_{\sigma,Y}$, respectively.

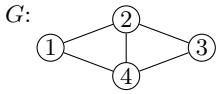
The k-vertex cover S_Γ of G is constructed from Γ as follows. For each $\rho\colon \sigma \rightsquigarrow \sigma^\star \in \Gamma$ that explains some example pair $(t, t^\star)$, we include exactly one node in S_Γ, depending on $\sigma^\star$:
- if $\sigma^\star = \ell_{u,v}$ for some edge (u, v), then include u in S_Γ;
- if $\sigma^\star = x$ for some node variable, then include $v_{\sigma,x}$ in S_Γ; and
- if $\sigma^\star = Y$ for some tree variable Y, then include $v_{\sigma,Y}$ in S_Γ.

To see that S_Γ is indeed a vertex cover of G, consider an arbitrary edge $e = (u, v) \in E$. Suppose that $\rho\colon \sigma \rightsquigarrow \sigma^\star \in \Gamma$ transforms $t_{u,v}$ into $t^\star_{u,v}$ (such a ρ exists as Γ solves $f(G, k)$). For each of the three cases (i)–(iii) of what $\sigma^\star$ can look like, a suitable node is included in S_Γ.

This concludes the hardness proof for LearningTreeTransformations with $s = 1$. ◀

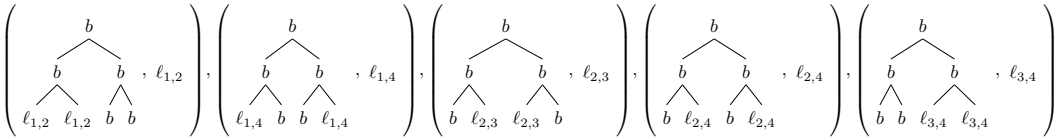
The following example illustrates the construction from the previous proof.

▶ **Example 6.** Consider the positive VertexCover instance with

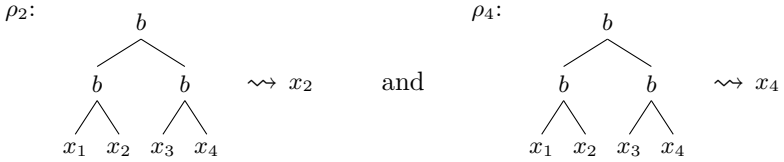


and $k = 2$.

We construct a LearningTreeTransformations instance with $s = 1$, $r = 2$, and the following pairs $(t_{1,2}, t^\star_{1,2}), (t_{1,4}, t^\star_{1,4})(t_{2,3}, t^\star_{2,3}), (t_{2,4}, t^\star_{2,4}), (t_{3,4}, t^\star_{3,4})$ of trees, one for each edge:



The 2-vertex cover $\{2, 4\}$ of G corresponds to the following two tree pattern transformations that explain all pairs of trees in one step:



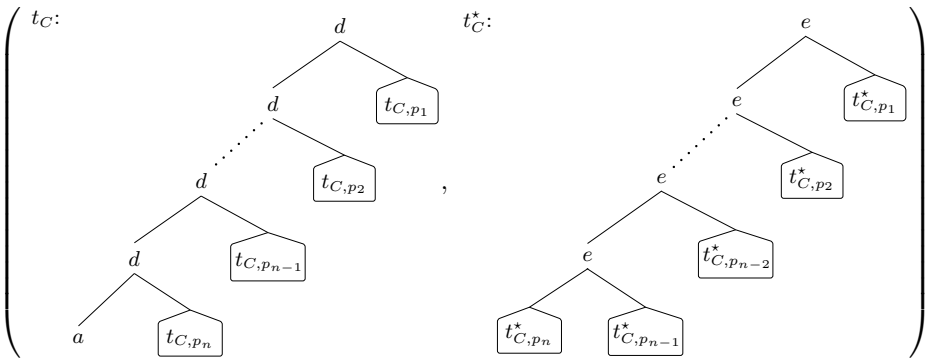
The transformation ρ_2 explains $(t_{1,2}, t^\star_{1,2}), (t_{2,3}, t^\star_{2,3})$ and $(t_{2,4}, t^\star_{2,4})$ in one step, while ρ_4 explains $(t_{1,4}, t^\star_{1,4}), (t_{2,4}, t^\star_{2,4})$ and $(t_{3,4}, t^\star_{3,4})$, also in one step. So the set $\Gamma = \{\rho_2, \rho_4\}$ explains all pairs of examples in one step. ⌟

▶ **Proposition 7.** LearningTreeTransformations is NP-hard, even when all examples are binary trees and when fixing both $s \stackrel{\text{def}}{=} 3$ and $r \stackrel{\text{def}}{=} 2$.

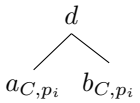
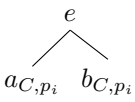
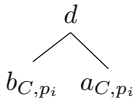
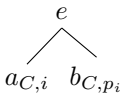
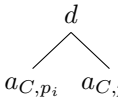
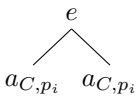
Proof sketch. We reduce from the NP-hard problem 3SAT where, given a propositional formula φ in conjunctive normal form with clauses $\mathcal{C} \stackrel{\text{def}}{=} \{C_1, \ldots, C_m\}$ with at most three literals and variables $P \stackrel{\text{def}}{=} \{p_1, \ldots, p_n\}$, one asks for a satisfying assignment for the variables. We assume, without loss of generality, that each clause has exactly three literals, each variable occurs at most once per clause, and that n is sufficiently large.

Construction. The idea for our reduction f is to construct from a 3SAT instance φ, a LearningTreeTransformations instance with $s \stackrel{\text{def}}{=} 3$, $r \stackrel{\text{def}}{=} 2$ and a set $\mathcal{T} = \mathcal{T}_\mathcal{C} \uplus \mathcal{T}_\mathcal{S}$ of examples. The set $\mathcal{T}_\mathcal{C}$ contains one example pair $(t_C, t_C^\star)$ for each clause C of φ; the set $\mathcal{T}_\mathcal{S}$ contains few additional example pairs that ensure that solution transformations have a certain structure. Intuitively, each satisfying assignment α of φ will correspond to a solution set Γ_α of tree pattern transformations containing one tree transformation ρ_α that *almost* explains all pairs in $\mathcal{T}_\mathcal{C}$ in one step and one additional transformation which helps to *fully* explain these pairs in at most two further steps.

We explain the example pairs in $\mathcal{T}$ next, and start with the pairs in $\mathcal{T}_\mathcal{C}$. For each clause $C \in \mathcal{C}$, the trees t_C and $t_C^\star$ in the example pair $(t_C, t_C^\star)$ have the following shape

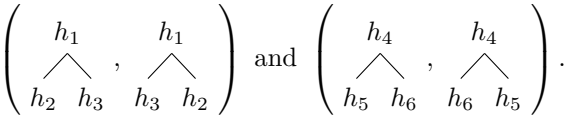
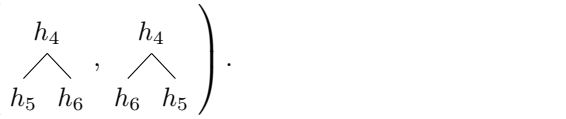


where the subtrees t_{C,p_i} and $t^\star_{C,p_i}$ are called *variable subtrees*. The trees t_{C,p_i} and $t^\star_{C,p_i}$ depend on whether and how the proposition variable p_i occurs in C:

- if p_i occurs positively in C, then $t_{C,p_i} \stackrel{\text{def}}{=}$  and $t^\star_{C,p_i} \stackrel{\text{def}}{=}$ 
- if p_i occurs negatively in C, then $t_{C,p_i} \stackrel{\text{def}}{=}$  and $t^\star_{C,p_i} \stackrel{\text{def}}{=}$ 
- if p_i does not occur in C, then $t_{C,p_i} \stackrel{\text{def}}{=}$  and $t^\star_{C,p_i} \stackrel{\text{def}}{=}$ 

Intuitively, the pairs in $\mathcal{T}_\mathcal{C}$ can *almost* be explained by one tree pattern transformation $\rho\colon \sigma \rightsquigarrow \sigma^\star$ with σ of the same shape as the trees t_C and $\sigma^\star$ of the same shape as $t_C^\star$. Such a transformation indicates in which of the t_{C,p_i} the children need to be swapped, which essentially encodes a satisfying assignment.

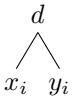
The set $T_\mathcal{S}$ contains the additional examples

 and  .

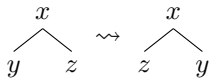
These two examples can be explained by a tree pattern transformation that swaps the leaves; we will see that such a transformation is also required for solutions.

The construction is demonstrated in Example 8. The reduction f is clearly computable in polynomial time.

Correctness. Suppose φ is a satisfiable 3-CNF formula with satisfying assignment α. We construct a solution set Γ_α of transformations for $f(\varphi)$, containing exactly two transformations that explain all examples in at most three steps. The first tree pattern transformation $\rho_\alpha \colon \sigma_\alpha \rightsquigarrow \sigma^\star_\alpha$ in Γ_α encodes the satisfying assignment α: it switches the leaves of the variable subtree for variable p_i if and only if $\alpha(p_i) = 0$. Its body and head are of the same form as the trees t_C and $t^\star_C$ with t_{C,p_i} and $t^\star_{C,p_i}$ replaced by subtrees σ_{C,p_i} and $\sigma^\star_{C,p_i}$ which depend on the value $\alpha(p_i)$ as follows:

- if $\alpha(p_i) = 0$, then $\sigma_{C,p_i} \stackrel{\text{def}}{=}$  and $\sigma^\star_{C,p_i} \stackrel{\text{def}}{=}$ 
- if $\alpha(p_i) = 1$, then $\sigma_{C,p_i} \stackrel{\text{def}}{=}$  and $\sigma^\star_{C,p_i} \stackrel{\text{def}}{=}$ 

The second transformation

$\rho_{\text{swap}}\colon$ 

swaps two siblings.

It is easy to check that $\Gamma_S \stackrel{\text{def}}{=} \{\rho_\alpha, \rho_{\text{swap}}\}$ explains all pairs in $\mathcal{T}$.

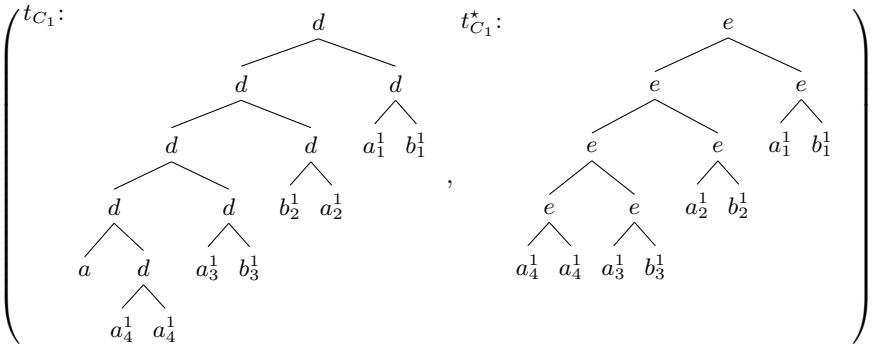
Suppose now that $f(\varphi)$ is a positive instance of LearningTreeTransformations with solution set Γ containing two tree pattern transformations that explain all examples in at most three steps. The idea for obtaining a satisfiable assignment α for φ is to first show that the transformations in Γ are of a very specific form: one of them has to be a transformation like ρ_{swap} that switches two children, and the structure of the other is induced by the shape of the example pairs. We then extract α from the latter transformation. We defer the technical proof and several more details to the long version of this paper [11]. ◀

The following example illustrates the construction from the previous proof.

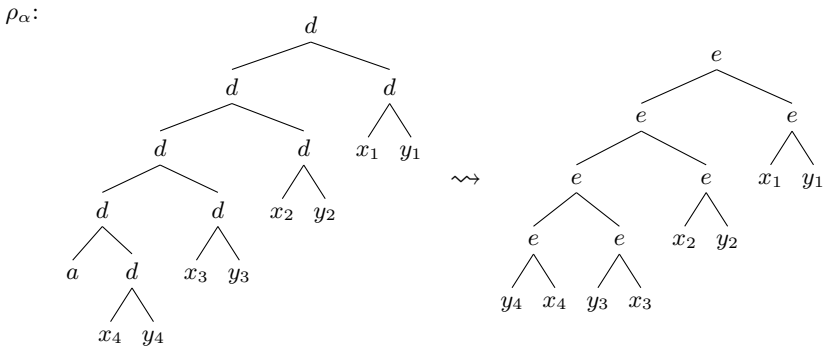
▶ **Example 8.** Consider the positive 3SAT instance

$\varphi = (p_1 \vee \neg p_2 \vee p_3) \wedge (p_2 \vee p_3 \vee p_4) \wedge (\neg p_1 \vee \neg p_3 \vee \neg p_4)$.

Following the construction from the proof of Proposition 7, we construct the pair $(t_{C_1}, t^\star_{C_1})$ for the first clause $C_1 = p_1 \vee \neg p_2 \vee p_3$:



The satisfying assignment α with $\alpha(p_1) = 1$, $\alpha(p_2) = 1$, $\alpha(p_3) = 0$, and $\alpha(p_4) = 0$ corresponds to the tree pattern transformation ρ_α which together with the transformation ρ_{swap} from above explains the examples constructed from φ in at most three steps:



Observe that x_i, y_i are swapped by ρ_α if and only if $\alpha(p_i) = 0$. ⌟

3.2 Learning Tree Pattern Transformations: A Pragmatic Approach

One approach for dealing with the NP-hardness of learning tree pattern transformations for small parameters is to employ the power of SAT solvers. We next sketch how the problem can be encoded into propositional formulas. We defer details and an application to data sets from CS education research to Section A.

We point out that the approach discussed in the following is rather straightforward. Yet, it is also useful: while theory suggests that learning of tree pattern transformations is computationally hard, the SAT solving approach works well enough for our actual data.

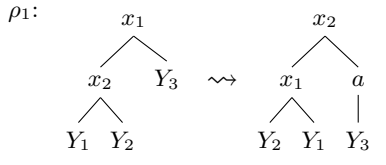
Suppose we are given pairs of trees $\mathcal{D} \stackrel{\text{def}}{=} \{(t_1, t_1^\star), \ldots, (t_n, t_n^\star)\}$ and numbers $s, r \in \mathbb{N}$ and are searching for a set of r transformations that explains each pair in $\mathcal{D}$ in at most s steps. Our encoding closely follows a straightforward non-deterministic algorithm, which proceeds in two steps: first, it guesses (1) a set of r transformations and (2) a sequence of these transformations for each pair $(t_i, t_i^\star)$ and how they are applied; second, it verifies that this information indeed represents a solution.

To encode this as an instance of SAT, we use propositional variables encoding the information (1) and (2). We then construct a propositional formula with the following properties: (a) it is satisfiable if and only if the instance of the tree pattern transformation learning problem it encodes is positive; and (b) from a model of the formula one can construct a set of transformations as well as sequences for each pair certifying that the instance is indeed positive. For simplicity, we start by only considering single-step instances (i.e. instances with $s = 1$). Afterwards, we will outline the additional challenges posed by multiple steps, as well as the adaptions to our encoding that enable us to also support them.

Encoding single-step instances. The basic idea for single-step instances is to encode the syntactical structure of transformations and constrain it in accordance with the example pairs. Our encoding uses three types of propositional variables: $\text{body}^j_{v,\lambda}$, $\text{head}^j_{v,\lambda}$, and $\text{map}^j_{v,i}$. Intuitively, the variables $\text{body}^j_{v,\lambda}$ and $\text{head}^j_{v,\lambda}$ are used to encode the body and head of the j-th transformation, where them being assigned the value 1 means that node v of the body, or head, respectively, of transformation j has label λ. The variables $\text{map}^j_{v,i}$ are used to indicate that transformation j transforms t_i into $t_i^\star$ in such a way that v is the root of a valid

match of the transformation's body into t_i. Example 9 demonstrates the correspondence between the propositional variables and transformations. We use ε to denote the root, 0 for its leftmost child, and so on.

▶ **Example 9.** A tree pattern transformation and the corresponding propositional variables:

ρ_1: 

$\text{body}^1_{\varepsilon,x_1}$		$\text{head}^1_{\varepsilon,x_2}$
body^1_{0,x_2},	body^1_{1,Y_3}	head^1_{0,x_1}, $\text{head}^1_{1,a}$
body^1_{00,Y_1},	body^1_{01,Y_2}	head^1_{00,Y_2}, head^1_{01,Y_1}, head^1_{10,Y_3}

⌟

The formula encoding the input instance is a conjunction of multiple formulas that roughly fall into two categories: (i) formulas ensuring syntactically valid transformations, and (ii), formulas ensuring that the chosen transformations do indeed explain all pairs of trees. We provide two sample formulas and refer to the appendix for more details.

$$\bigwedge_{1\le j\le r}\ \bigwedge_{\lambda\in\mathcal{N}\cup\mathcal{T}}\left(\left(\bigvee_{v\in V}\text{head}^j_{v,\lambda}\right)\rightarrow\left(\bigvee_{v'\in V}\text{body}^j_{v',\lambda}\right)\right) \tag{1}$$

$$\bigwedge_{1\le i\le n}\ \bigwedge_{1\le j\le r}\ \bigwedge_{\text{nodes } v}\left(\text{map}^j_{v,i}\rightarrow\bigwedge_{\substack{\text{non-descendants}\\ v' \text{ of } v}}\psi^{=}_{i,v'}\right) \tag{2}$$

Formula (1) is of category (i) and ensures that a variable λ (ranging over all node variables $\mathcal{N}$ and tree variables $\mathcal{T}$) may only be used in the head of a transformation, if it also occurs in its body. The formulas in category (ii) all "depend" on the previously mentioned $\text{map}^j_{v,i}$ variables. For example, Formula (2) ensures that a transformation may only be applied to a node v of pair i, if the contexts of the i-th source and target tree of this node are isomorphic. We achieve this here by requiring that the labels of t_i and $t_i^\star$ are equal at all "non-descendant positions" v' of v. The verification of equal labels is delegated to another simple subformula $\psi^{=}_{i,v'}$. Additional formulas are used to ensure consistent mappings of labels, node variables and tree variables in both body and head.

Encoding multi-step instances. We now lift the encoding for single-step instances of the learning tree pattern transformations problem to multi-step instances. In contrast to single-step instances, we now need to deal with "intermediate trees", i.e. trees that are neither source nor target tree of a pair but occur when applying a sequence of transformations to a source tree. As we do not know what intermediate trees look like in advance, they are encoded by propositional variables as well. Therefore, in addition to the variables we used before, we now also use variables $\text{int}_{i,k,v,\lambda}$ indicating that node v of the intermediate tree for pair $(t_i, t_i^\star)$ after k steps has label λ. The formulas from the previous section need to be adapted to these new variables. The variables $\text{map}^j_{v,i}$ are replaced by variables $\text{map}^{j,k}_{v,i}$, where $k \le s$ refers to a step. The intended meaning for this variable is that the j-th transformation is applied to node v of the intermediate tree of pair $(t_i, t_i^\star)$ after step $k-1$, where intermediate tree 0 is just t_i. The other parts of the formulas are updated accordingly. For instance, in the single-step case, one formula ensures that a transformation with labels from Σ in its head only explains a pair $(t_i, t_i^\star)$, if the labels in $t_i^\star$ match those in the head. Here, instead of requiring that the label of $t_i^\star$ must be the same as the one in the head at the corresponding position, this requirement would be stated using the new variables for intermediate trees. The relevant subformula would look as follows:

$$\mathrm{map}^{j,k}_{v,i} \to \left(\bigwedge_{w \in \mathbb{N}^* \text{ s.t. } vw \in V} \bigwedge_{\lambda \in \Sigma} \left(\mathrm{head}^{j}_{w,\lambda} \to \mathrm{int}_{i,k,vw,\lambda} \right) \right).$$

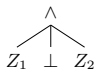
This states that for transformation ρ_j to be applicable to pair i at node v in step k, a label from Σ in the head of transformation ρ_j must be identical to the label at the corresponding position in the intermediate tree of pair i constructed in the previous $k-1$ steps. All the other formulas need to be adapted similarly. Note that the adapted formulas can still be used for the case where $s = 1$, because there is a one-to-one correspondence between $\mathrm{map}^{j}_{v,i}$ and $\mathrm{map}^{j,1}_{v,i}$.

Dealing with noisy data. Data can be noisy if, for example, the dataset contains pairs of trees that differ in such a way that no good explanations can be found for their difference. Therefore, it may not always be possible or desirable to find explanations (i.e. tree pattern transformations) that explain *all* example pairs, but only most of them. To account for this, we have introduced the possibility of specifying a ratio of the number of pairs of trees that must be explainable for the instance to be positive. This does not affect the complexity of the learning problem, but has proved useful in our practical application of identifying mistakes students made when modelling with propositional logic.

4 Extending Tree Pattern Transformations

A tree pattern transformation language for explaining structural differences between trees has to be sufficiently expressive to express typical differences. Yet, it should not be too expressive for two reasons: (1) algorithmic problems such as whether a transformation explains differences betweens two pairs of trees become harder; and (2) learned transformations may overfit the data.

In this section, we discuss one extension of our transformation language which is natural, but too powerful for our application. One shortcoming of our language is that patterns need to exactly adhere to the structure of the (sub)tree they are matching. A potential extension to circumvent this is to include, besides node and tree variables, also *interval variables* which can match contiguous intervals of children of a node. Such variables loosen the strict adherence to the structure while still yielding good explanations of differences by requiring to "capture" everything that can be manipulated.

▶ **Example 10.** Consider the transformation ρ:  with interval variables Z_1 and Z_2. This transformation encodes the equivalence transformation for propositional formulas which replaces conjunctions containing a "false" by a sole "false", e.g. it can be used to transform the syntax trees of $A \wedge \bot \wedge B$ and $A \wedge B \wedge \bot \wedge C$ into $\bot$. ⌟

We formally introduce tree patterns with interval variables before briefly discussing their algorithmic properties and why they are too powerful for learning tree transformations.

Tree pattern with interval variables. A *tree pattern with interval variables* σ is a $\Sigma \uplus \mathcal{N} \uplus \mathcal{T} \uplus \mathcal{I}$-labelled tree, where $\mathcal{T}$- and $\mathcal{I}$-labelled nodes must be leaves. The components Σ, $\mathcal{N}$, and $\mathcal{T}$ are as in the definition of tree patterns. The elements of $\mathcal{I}$ are called *interval variables*. A *match* μ of a tree pattern $\sigma = (V_\sigma, E_\sigma, \ell_\sigma)$ with interval variables in a tree $t = (V_t, E_t, \ell_t)$ is a mapping $\mu \colon V_\sigma \to 2^{V_t}$ from nodes of σ to sets of nodes of t such that (1) the image sets are disjoint, and (2) $\Sigma \uplus \mathcal{N} \uplus \mathcal{T}$-labelled nodes of σ are mapped to singletons. Also, additionally to the conditions on tree pattern matches, we have

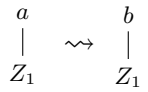
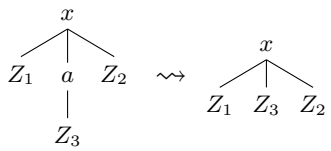
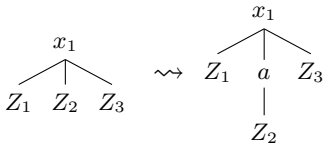
(iv) interval variables I are mapped to (possibly empty) contiguous intervals, i.e. $\mu(I) = \{u_\ell, \ldots, u_{\ell+k-1}\}$ for a contiguous sequence $u_\ell, \ldots, u_{\ell+k-1}$ of children of some node v (called *interval*); and

(v) interval variables are mapped consistently, i.e. if $u, v \in V_\sigma$ are labelled with the same interval variable, then $|\mu(u)| = |\mu(v)|$ and the subtree sequences of $(t_{u_1}, \ldots, t_{u_k})$ and $(t_{v_1}, \ldots, t_{v_k})$ are isomorphic, where u is mapped to the interval $u_1, \ldots, u_k$ and v is mapped to the interval $v_1, \ldots, v_k$ by μ.

Tree pattern transformations with interval variables are defined analogously to tree pattern transformations.

An example use case is to "simulate" the operations considered for tree edit distance:

▶ **Example 11.** Tree pattern transformations with interval variables can easily express the three usual tree edit distances operations relabel, insert and delete [3]:

- Relabelling an a-labelled node to a b-labelled node: 
- Deleting an inner a-labelled node: 
- Inserting an a-labelled node: 

Note that these edits could also be expressed without interval variables. However, a different transformation would be needed for every length of matched child sequence(s). ⌟

Algorithmic properties. Unfortunately, already testing whether a tree pattern transformation with interval variables transforms a tree t into a tree $t^\star$ is NP-hard. This can be seen by reducing from the following NP-hard problem from [1]. A *string pattern* p is a string from $\Sigma \cup \mathcal{X}$ where $\mathcal{X}$ is a set $\{x_1, x_2, \ldots\}$ of variables. The language $L(p)$ of a string pattern contains strings w which are obtained from p by replacing variables "consistently" by strings from Σ^+. Here, consistently means that all occurrences of a variable $x \in X$ are replaced by the same string.

Problem: StringPatternMembership
Input: String $s \in \Sigma^*$, a string pattern p.
Question: Is $s \in L(p)$?

For instance, (s, p) with $s = 001110110$ and $\rho = 0x_1 1x_1 0$ is a positive instance of StringPatternMembership, as x_1 can be replaced by 011 to yield s. The problem StringPatternMembership is NP-complete [1, Theorem 3.6].

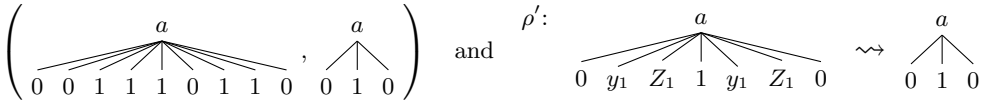
▶ **Proposition 12.** *Testing whether a tree pattern transformation with interval variables transforms a tree t into a tree $t^\star$ is* NP*-complete.*

Proof sketch. Membership in NP is straightforward. To show hardness, we reduce from StringPatternMembership. From a string s and a pattern p, we construct trees t, $t^\star$ and a tree pattern transformation $\rho\colon \sigma \rightsquigarrow \sigma^\star$ as follows:

- t has an a-labelled root with children labelled with the symbols of s;
- $t^\star$ has an a-labelled root with children labelled with the Σ-labels of p;
- σ has an a-labelled root with children like p', where p' is obtained from p by replacing variables x_i by $y_i Z_i$. Here, y_i are node variables and Z_i are interval variables. We need both a node and an interval variable here, because the variables in StringPatternMembership are required to be substituted by non-empty strings.
- $\sigma^\star \stackrel{\text{def}}{=} t^\star$

For an illustration of this construction, we refer to the following Example 13. ◀

▶ **Example 13.** The reduction from the proof sketch above constructs, from an instance (s, p) with $s = 001100110$ and $\rho = 0x_1 1x_1 0$, the following pair of trees and transformation:



⌟

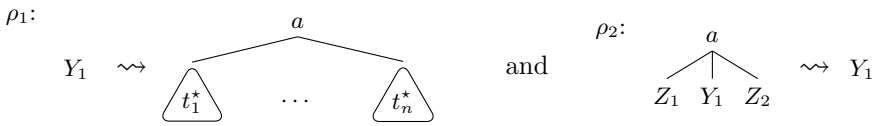
The hardness in the above proof comes from the reuse of interval variables. It is tempting to restrict the reuse of interval variables to circumvent hardness – also because reuse is often not necessary in practical examples (see Examples 10 and 11 above). However, even if each interval variable may only be used once, the problem remains NP-hard.

▶ **Theorem 14.** *Testing whether a tree pattern transformation $\rho\colon \sigma \rightsquigarrow \sigma^\star$ with interval variables transforms a tree t into a tree $t^\star$ is* NP*-hard, even if each interval variable may occur at most once in σ and at most once in $\sigma^\star$.*

The proof of this theorem is rather technical and can be found in the long version.

Interval variables are (too) powerful. Adding interval variables makes tree pattern transformations very powerful. The following example indicates that for using tree pattern transformations for learning from examples, suitable restrictions need to be found. The example shows that, given pairs $\{(t_i, t_i^\star)\}_i$, one can easily find two tree pattern transformations with interval variables that can be used to transform each t_i into $t_i^\star$ in two steps.

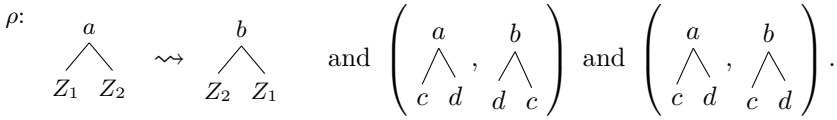
▶ **Example 15.** For a set $\{(t_i, t_i^\star)\}_i$ of pairs of trees, the following two transformations, in which Y_1 is a tree variable and Z_1 and Z_2 are interval variables, explain all pairs:



⌟

Another possible disadvantage of interval variables is that they can lead to a loss of types of differences we want to be able to explain. For example, the same transformation with interval variables can swap as well as not swap the children of a node:

▶ **Example 16.** Consider



Both pairs of trees are explained by ρ in one step. ⌟

While introducing interval variables can be useful, it also has disadvantages. The decision whether to include them in the language when learning differences depends on the differences one might expect to learn.

5 Summary and Perspectives

This paper takes a first step towards understanding how to algorithmically learn explanations for structural differences between trees. A language for specifying tree transformations based on tree patterns was introduced. It was shown that learning such transformations from examples is computationally hard, even for restricted cases. Towards practical application, we proposed an encoding of the problem as a propositional satisfiability problem. We validated the usefulness of the specification language and the effectiveness of the encoding exemplarily on a dataset for educational tasks for logical modelling obtained with the educational support system Iltis.

In the future, we plan to address open theoretical questions as well as explore further applications.

Several theoretical questions remain open regarding properties of (variants of) our tree transformation language. While we saw that already severe restrictions of the learning problem are NP-hard, it is unclear whether the general problem is even in NP. The main issue is that the size of intermediate trees can become exponential in the number of steps, since a transformation with m nodes can introduce $\mathcal{O}(m-1)$ copies of the tree to which it is applied, potentially resulting in trees of size $\mathcal{O}(m^s)$ after s steps. We conjecture that it is never necessary to construct large intermediate trees and that, therefore, the answer to the following question is positive.

▶ **Open problem.** Is LearningTreeTransformations in NP?

All restrictions studied here remain NP-hard. We leave open the complexity of the restriction asking whether there exists a single transformation explaining all given pairs in one step (i.e. the case $s \stackrel{\text{def}}{=} 1$ and $r \stackrel{\text{def}}{=} 1$).

▶ **Open problem.** Is LearningTreeTransformations in PTime when fixing $s \stackrel{\text{def}}{=} 1$ and $r \stackrel{\text{def}}{=} 1$?

As a first step, we can prove that this case is solvable in polynomial time if all transformations must be applied at the root node of a tree (see long version [11]).

We made several design choices, including injective semantics and patterns that must match the shapes of trees very closely. Studying variants where different choices are made may be insightful as well.

▶ **Open problem.** How do slight changes in the definition of tree pattern transformations (e.g. injective vs. non-injective semantics) impact algorithmic properties and expressive power?

Besides addressing the theoretical questions, we also plan to explore further applications, among other things, (1) the (empirical) application of this approach to identify typical mistakes in modelling for other formalisms (including further logics, query languages, etc.), (2) the extension of the approach to other educational tasks, such as the identification of mistakes in equivalence transformation tasks, and (3) similar approaches for educational tasks in other domains, such as the formal language domain. These applications are at the borderline of CS education research and theoretical computer science, in particular database theory, but we expect that they raise further interesting theoretical questions.

References

1 Dana Angluin. Finding patterns common to a set of strings. *Journal of Computer and System Sciences*, 21(1):46–62, 1980. doi:10.1016/0022-0000(80)90041-0.

2 Florent Avellaneda and Alexandre Petrenko. Inferring DFA without negative examples. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 17–29. PMLR, 2018. URL: http://proceedings.mlr.press/v93/avellaneda19a.html.

3 Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1-3):217–239, 2005. doi:10.1016/J.TCS.2004.12.030.

4 Mikolaj Bojanczyk and Amina Doumane. First-order tree-to-tree functions. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 252–265. ACM, 2020. doi:10.1145/3373718.3394785.

5 Sara Cohen and Yaacov Y. Weiss. The complexity of learning tree patterns from example graphs. *ACM Trans. Database Syst.*, 41(2):14:1–14:44, 2016. doi:10.1145/2890492.

6 Wojciech Czerwinski, Wim Martens, Matthias Niewerth, and Pawel Parys. Minimization of tree patterns. *J. ACM*, 65(4):26:1–26:46, 2018. doi:10.1145/3180281.

7 Michael R Garey and David S Johnson. *Computers and intractability*, volume 174. freeman San Francisco, 1979.

8 Aurélien Lemay, Sebastian Maneth, and Joachim Niehren. A learning algorithm for top-down XML transformations. In Jan Paredaens and Dirk Van Gucht, editors, *Proceedings of the Twenty-Ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2010, June 6-11, 2010, Indianapolis, Indiana, USA*, pages 285–296. ACM, 2010. doi:10.1145/1807085.1807122.

9 Aurélien Lemay, Joachim Niehren, and Rémi Gilleron. Learning n-ary node selecting tree transducers from completely annotated examples. In Yasubumi Sakakibara, Satoshi Kobayashi, Kengo Sato, Tetsuro Nishino, and Etsuji Tomita, editors, *Grammatical Inference: Algorithms and Applications, 8th International Colloquium, ICGI 2006, Tokyo, Japan, September 20-22, 2006, Proceedings*, volume 4201 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2006. doi:10.1007/11872436_21.

10 Daniel Neider and Ivan Gavran. Learning linear temporal properties. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–10. IEEE, 2018. doi:10.23919/FMCAD.2018.8603016.

11 Daniel Neider, Leif Sabellek, Johannes Schmidt, Fabian Vehlken, and Thomas Zeume. Learning tree pattern transformations, 2024. doi:10.48550/arXiv.2410.07708.

12 Rajarshi Roy, Dana Fisman, and Daniel Neider. Learning interpretable models in the property specification language. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 2213–2219. ijcai.org, 2020. doi:10.24963/IJCAI.2020/306.

13 Rajarshi Roy, Jean-Raphaël Gaglione, Nasim Baharisangari, Daniel Neider, Zhe Xu, and Ufuk Topcu. Learning interpretable temporal properties from positive examples only. In Brian Williams, Yiling Chen, and Jennifer Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 6507–6515. AAAI Press, 2023. doi:10.1609/AAAI.V37I5.25800.

14 Marko Schmellenkamp, Alexandra Latys, and Thomas Zeume. Discovering and quantifying misconceptions in formal methods using intelligent tutoring systems. In Maureen Doyle, Ben Stephenson, Brian Dorn, Leen-Kiat Soh, and Lina Battestilli, editors, *Proceedings of the 54th ACM Technical Symposium on Computer Science Education, Volume 1, SIGCSE 2023, Toronto, ON, Canada, March 15-18, 2023*, pages 465–471. ACM, 2023. doi:10.1145/3545945.3569806.

15 Marko Schmellenkamp, Fabian Vehlken, and Thomas Zeume. Teaching formal foundations of computer science with Iltis. *To be published in the Educational Column of the Bulletin of EATCS , Preprint:* https://ruhr-uni-bochum.sciebo.de/s/l4JS7d2H3nypWbP, 2024.

16 Thomas Schwentick. Automata for XML - A survey. *J. Comput. Syst. Sci.*, 73(3):289–315, 2007. doi:10.1016/J.JCSS.2006.10.003.

17 Kaizhong Zhang, Richard Statman, and Dennis E. Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992. doi:10.1016/0020-0190(92)90136-J.

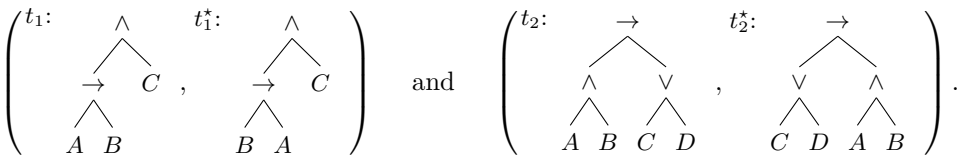
A Learning Tree Pattern Transformations in Practice: SAT Solving

As sketched in the introduction, our interest in learning tree pattern transformations comes from applications in CS education research. For overcoming the NP-hardness results shown in Section 3.1, we follow a common strategy in the literature: we encode the learning problem for tree pattern transformations in propositional logic and employ a SAT solver to find solutions.

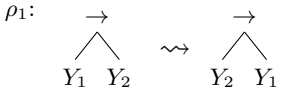
We list all the formulas used in our encoding in Tables 1 and 2 and report on our implementation and its prototypical application. For details we refer to the long version [11].

Let us denote the conjunction of Formulas (3) to (12) as $\Phi^{\mathcal{D}}_{s,r}$. The following example illustrates an application of the approach to two pairs of trees.

▶ **Example 17.** Given the two pairs of trees $\mathcal{D} \stackrel{\text{def}}{=} \{(t_1, t_1^\star), (t_2, t_2^\star)\}$, we want to find a single transformation ρ_1 such that $t_1 \rightsquigarrow_{\rho_1} t_1^\star$ and $t_2 \rightsquigarrow_{\rho_1} t_2^\star$, where



After constructing the formula $\Phi^{\mathcal{D}}_{1,1}$ and feeding it to a SAT solver, we will receive a model which might contain the following, positively assigned variables: $\text{body}^1_{\varepsilon,\to}$, body^1_{0,Y_1}, body^1_{1,Y_2}, $\text{head}^1_{\varepsilon,\to}$, head^1_{0,Y_2}, head^1_{1,Y_1}, $\text{map}^1_{0,1}$, and $\text{map}^1_{\varepsilon,2}$. The body and head variables induce the transformation



and the two map variables indicate the nodes to which ρ_1 can be applied successfully: the left child of the root of t_1, i.e. node 0, and the root of t_2, i.e. node ε. Observe that applying ρ_1 to the subtrees induced by these nodes does indeed result in $t_1^\star$ and $t_2^\star$, respectively. ⌟

We implemented this encoding and applied it to datasets from CS education. Our implementation supports single-step and multi-step instances, an incremental mode, as well as some practical extensions such as providing a ratio of how many examples need to be explained.

Before applying our encoding to real data, we tested that it works as expected by applying it to the LearningTreeTransformations instances from Examples 6 and 8. Encodings of both example sets were solved and the returned models were as expected.

Table 1 Constraints ensuring the syntactic validity of r transformations.

$$\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\bigvee_{\lambda\in\Omega}\mathrm{body}^j_{v,\lambda}\wedge\bigwedge_{\lambda_1,\lambda_2\in\Omega,\,\lambda_1\neq\lambda_2}\left(\neg\mathrm{body}^j_{v,\lambda_1}\vee\neg\mathrm{body}^j_{v,\lambda_2}\right)\right) \tag{3}$$

$$\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\bigvee_{\lambda\in\Omega}\mathrm{head}^j_{v,\lambda}\wedge\bigwedge_{\lambda_1,\lambda_2\in\Omega,\,\lambda_1\neq\lambda_2}\left(\neg\mathrm{head}^j_{v,\lambda_1}\vee\neg\mathrm{head}^j_{v,\lambda_2}\right)\right) \tag{4}$$

$$\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\left(\mathrm{body}^j_{v,-}\to\bigwedge_{w\in\mathbb{N}\,\text{s.t. }vw\in V}\mathrm{body}^j_{vw,-}\right)\wedge\left(\mathrm{head}^j_{v,-}\to\bigwedge_{w\in\mathbb{N}\,\text{s.t. }vw\in V}\mathrm{head}^j_{vw,-}\right)\right) \tag{5}$$

$$\bigwedge_{1\leq j\leq r}\bigwedge_{\lambda\in\mathcal{T}}\bigwedge_{v\in V}\left(\mathrm{body}^j_{v,\lambda}\to\varphi_{\mathrm{leaf}}(v)\right) \tag{6}$$

$$\bigwedge_{1\leq j\leq r}\bigwedge_{\lambda\in\mathcal{T}}\bigwedge_{v\in V}\left(\mathrm{head}^j_{v,\lambda}\to\varphi_{\mathrm{leaf}}(v)\right) \tag{7}$$

$$\bigwedge_{1\leq j\leq r}\bigwedge_{\lambda\in\mathcal{N}\cup\mathcal{T}}\left(\left(\bigvee_{v\in V}\mathrm{head}^j_{v,\lambda}\right)\to\left(\bigvee_{v'\in V}\mathrm{body}^j_{v',\lambda}\right)\right) \tag{8}$$

Table 2 Constraints ensuring correct semantics based on matches induced by the $\mathrm{map}^j_{v,i}$ variables. Here we denote the subtree of a tree t_i rooted at a node v as $\mathrm{subtree}_{t_i}(v)$, rather than $t_{i\,v}$, to avoid confusion with indices.

$$\bigwedge_{1\leq i\leq n}\bigvee_{1\leq j\leq r}\bigvee_{v\in V}\mathrm{map}^j_{v,i} \tag{9}$$

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\mathrm{map}^j_{v,i}\to[\mathrm{context}(t_i,v)\cong\mathrm{context}(t_i^\star,v)]\right) \tag{10}$$

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\mathrm{map}^j_{v,i}\to\varphi^{i,j,v}_{\mathrm{labels}}\wedge\varphi^{i,j,v}_{\mathrm{nodevars}}\wedge\varphi^{i,j,v}_{\mathrm{treevars}}\right) \tag{11}$$

where

$$\varphi^{i,j,v}_{\mathrm{labels}}\overset{\mathrm{def}}{=}\bigwedge_{w\in\mathbb{N}^*\,\text{s.t. }vw\in V}\bigwedge_{\lambda\in\Sigma}\left(\mathrm{body}^j_{w,\lambda}\to[\ell_{t_i}(vw)=\lambda]\right)$$

$$\varphi^{i,j,v}_{\mathrm{nodevars}}\overset{\mathrm{def}}{=}\bigwedge_{\substack{w_1\in\mathbb{N}^*\,\text{s.t. }vw_1\in V,\\ w_2\in\mathbb{N}^*\,\text{s.t. }vw_2\in V}}\bigwedge_{\lambda\in\mathcal{N}}\left(\left(\mathrm{body}^j_{w_1,\lambda}\wedge\mathrm{body}^j_{w_2,\lambda}\right)\to[\ell_{t_i}(vw_1)=\ell_{t_i}(vw_2)]\right)$$

$$\varphi^{i,j,v}_{\mathrm{treevars}}\overset{\mathrm{def}}{=}\bigwedge_{\substack{w_1\in\mathbb{N}^*\,\text{s.t. }vw_1\in V,\\ w_2\in\mathbb{N}^*\,\text{s.t. }vw_2\in V}}\bigwedge_{\lambda\in\mathcal{T}}\left(\left(\mathrm{body}^j_{w_1,\lambda}\wedge\mathrm{body}^j_{w_2,\lambda}\right)\to[\mathrm{subtree}_{t_i}(vw_1)\cong\mathrm{subtree}_{t_i}(vw_2)]\right)$$

$$\bigwedge_{1\leq i\leq n}\bigwedge_{1\leq j\leq r}\bigwedge_{v\in V}\left(\mathrm{map}^j_{v,i}\to\bigwedge_{w\in\mathbb{N}^*\,\text{s.t. }vw\in V}\psi^{i,j,v,w}_{\mathrm{labels}}\wedge\psi^{i,j,v,w}_{\mathrm{nodevars}}\wedge\psi^{i,j,v,w}_{\mathrm{treevars}}\right) \tag{12}$$

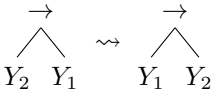
where

$$\psi^{i,j,v,w}_{\mathrm{labels}}\overset{\mathrm{def}}{=}\bigwedge_{\lambda\in\Sigma}\left(\mathrm{head}^j_{w,\lambda}\to[\ell_{t_i^\star}(vw)=\lambda]\right)$$

$$\psi^{i,j,v,w}_{\mathrm{nodevars}}\overset{\mathrm{def}}{=}\bigwedge_{\lambda\in\mathcal{N}}\left(\mathrm{head}^j_{w,\lambda}\to\bigvee_{w'\in\mathbb{N}^*\,\text{s.t. }vw'\in V}\left(\mathrm{body}^j_{w',\lambda}\wedge[\ell_{t_i}(vw')=\ell_{t_i^\star}(vw)]\right)\right)$$

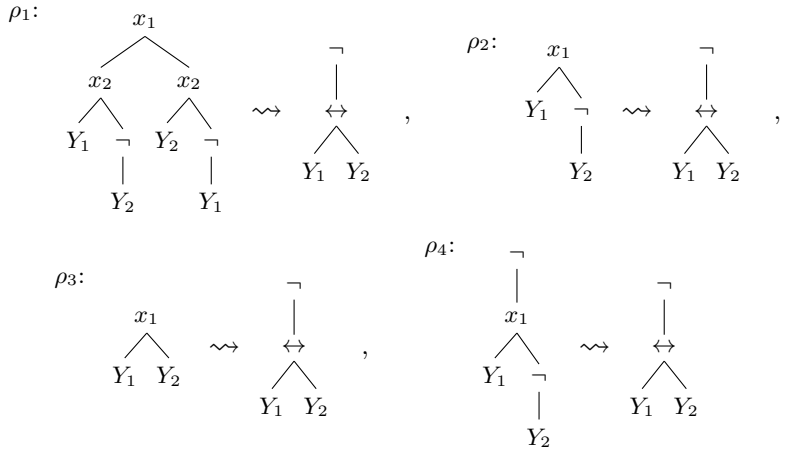
$$\psi^{i,j,v,w}_{\mathrm{treevars}}\overset{\mathrm{def}}{=}\bigwedge_{\lambda\in\mathcal{T}}\left(\mathrm{head}^j_{w,\lambda}\to\bigvee_{w'\in\mathbb{N}^*\,\text{s.t. }vw'\in V}\left(\mathrm{body}^j_{w',\lambda}\wedge[\mathrm{subtree}_{t_i}(vw')\cong\mathrm{subtree}_{t_i^\star}(vw)]\right)\right)$$

We then prototypically tested our framework on datasets from CS education. Many typical mistakes made by students in propositional modelling were identified by Schmellenkamp et al. [14] with a custom approach. Their dataset consists of pairs of formulas $(\varphi, \varphi^\star)$, where the second one is a valid solution while the first is an erroneous modelling attempt. They also provided a clustering of the dataset by typical mistakes that have been identified.

To test the applicability of our framework in this context, we performed two tests. First, we verified that our tree pattern transformations are suitable to express typical mistakes in this context. For example, a typical mistake is to model "only if" statements as if they were "if" statements. In syntax trees of formulas this corresponds to swapping the two children of an implication node. The dataset [14] contains 83 pairs of formulas with this mistake. For instance, it contains pairs like $(E \to (\neg A \wedge \neg C), (\neg A \wedge \neg C) \to E)$ and $(B \to D, D \to B)$. For the LearningTreeTransformations instance with these 83 pairs of (formula) trees and with $s \stackrel{\text{def}}{=} 1$ and $r \stackrel{\text{def}}{=} 1$, the model provided by the SAT solver encodes the expected transformation 

.

Second, we tested whether our framework can be used to automatically identify candidates for common modelling mistakes. To this end, ideally, datasets with pairs of formulas where similar mistakes are expected to happen are used in order to find few tree pattern transformations that explain many of the formula pairs. Such datasets naturally occur when clustering by linguistic operators that occur in the natural language statement that is part of the assignment for students and are provided in the dataset by Schmellenkamp et al. As an example, we consider the operator "either-or" in statements like *I'm travelling to either Italy or Spain this year.* After a preprocessing step of unifying the propositional variables used in all the erroneous formula pairs for this linguistic operator, 19 different pairs remained. The encoding of the LearningTreeTransformations instance containing these 19 pairs with $s \stackrel{\text{def}}{=} 1$ was solved by using an incremental approach for $r = 1, 2, \ldots$. The model provided by the SAT solver encodes the following four transformations:



These transformations correspond to mistakes one would expect. For instance, transformations ρ_2 and ρ_3 explain pairs for which the erroneous formula uses a faulty operator like a conjunction instead of a bi-implication as in $(A \wedge \neg B, \neg(A \leftrightarrow B))$ or $(A \wedge B, \neg(A \leftrightarrow B))$.