Freie Universität Berlin

Masters's Thesis

Syntactic Similarity in human-oriented ATP

Jonas Bayer

First Assessor Prof. Christoph Benzmüller
Second Assessor Prof. Timothy Gowers

August 30, 2023

How to read this thesis

This thesis is directed at the interested mathematician and can be read without previous knowledge in formalization or automated theorem proving. When code is presented, however, prior knowledge of basic concepts in functional programming will be assumed. All source code can be found in the project repository at the link below.

`https://permalink.jonasbayer.de/masters-thesis`

Acknowledgements

I extend my sincere appreciation to my supervisor, Timothy Gowers, for his inspiration, guidance, and encouragement. His expertise and insights have been instrumental in shaping the direction of this research. Moreover, I am deeply thankful to my supervisor, Christoph Benzmüller, for the enlightening discussions on preexisting research related to Automated Theorem Proving from a Classical AI perspective. His constructive feedback and explanations have greatly enriched this work.

I wish to acknowledge Fabian Glöckle for proposing the idea that became the topic of this thesis: to investigate how search can be done using a modification of tree edit distances. I also want to thank Fabian for his suggestions of hole-tree edit actions, our stimulating joint discussions not only on tree editing but also on human-oriented ATP as a whole, and his feedback on this text.

My gratitude extends to William Hart for the captivating discourse on the foundational aspects of theorem proving. Moreover, I am indebted to Jannis Limperg and Anand Rao Tadipatri for generously sharing their expertise and responding to my questions regarding programming in Lean. I would also like to express my thanks to Katie Collins, Anshula Gandhi, Angeliki Koutsoukou-Argyraki, Johannes Lindner, Miroslav Olšák, Fredi Yip, and all the participants in the online team meetings. Their interest in my work, along with their thoughtful pointers and references, have been valuable contributions to the project.

A special mention goes to my parents, family, and friends for their unwavering support and understanding throughout my academic pursuits. Their encouragement has been a constant source of motivation.

Contents

1 Introduction **4**
- 1.1 ATP, an overview . . . 4
- 1.2 A third perspective . . . 4
- 1.3 Researching human-oriented ATP . . . 5
- 1.4 Search and Syntactic Similarity . . . 5
- 1.5 The goal of this thesis . . . 6

2 A theoretical perspective on syntactic similarity **8**
- 2.1 Tree edit distance . . . 8
- 2.2 Tree edit distance for syntax trees . . . 9
- 2.3 Hole expressions . . . 10
- 2.4 Tree editing with holes . . . 10
- 2.5 Defining syntactic distance . . . 12
- 2.6 A comparison with rippling . . . 13

3 Calculating syntactic distance and generalizers **15**
- 3.1 Properties of hole-tree editing . . . 15
  - 3.1.1 Maps induced by editing actions . . . 15
  - 3.1.2 Permuting editing actions . . . 16
- 3.2 Derivation of the Algorithm . . . 16
  - 3.2.1 Base cases . . . 17
  - 3.2.2 Recursive step . . . 17

4 Implementing the recursive algorithm **21**
- 4.1 Trees with holes in Lean . . . 21
- 4.2 Computing syntactic similarity . . . 23
- 4.3 Structuring the computation through auxiliary functions . . . 25
- 4.4 Concluding remarks on the implementation . . . 26

5 A small case study on search **27**
- 5.1 Translating Lean terms to hole trees . . . 27
- 5.2 Similarity-based Search . . . 28
- 5.3 Aesop plus Search . . . 29
- 5.4 Testing on toy examples . . . 29
- 5.5 Towards a faster algorithm . . . 30
  - 5.5.1 Theoretical considerations on the algorithm . . . 30
  - 5.5.2 Implementational aspects . . . 31
- 5.6 A first evaluation . . . 32

6 Conclusion **33**
- 6.1 Development of a concept of syntactic similarity . . . 33
- 6.2 Derivation and implementation of a recursive algorithm . . . 33
- 6.3 Building a testing framework . . . 33
- 6.4 Outlook . . . 34

7 References **35**

1 Introduction

1.1 ATP, an overview

Automated theorem proving (ATP) has long been a significant field in computer science, aiming to develop algorithms for finding formal proofs automatically. With the advent of proof assistants in mathematical research and, more generally, formal methods, these systems will become increasingly relevant for mathematicians, too.

Over the years, researchers have explored various approaches to tackle the inherent challenges of ATP, resulting in two paradigms being heavily researched currently: Satisfiability Modulo Theories (SMT) solving and Machine Learning (ML).

Satisfiability modulo theories is a generalization of the boolean satisfiability problem (SAT). In SMT solving, a given problem is first translated into a particular format which can subsequently be passed to a highly optimized algorithm. These algorithms make use of decision procedures for specific theories and data structures, such as linear arithmetic and lists. By combining the efficient subroutines in a clever way SMT solvers have been able to solve increasingly complex problems [Des+22].

On the other hand, the general trend towards machine learning can also be observed to enter the field of automatic theorem proving. There are multiple ways how ML can be employed in this context, for example, it can improve premise selection [Mik+23], i.e. support other provers by supplying them lists of helpful lemmas to use in proof finding. Another envisaged application of ML is based on auto-formalization [Wu+22]. In that approach, neural networks are used to learn the structure and semantics of conventional mathematical texts, with the goal of translating them into formal logic. Thereby, one can make use of the vast body of mathematical literature and the recent advances with large language models. These examples are only given for illustration, there exist many further ideas how machine learning can be employed in the context of ATP.

1.2 A third perspective

However, despite the advancements made through SMT solving and ML-assisted ATP, there remains an untapped potential in incorporating a human perspective into the theorem proving process. The goal of human-oriented ATP is to develop algorithms whose proving behaviour is as similar as possible to the way that human mathematicians find proofs. Since humans manage to solve challenging mathematical problems one can hope for the success of algorithms centred around leveraging insights from numerous observations on humans problem-solving.

A key property of human-oriented ATP systems is that they should not use brute-force search in an essential way and the constructed proofs should correspond closely to a typical human-written proof. Next to this, an ideal human-oriented ATP will not be domain specific but should in principle be able to find proofs in any given area of mathematics.

However, outlining in precise terms what counts as a human-oriented prover

strongly depends on what criteria one considers for a proof to be typically human. Therefore, a concise definition is not attempted here and the reader instead referred to Timothy Gowers's research manifesto [Gow22]. When used subsequently in this thesis, the term human-oriented ATP is meant in the sense outlined in that document.

Naturally, human-knowledge-based approaches have already been explored and notable contributions to the field have been made, e.g. by Alan Bundy [Bun83]. In recent years, however, the human perspective has received less attention in ATP, as SMT solvers could be observed to generally perform better. These provers have been integrated within proof assistants allowing their users to focus on more high-level aspects of the formalization while the automatic tool can fill in laborious details. Nevertheless, there remain a lot of problems which current SMT solving cannot handle well and it is possible that overcoming these barriers will ultimately require different approaches.

1.3 Researching human-oriented ATP

How is it possible to successfully research human-oriented ATP and filter the best ideas from the myriad of conceivable approaches? Certainly, rapidly prototyping provers plays an important role in finding concepts that enable better automatic proving. At the same time, matters like performance optimization only play a secondary role; such can be done at a later stage when conceptual findings get turned into algorithms meant for end-users in proof assistants.

Instead, it is much more relevant that there is an efficient interface to the prover prototype. The possibility to promptly validate that a computer-generated proof adheres to human line-of-thought is crucial: it can enable the integration of results obtained through rapid prototyping into more theoretical discussions.

Taking all of these matters into consideration suggests using an off-the-shelf proof assistant as a basis for researching human-oriented ATP. Thereby, infrastructural necessities ranging form parsers to the proper display of mathematical notation are readily available, permitting the research to focus on the challenge of finding human-style proofs.

A proof assistant which is particularly suitable for such purposes is Lean 4. It is designed to provide excellent metaprogramming support with Lean being its own metaprogramming language. Next to this, the growing body of mathematical concepts formalized in Mathlib 4 can serve as an extensive testing field.

Of course, these considerations only hold within a certain scope and there might well exist approaches to human-oriented ATP that are more efficiently researched in a custom-built system. In the context of this thesis, however, the capabilities of Lean will be more than sufficient and all implementations will be done in that language.

1.4 Search and Syntactic Similarity

A central challenge that occurs in any kind of theorem proving is that of search: given a proof goal and previous knowledge, e.g. in the form of a library, how does one find the right definition, lemma or theorem that advances the proof?

The importance of this problem is illustrated by the vast amount of previous research on the topic. For example, the Sledgehammer tool of the proof assistant

Isabelle contains a mechanism to select promising candidates for proving among a potentially huge list of lemmas. The original version of that algorithm is symbolic [BDP22] whereas, later, machine learning approaches for premise selection have been investigated [Küh+13]. Very recently, the neural transformer-based Magnushammer [Mik+23] has been demonstrated to outperform the traditional methods.

However, in the context of human-oriented theorem proving the use of such search algorithms would mean to introduce black boxes. Therefore, it is necessary to research human-oriented search, that is find algorithms that mimic the way humans include their previous knowledge in proving. There are various ways human mathematicians do this, for example, based on:

- Syntactic Similarity: the proof goal is similar to a known result as a syntactic expression.
- Meta-level Considerations: there is salient information about how the proof must proceed which can be used to isolate a particular known result. For example, the proof goal could contain a constant on which there is very little previous knowledge. In that case, search can be restricted to that particular subset of previous knowledge.
- Plan-guided Search: assume that the prover follows a planning-based approach, that is it first tries to sketch the searched for proof and then construct it in concise terms. In this case, the information from the sketch can be used when searching previous knowledge for constructing the concise formal proof.

The list is, of course, not exhaustive and many further approaches to human-inspired previous knowledge retrieval are conceivable.

In this thesis, the focus lies on the first aspect, that of syntactic similarity. How can a notion of similarity between syntactic expressions be made precise? The context of search demands a quantitative measure, a metric capable of assigning a similarity value to any pair of syntactic expressions.

However, beyond this quantitative perspective, syntactic similarity encompasses additional dimensions. Human perception often involves assessing the specific aspects in which two expressions are similar. Such qualitative insights can be invaluable for guiding the subsequent reasoning steps of a prover. How can one find a qualitative description of syntactic similarity?

1.5 The goal of this thesis

This work intends to provide first answers to these questions. Firstly, the goal is to define a similarity metric and a corresponding notion of qualitative similarity between terms. Secondly, an algorithm can be devised and implemented that calculates these quantities explicitly. Lastly, the ultimate goal is to set up a testing framework which permits a first, rough evaluation of the algorithm on simple examples in the context of search.

Such a research program is ambitious because of its wide range of tasks including both theoretical and implementational challenges. Each of the three goals mentioned holds the potential to be extensively explored, forming the foundation for an entire master's thesis. However, the deliberate selection of the

three-step plan aims to mitigate the risk of excessive theorizing. Rather than following the traditional approach of developing theory first and implementing it at the end, this program adopts a different strategy. Early on, it involves conducting programming experiments, which, in turn, serve as valuable guides for theory development.

As a result, the theoretical notions, algorithms, and programs developed in this thesis may not be entirely comprehensive or universally applicable to all cases. Instead, the primary objective is to establish a framework that facilitates further research for refining both the theory and programs presented here.

The structure of this text is as follows: Section 2 introduces syntactic similarity as a theoretical concept, aligning with the first step of the research plan. In the subsequent Section 3, an algorithm for computing syntactic similarity is proposed, and its implementation is detailed in Section 4, thereby achieving the second goal. Section 5 further explores the algorithm by defining a simple similarity-based search tactic, allowing evaluation on first examples. This fulfils the three-step research program, which concludes with a summary in Section 6.

2 A theoretical perspective on syntactic similarity

Before starting to think about how terms are similar it is important to have a definition of what a term is. For developing the theory of syntactic similarity we will start by investigating special cases and not yet make the definition precise.

Such an approach could be considered as non-standard since usually ATP systems get developed for a specific logic that is fixed early on. The rationale of the more human-oriented perspective chosen here is that mathematics can be done successfully without explicitly choosing a formal background. Therefore, it is to be hoped that the notion of syntactic similarity developed here can be realized in several logics.

Still, we can put up some general guide-rails as to what a term is: It is clear that the terms we want to compare will contain constants, free variables and function application. Next to this, more complicated terms can contain lambda abstraction and corresponding bound variables. Additionally, we think of all of the constituents to be typed as types usually play an important role in the thought processes of mathematicians and are equally essential in most proof assistants popular today.

2.1 Tree edit distance

Since syntactic expressions naturally have the shape of a tree, a good starting point for thinking about how terms are similar is to think about how trees can be compared. In fact, there is a lot of pre-existing research on so-called tree edit distances [PA] which will be described in the following.

Tree edit distance is a concept that describes how similar two given trees are. More concisely, it is defined as the minimum number of edit actions that are needed to convert the first tree into the second one. The most common actions considered are:

I. Relabelling: Change the label of a node.

II. Insertion: Insert a node between a node in the tree and a subsequent sequence of its children.

III. Deletion: Remove a node n and make its children the direct children of the parent node of n. The order of the children must stay the same.

It is possible to consider a unit cost for all of these actions, more generally, one considers an arbitrary function that assigns a cost value to any edit action. This might not just depend on the type relabelling/insertion/deletion of the action but also on the node that the action gets applied to.

Next to being defined in a very intuitive way, tree edit distance has the advantage that it can be very efficiently calculated making use of dynamic programming techniques. The current state-of-the art is an algorithm called APTED [PA15; PA16] which can efficiently calculate edit distances for various tree shapes.

2.2 Tree edit distance for syntax trees

Given this background, it is natural to ask if tree edit distance could yield a way to measure the similarity of syntax trees. Of course, one could simply run APTED on two given such trees and it would return a value that measures similarity in some way. However, the meaningfulness of such a result heavily depends on the chosen cost function. For example, relabelling of nodes allows for $A \cup B$ to be transformed to $A + B$ in one single action whereas mathematicians would consider the two quite different, the first being a set and the second of a type that allows for addition.

In fact, types not being preserved is the central issue when tree editing syntax trees. When removing the leaf a in the tree from Figure 1 the result is no longer a valid syntax tree.[1] Therefore, neither tree edit action in its original format is

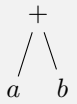
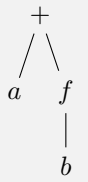


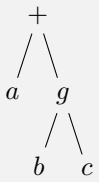
Figure 1: Syntax tree for $a + b$

suited for working with syntax trees in a type-preserving way.

One could overcome this by setting the cost of actions which disregard the well-typedeness to infinity. This would mean that, for instance, relabelling $\cup$ to $+$ has cost infinity whereas relabelling $\cup$ to $\cap$ gets assigned some finite cost. However, the set of actions obtained in such a way would be very limited and not always sufficient to represent structure that humans recognize as similar. As an example for this issue, consider the trees in Figure 2.



(a) Syntax tree for $a + f(b)$



(b) Syntax tree for $a + g(b, c)$

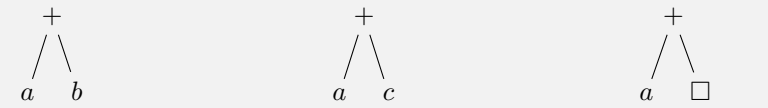
Figure 2: Two syntax trees to be compared

A natural language characterization of the common structure could be "a plus an expression that depends on b" or, perhaps, "a plus the value of a function that takes b as an input". Such a notion of similarity cannot be described accurately with the actions we have seen so far. In fact, it seems that the concept of tree that we have used up to now does not allow to represent similarity as concisely as these two phrases do.

[1] Here we do not allow currying, but even if we did the problem persists: Assume the tree in Figure 1 to be the subtree of a larger syntax tree. Removing a changes its type and thus renders the larger syntax tree invalid.

2.3 Hole expressions

A possible solution to this problem is to allow for holes in expressions. For example, the common characterization of the terms $a + b$ and $a + c$ would then be "a plus hole" (Figure 3). Holes can be instantiated and if one does so with the correct value one either receives the first or second original tree.



(a) $a + b$ (b) $a + c$ (c) "a plus hole"

Figure 3: Syntax trees with holes

In order to allow for the representation of common structure below the level of the tree where a hole needs to be inserted, one can further introduce annotations for holes. That is, also a hole node can have child nodes $c_1, \ldots, c_n$ which means that when the hole node gets instantiated the instantiation should include $c_1, \ldots, c_n$ as terms. Coming back to the example given in Figure 2, this means that the nodes f (or g) get replaced by a hole, but the common dependence on b is indicated through the annotation (Figure 4).

Figure 4: A syntax tree with an annotated hole

Unlike with ordinary nodes we consider the child nodes of a hole node to be unordered. We will also refer to hole nodes as metanodes because of their similarity to how so-called meta-variables get used in proof assistants.

2.4 Tree editing with holes

With this new notion of tree at hand, we can proceed and define the actions that we want to use in hole-tree editing.

Definition 2.1 Hole-Tree edit actions.
We consider the following three edit actions on a tree T:

I. Metanode-Conversion: The action $\mathrm{convert}_n(T)$ consists of converting the ordinary node n (which can also be a leaf) into a metanode. Its children become annotations to the metanode.

II. Metanode-Insertion: Given a node n, the action $\mathrm{insertAbove}_n(T)$ consists of inserting a metanode between n and its direct parent node. If n is the root of the tree the metanode becomes the new root and n its only child node.

III. Annotation-Deletion: If n is the direct child of a metanode then the action $\mathrm{delete}_n(T)$ deletes the subtree starting at n from T.

Let us reconsider the trees from Figure 2. We observe that the actions suffice to arrive at a good characterization of the similar structure; it corresponds to the natural-language description "a plus an expression that depends on b". The editing process is detailed in Figure 5, the number above the arrow indicates which edit action was used.

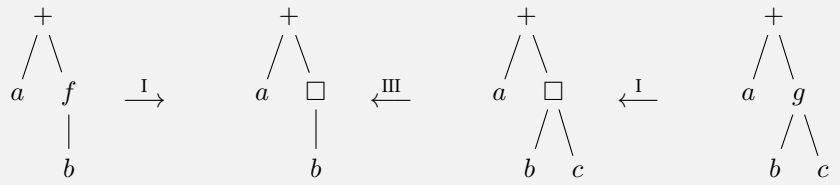
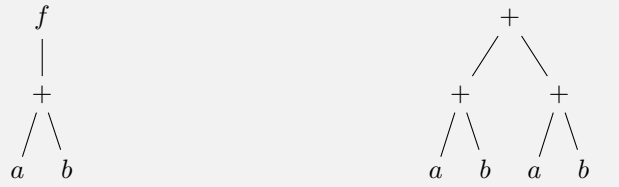


Figure 5: Edit actions applied to the trees from Figure 2. The original trees on the very left and the very right are transformed towards the centre.

Calculating some more simple examples, one can find that these actions usually allow us to arrive at a tree that one would intuitively consider to appropriately describe the similar structure of the trees. This suggests that the three actions defined above suffice for defining a first notion of syntactic similarity.

Note that here we limit ourselves to similarity that can be described in a one-to-one way. When considering Figure 6 we can see that the substructure $a + b$ of the left tree appears twice in the right tree.

It is not possible to represent such a relationship with the hole trees we have defined so far. A possible solution would be to add multiplicity information to (meta)nodes indicating how often that node needs to appear in the left or right tree.



(a) $f(a + b)$ (b) $(a + b) + (a + b)$

Figure 6: One-to-one similarity

Alternatively, one could preprocess or compress syntax trees in a way that reduces multiple occurrence of substructures. For example, the right tree in Figure 6 could be preprocessed to $\mathrm{addToItself}(a + b)$ which then allows unambiguous one-to-one similarity. For this reason, we will not consider multiplicity annotations for now and start defining syntactic similarity in a one-to-one sense.

2.5 Defining syntactic distance

Given the notion of hole-tree and edit action we can proceed to make precise what syntactic similarity should mean. We start with an auxiliary definition that extends the notion of cost from single actions to sequences of actions:

Definition 2.2 Editing sequences and their cost.
Let T_0 be a hole-tree and c a function that associates a positive cost to the edit actions from Definition 2.1. Consider a sequence of actions $a = (a_1, \ldots, a_k)$ such that a_i can be applied to T_{i-1} where $T_i := a_i(T_{i-1})$ for $i = 1, \ldots, k$. We call such a sequence a valid editing sequence and say that it transforms T_0 into T_k. Its cost is defined to be

$$c(a) := \sum_{i=1}^{k} c(a_i).$$

For most cases, we will assign a constant cost of 1 for metanode-conversion or insertion, while the cost of annotation-deletion will correspond to the number of nodes within the subtree that is being deleted. It is important to note that the cost is influenced not only by the type of action but also by the location where it is applied within the tree. For example, one could define a cost function which has cost 2 when the action $\mathrm{convert}$ is applied at the root and unit cost otherwise. Definition 2.2 is designed to accommodate arbitrary cost functions, enabling adjustments to the similarity measure by tweaking the cost in later stages.

Moreover, note that the chosen actions are directional, that is, they transform trees which are more specific into less specific ones. It thus makes sense to say that a common description of two trees T, S should be a tree G that is approached by the actions from both sides. Formally, we define

Definition 2.3 Syntactic generalizer.
Let the hole-trees T, S be given. Assume that there are valid editing sequences a, b that transform T, S into the tree G, respectively. Then we call G a syntactic generalizer of T, S. We define the cost of G to be $c(a) + c(b)$.

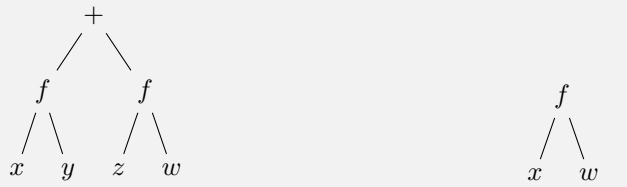
This name is justified since hole-trees can be viewed as generalizing a given syntactic expression. For any hole-tree T, one can consider the set of trees $I(T)$ that can be obtained by instantiating the holes of T.

Recall that the edit actions from Definition 2.1 are information-decreasing: adding metanodes removes information and so does the deletion of metanode annotations. Hence, if $T = a(S)$ then the set $I(T)$ of trees characterized by T is larger than $I(S)$. In other terms, T is a more general description than S.

Using the notion of syntactic generalizer we can define a notion of syntactic distance:

Definition 2.4 Syntactic distance.
Let again two hole-trees T, S be given. We define their syntactic distance to be the minimal natural number d such that there exists a syntactic generalizer of cost d. We also write $d(T, S)$ for this value d and define $G(T, S)$ to be the set of generalizers of T, S that have cost d.

While the distance value is unique by definition, it is not the case that for any two given hole-trees a unique generalizer exists. An example for this can be found in Figure 7, there are two generalizers conceivable depending on whether the similarity in the first or second argument of f is described. It is for this



(a) $f(x, y) + f(z, w)$ (b) $f(x, w)$

Figure 7: Trees that do not have a unique generalizer

reason that the definition refers to a set $G(T, S)$ of generalizers.

Note that a pair (G, d) where G is a generalizer of minimal cost d can be seen to give both a similarity metric between terms and an appertaining common description. It therefore accomplishes the first goal from the three-step program given in the introduction, to make a precise definition of syntactic similarity consisting of both a quantitative and qualitative component. In the following, we will hence refer to such tuples (G, d) as *syntactic similarity*.

2.6 A comparison with rippling

The foundational ideas for this notion of similarity are not new, for instance, holes in terms have been employed in theorem proving for a long time. To demonstrate the novelty of the definition of syntactic similarity given here we make a brief comparison with some of the notions in Alan Bundy's rippling framework [Bun+93]. Here we only give a superficial introduction to rippling, the reader is referred to Bundy's article for its motivation and more in-depth explanations.

The goal of rippling is to automate inductive proofs making use of their specific structure. To that end, hypothesis and conclusion of the inductive step are compared to observe their commonality (called "skeleton") and differences ("wave-front").

To illustrate this, we consider the inductive proof showing that addition of natural numbers is associative. In this proof [Bun+93, p. 189], one assumes that

$$x + (y + z) = (x + y) + z \tag{1}$$

and has to conclude

$$s(x) + (y + z) = (s(x) + y) + z \tag{2}$$

where s is the successor function.

The rippling framework calls expressions which appear in the conclusion but not in the hypothesis a wave-front; in this example the wave-front is $s(\cdots)$. The subterm x which occurs in both hypothesis and conclusion is not considered to be part of the wave-front. In other terms, the wave-front is only the part of a

syntactic expression; in the spirit of the notation above it can be expressed as $s(\Box)$.

This illustrates that rippling and syntactic similarity as described here share a fundamental idea, to consider parts of syntax trees. In Bundy's description of rippling this is done implicitly whereas in this text hole-trees are used to make subtree relationships explicit.

Bundy also considers the common structure between inductive hypothesis and conclusion which he calls skeleton. In the above example, it is $a + (b + c) = (a + b) + c$ where one has $a = x, b = y, c = z$ for the inductive hypothesis and $a = s(x), b = y, c = z$ for the conclusion. This notion appears to be very similar to that of a generalizer. However, skeletons only capture the identical structure occurring at and directly below the root node. For example, the terms $f(a + b)$ and $g(a + b)$ do not have a skeleton because their root node is different. Yet, the non-trivial generalizer $\Box(a + b)$ can describe their syntactic similarity.

In the context of inductive proofs, this limitation of skeletons is not a problem. Usually, inductive rules are such that the syntax trees of inductive hypothesis and conclusion share a root node. However, the example demonstrates that syntactic similarity as presented here is strictly more general for the purpose of describing similarity. In some situations, it can capture commonalities in an accurate way which rippling cannot, the latter having been designed for a different goal. Nevertheless, it is interesting to note that both approaches share fundamental ideas and future research should continue to examine such connections.

3 Calculating syntactic distance and generalizers

Now that we have a concept of syntactic distance and generalizer fixed, it becomes necessary to find a way that allows us to (efficiently) compute these notions. We start by deriving a naive recursive definition from which more efficient ways to calculate syntactic similarity can be developed subsequently. Before delving into the derivation itself, we establish notation and make general observations on hole-tree editing.

Definition 3.1 Notation for hole-trees.
If v is a node of the tree T we write $v \in T$. We denote by $p(v)$ its parent node and by $r(T)$ the root node of the tree T.

Furthermore, we denote an unannotated metanode by $\square$. If it has annotations $a_1, \ldots, a_n$ we write this as $\square(a_1, \ldots, a_n)$.

3.1 Properties of hole-tree editing

3.1.1 Maps induced by editing actions

Let us now start to make observations on hole-tree editing. Note that if the action a is applied to the tree T this induces a map from the nodes of T to the nodes of $a(T)$. More concretely, if $a = \mathrm{insertAbove}_n$ then all nodes of T appear in $a(T)$ and the induced map simply sends every node from T to its counterpart in $a(T)$. Similarly, if $a = \mathrm{convert}_n$ all nodes $m \in T$, $m \neq n$ are mapped to their counterparts in $a(T)$ and n is mapped to the metanode which $\mathrm{convert}_n$ converts it to. For $a = \mathrm{delete}_n$ we establish the convention that all nodes from the subtree getting deleted are mapped to the parent $p(n)$. Thereby, we obtain a well-defined map of nodes T to the nodes of $a(T)$ for any type of action.

Remark that this induced map is surjective onto the nodes of $a(T)$ unless a is of type $\mathrm{insertAbove}_n$. In that case the map is still almost surjective, the only node not being mapped to being the inserted metanode.

We can extend the notion of induced map to editing sequences:

Definition 3.2 Map induced by an editing sequence.
Let T_0 be a tree and $a_1, \ldots, a_k$ be a valid editing sequence when starting with T_0. We set $T_i := a_i(T_{i-1})$ for $i = 1, \ldots, k$ and define α_i to be the map induced by $a_i : T_{i-1} \to T_i$. We can then set $\varphi : T_0 \to T_k$

$$\varphi := \alpha_k \circ \cdots \circ \alpha_1$$

to obtain a map from the nodes of T_0 to those of T_k. We call φ the map induced by the editing sequence.

Informally, this corresponds to following a node n along in the editing process to see where it ends up in the tree T_k. It can still appear as an ordinary node in the final tree T_k or it can be converted into a metanode by some action a_i. It is also possible that a parent of n is transformed into a metanode and that

afterwards n, now an annotation, gets deleted.

We continue to make the following observation on induced maps:

Lemma 3.3 The root of a minimal-cost generalizer is mapped to.
Let $G \in G(T, S)$ be a minimal-cost generalizer of the trees T, S. There exist editing sequences $a_1, \ldots, a_k$ (resp. $b_1, \ldots, b_l$) which transform T (resp. S) into G. Let φ_a, φ_b be their induced maps. Then $\varphi_a(r(T)) = r(G)$ or $\varphi_b(r(S)) = r(G)$.

That is, the root node of G gets mapped to by at least one of the root nodes of T, S.

Proof. Remember that we observed induced maps to be surjective apart from metanode insertion. Therefore, the only possibility for $r(G)$ not to be in the image of either φ_a or φ_b would be for it to be an inserted metanode in both editing sequences. Moreover, this insertion must have happened above the root node of the tree being edited since otherwise $r(G)$ would not be the root of G.

But clearly, this contradicts G being a minimal-cost generalizer. Both editing sequences contain an action that inserts a metanode above the root. One can delete these actions to obtain editing sequences of strictly lower cost. □

It will often also be helpful to make a case distinction depending on where two nodes $n \in T$ and $m \in S$ end up in a generalizer of T, S. We say that n, m match if they are mapped to the same node $g \in G$ by the induced maps. We say that n matches below m if n is mapped to a node that is a (not necessarily direct) child of the node that m gets mapped to. Note that this notion of matching depends on the chosen generalizer G.

3.1.2 Permuting editing actions

We make one last observation before starting the derivation of the syntactic similarity calculation: note that sometimes the order of actions can be permuted.

Assume that there exists a sequence of edit actions transforming T_0 into T_k such that $n \in T_0$ gets converted into a metanode by one of the actions. Then there exists a modification of that editing sequence which starts with $\mathrm{convert}_n$. Note that the edit actions are not permutable in general; it is not possible to execute the action delete_n before the parent of n has been converted into a metanode.

The correct notion of permutability is that the actions $\mathrm{insertAbove}_n$ and $\mathrm{convert}_n$ can be brought to the beginning. If they occur as a_j in a sequence $a = (a_1, \ldots, a_k)$, then the sequence $a' := (a_j, a_1, \ldots, a_{j-1}, a_{j+1} \ldots, a_k)$ is still a valid editing sequence and has the same cost as a.

3.2 Derivation of the Algorithm

With these observations at hand we can start to derive a recursive algorithm that calculates syntactic generalizer and distance. More concisely, we want the algorithm to calculate the (unique) minimal-cost edit distance between two given trees and one of the possibly multiple generalizers. As mentioned before, the set of minimal-cost generalizers for two given trees is not always unique, however, a simple modification to the algorithm presented here allows us to calculate the full set of generalizers.

3.2.1 Base cases

Let us first consider the base cases, i.e. we are given two single-node trees T, S the syntactic similarity of which we need to calculate. This means that if T is an ordinary node then it has no children, and if it is a metanode then it does not have any annotations. There are thus three cases to consider:

I. Both T and S consist of ordinary (leaf) nodes n, m. If the values l_n and l_m of their labels coincide the generalizer will simply be the tree consisting of a single leaf labelled l_n and the syntactic distance is 0 since no action needs to be executed. Otherwise, the generalizer can only be obtained by transforming both leaf nodes into a metanode. Therefore the generalizer will be $\square$, a metanode without annotations, and its cost will be $c(\mathrm{convert}_n) + c(\mathrm{convert}_m)$.

II. Both T and S consist of metanodes. Since they do not have any annotations they are in fact equal, therefore the distance is 0 and the generalizer is an unannotated metanode $\square$.

III. Without loss of generality, T consists of an ordinary node n and S of a metanode m. In order to obtain the generalizer we apply the metanode-conversion action once to T which yields $\square$ as a generalizer and syntactic distance $c(\mathrm{convert}_n)$.

3.2.2 Recursive step

We can now think about how calculating syntactic similarity for more complicated nodes can be broken up into several smaller calculations. We will again need to consider three cases for the given trees T, S depending on their root node: Either both are ordinary nodes, both are meta-nodes or one is an ordinary node and the other one is a metanode.

In the following, it will be helpful to think about which edit actions are essential, i.e. actions which must be executed in order to arrive at a cost-minimal generalizer of T, S. If such an action is a metanode insertion or conversion, by the remark on permutability of actions, it can be executed immediately and the calculation can continue recursively on the trees thus obtained.

Let us start with the case distinction:

I. The given trees T, S have root nodes n, m with labels l_n, l_m and child subtrees $n_1, \ldots, n_k$ and $m_1, \ldots, m_l$, respectively. We perform a case distinction on where n, m can get mapped to in a generalizer G. Because of Lemma 3.3 we know that at least one of n, m must be mapped to the root node of G. Let us thus consider the cases that either both n, m get mapped to the root node or only one of them.

    1. Both n and m get mapped to the root node $r(G)$. We do yet another case distinction on the type of node of $r(G)$.

        a) Assume that $r(G)$ is an ordinary node. This is only possible if the labels l_n and l_m coincide as well as the number of their children. In that case, the children of $r(G)$ are the corresponding generalizers of the children of n, m. That is, we calculate the generalizer g_i of n_i, m_i to obtain the ith child of $r(G)$.

The generalizer of T, S can then be given to be a node with label l_n and children $g_1, \ldots, g_k$. Moreover, the syntactic distance of T, S is the sum of the cost of $g_1, \ldots, g_k$.

b) Assume that $r(G)$ is a metanode. This means that both n, m become converted to metanodes at some point in the tree editing process. By the remark on the permutability of the metanode-conversion action, we can assume that the conversion takes place as the first action.
Accordingly, we define $T' = \mathrm{convert}_n(T)$ and $S' = \mathrm{convert}_m(S)$ and (recursively) calculate a syntactic generalizer and distance (G', d').
Then, G' must also be generalizer of T, S and its cost is

$$d' + c(\mathrm{convert}_n) + c(\mathrm{convert}_m)$$

to account for the two additional metanode conversion actions.

2. Only one of the nodes n, m gets mapped to the root node $r(G)$ of the generalizer. Without loss of generality, let us assume that n gets mapped to $r(G)$ and that m gets mapped to a node below the root $r(G)$. This is only possible if n becomes converted into a metanode and at least one metanode gets inserted above m. Similarly to the previous case, we can think of these actions as being executed immediately which yields the situation depicted in Figure 8.

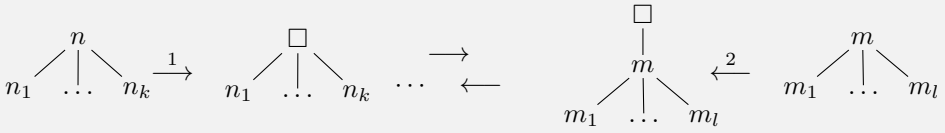


Figure 8: The situation in case I.2

Note that the child subtree of $r(G)$ must therefore be the generalizer of m and one of the $n_1, \ldots, n_k$. Hence, we calculate which of the $n_1, \ldots, n_k$ has the least syntactic distance to m.[2] Assume that n_i is such and that its generalizer with m is g and has cost c_g.
From this, a generalizer and the syntactic distance of S, T can be given explicitly: the generalizer is simply $\square(g)$. The syntactic distance of T, S is the cost of g plus the cost of deleting annotations $n_1, \ldots n_{i-1}, n_{i+1}, \ldots, n_k$ and the cost of the actions being executed in Figure 8:

$$c_g + \sum_{\substack{j=0 \\ j \neq i}}^{k} c(\mathrm{delete}_{n_j}) + c(\mathrm{convert}_n) + c(\mathrm{insertAbove}_m).$$

II. The given trees T, S have root nodes n, m which are metanodes. We let $n_1, \ldots, n_k$ and $m_1, \ldots, m_l$ be their annotations. One can do a case distinction on whether n, m match in a generalizer G or not, i.e. whether they get mapped to the same node by the map induced by the editing sequence.

[2] If there are multiple v_k with minimal syntactic distance to m then there exist multiple generalizers. In this case we can make an arbitrary choice since we set out to calculate one arbitrary generalizer.

1. The nodes n and m match. That is, they get mapped to the same node in G and by Lemma 3.3 it must be the root $r(G)$. Since n, m are both metanodes it follows that $r(G)$ is a metanode.

   In order to give the generalizer explicitly, we need to find the similarity between the (unordered) annotations of n and those of m.

   We are therefore looking for an injective assignment

   $$\sigma : \{1, \dots, k\} \to \{1, \dots, l, \star\}$$

   of the $m_1, \dots, m_l$ to the $n_1, \dots, n_k$ such that the sum of the syntactic distances between $n_i, m_{\sigma(i)}$ is minimal. Here, $\sigma(i) = \star$ means that n_i does not get assigned to any of the $m_1, \dots, m_l$. More precisely, we minimize the cost function

   $$\sum_{\substack{i=1 \\ \sigma(i) \neq \star}}^{k} d(n_i, m_{\sigma(i)}) + \sum_{\substack{i=1 \\ \sigma(i) = \star}}^{k} c(\text{delete}_{n_i}) + \sum_{\substack{i=1 \\ i \notin \text{Im}(\sigma)}}^{l} c(\text{delete}_{m_i})$$

   taking into account the cost for deleting annotations that do not get assigned.

   Determining such a σ of minimal cost is also known as the problem of finding a minimal-cost matching for which an efficient solution exists, known as the Hungarian Method [Kuh55]. This requires that the syntactic distances of all the pairs n_i, m_j are known which can be calculated recursively.[3] Then, one obtains a minimal-cost assignment σ using the Hungarian Algorithm.

   Finally, a generalizer of T, S is explicitly given by $\square(g_1, \dots, g_k)$ where g_i is a generalizer of $n_i, m_{\sigma(i)}$ and $\sigma(i) \neq \star$. The syntactic distance between T, S is the cost of σ.

2. The nodes n and m do not get mapped to the same node in G. One of them must be mapped to the root node $r(G)$. Without loss of generality, assume that n is mapped to $r(G)$ and m is mapped to a node below $r(G)$. Similarly to case I.2, we can recursively calculate the minimal-cost generalizers $g_1, \dots, g_k$ between m and the $n_1, \dots, n_k$. Let g be one of $g_1, \dots, g_k$ which has minimal syntactic distance.

   Then, a generalizer of T, S is given by $\square(g)$. Its cost is the cost of g plus the cost of deleting the remaining annotations of n and the cost of insertAbove_m.

III. The first tree T has the root node n with label l_n and children $n_1, \dots, n_k$. The second tree S has the root node m which is a metanode with annotations $m_1, \dots, m_l$. The root node $r(G)$ of any generalizer of T, S can only be a metanode. We do a case distinction on where it can stem from; Lemma 3.3 ensures that these cases are exhaustive.

   1. The nodes n and m both get mapped to the root $r(G)$. This is only possible if n is converted into a metanode. We can execute this action and let the calculation continue recursively from there.

[3] Note that this creation of many recursive calls constitutes the computational bottleneck of the algorithm. In Section 5, a solution for alleviating this issue will be given.

2. Only the ordinary node n is mapped to the root $r(G)$. This means that m matches in one of the children of n. Thus, we calculate the generalizers between m and $n_1, \ldots, n_k$ and obtain the syntactic similarity between T, S as in case I.2.
3. Only the metanode m is mapped to to the root $r(G)$. In this case, n must match in one of the children of m. The syntactic similarity is obtained as in the case directly before this one.

These are all possible cases that can occur when calculating the syntactic similarity between two given trees. However, when calculating the edit distance one usually does not know yet where the given nodes will end up in the generalizer as is needed to e.g. distinguish between case II.1 and case II.2.

In such cases, the calculation can branch and calculate the syntactic similarity under either assumption. Finally, the minimal result obtained from the branches is returned.

4 Implementing the recursive algorithm

Now that we have derived an algorithm for calculating syntactic distance we can start implementing it in Lean. This and the following section assume the reader to be acquainted with basic concepts in functional programming.

4.1 Trees with holes in Lean

First of all, we declare an inductive data type for trees with metanodes as follows:

```
1 inductive Tree (α : Type) :=
2   | node (label : α) (children : List (Tree α)) : Tree α
3   | metanode (annotations : List (Tree α)) : Tree α
```

This declaration is type-polymorphic which means that the labels of such a tree can be of any type α. The definition states that a tree is either a node or a metanode. In the first case, it consists of a label and a list of child subtrees. In the second case of a metanode, it has a list of annotations which are also of type tree. Note that there is no separate case for leaf nodes which are covered by the list of children being empty.

Using pattern-matching one can then start to define simple functions on such trees, for example, a function that calculates the number of nodes of a given tree.

```
1 def Tree.numberOfNodes : Tree α → Nat
2   | node _ [] => 1
3   | metanode [] => 1
4   | node v (x::xs) => x.numberOfNodes + (node v xs).numberOfNodes
5   | metanode (x::xs) => x.numberOfNodes + (metanode xs).numberOfNodes
```

The first line indicates that `numberOfNodes` takes a tree as an argument and returns a natural number. The subsequent two lines cover the base case of when the root node does not have any children or annotations. Since the function was defined as `Tree.numberOfNodes` the recursive case in lines four and five can make use of dotted identifier notation which allows us to write `x.numberOfNodes` instead of `numberOfNodes x`.[4] The implementation also includes a function calculating the number of nodes in a list of trees which can be defined in a similar way.

In Section 2, we established the convention that syntactic similarity should mean a pair of a minimal generalizer and its associated cost. Accordingly, a named tuple can be defined in Lean:

[4] Note that this function can be implemented in a more efficient way with respect to the number of recursive calls. The trivial implementation is given here for ease of explanation as will be the case for the functions to follow.

```
structure Similarity (α : Type u) where
  generalizer : Tree α
  distance : Nat
```

This simply means that `Similarity` has two fields, the first is the generalizer which is a tree and the second one is the distance which is a natural number. Before turning to the implementation of the algorithm derived in Section 3 we define the following helper functions:

```
def distances (xs : List (Similarity α)) : List Nat :=
  xs.map (fun x => x.distance)

def generalizers (xs : List (Similarity α)) : List (Tree α) :=
  xs.map (fun x => x.generalizer)

def cumulativeDistance (xs : List (Similarity α)) : Nat :=
  (distances xs).foldl (· + ·) 0

instance : HAdd (Similarity α) Nat (Similarity α) where
  hAdd c a := {c with distance := c.distance + a}

def minimalDistanceSimilarityAndIdx :
  List (Similarity α) → Similarity α × Nat
  | [x] => (x, 0)
  | x :: xs =>
  if x.distance == 0 then -- skip tail calculation if x is optimal
    (x, 0)
  else
    let (tail, idx) := minimalDistanceSimilarityAndIdx xs
    if x.distance < tail.distance then
      (x, 0)
    else
      (tail, idx + 1)
  | [] => panic! "Cannot find the minimal distance syntactic
    similarity in empty list"

def minimalDistanceSimilarity
  (xs : List (Similarity α)) : Similarity α :=
  (minimalDistanceSimilarityAndIdx xs).fst
```

The functions `distances` and `generalizers` allow us to quickly access the list of syntactic distances (resp. generalizers) from a list of `Similarity` objects. With this, one can easily define the sum of distances over such a list which is done with `cumulativeDistance` in lines 7-8.

Subsequently, lines 10-11 define an instance of addition for `Similarity` and a natural number. This is done to enable convenient syntax for adding a value to the distance field of `Similarity`. For example, if `s` is of type `Similarity` one can write `s + 1` to obtain a `Similarity` with the same generalizer but distance incremented by one.

Finally, lines 13-23 declare a function which given a list of similarity values returns the first list element that has a minimal distance value. Moreover, it returns the index of that entry in the list which will be useful in the definition of

the syntactic similarity computation function. A version that only returns the `Similarity` object and not the index is defined thereafter.

4.2 Computing syntactic similarity

We can now turn to implementing the recursive algorithm from Section 3. The type signature of the computation function will be as follows:

```
1 partial def compute [BEq α] (tree1 : Tree α) (tree2 : Tree α) :
      SyntacticSimilarity α
```

The `partial` keyword indicates to Lean that it should not try to prove the termination of this function. This is because the termination proof is non-trivial and would have to be supplied manually. Such a proof would be important when trying to formalize properties of the function in Lean. However, since we mainly intend to use it in meta-programming this is not relevant to us.

Moreover, the function takes a typeclass instance `[BEq α]` as an input. This means that there needs to exist a notion of (boolean) equality on the type α. Of course, there is a notion of equality between the symbols in syntactic trees and therefore this requirement will not lead to issues when using the algorithm later.

We can now turn to the core part of the implementation of this function. The following code makes use of auxiliary functions that have not been defined yet but some of which will be explained later.

```
1  match tree1, tree2 with
2  | node v [], node w [] => if v==w then
3                              ⟨node v [], 0⟩
4                            else
5                              ⟨metanode [], 2⟩
6
7  | metanode xs, metanode []
8  | metanode [], metanode xs => let generalizer := metanode []
9                                 let distance := numberOfNodes xs
10                                ⟨generalizer, distance⟩
11
12 | node v xs, node w ys =>
13   if v == w && xs.length == ys.length then
14     similarityOfNodesWithIdenticalLabels v xs ys
15   else
16     similarityOfNodesWithDifferentLabels v xs w ys
17
18 | metanode xs, metanode ys =>
19   let case1 := metaNodesMatchAtRoot xs ys
20   let case2 := metaNodesDontMatch xs ys
21   minimalDistanceSimilarity [case1, case2]
22
23 | node v xs, metanode ys
24 | metanode ys, node v xs =>
25   let matchAtRoot := compute (metanode xs) (metanode ys) + 1
26   let onlyNodeMappedToRoot := matchBelowOneOf (metanode ys) xs + 1
27   let onlyMetanodeMappedToRoot := matchBelowOneOf (node v xs) ys
28   minimalDistanceSimilarity [matchAtRoot, onlyNodeMappedToRoot,
29                              onlyMetanodeMappedToRoot]
```

The first line instructs Lean to pattern match on the two input trees `tree1` and `tree2`. Lines 2-5 then implement the base case where the root nodes are ordinary nodes without children. Recall that if the nodes share a label v then the generalizer is simply a node labelled v and the distance is zero. In Lean, this generalizer is expressed as `node v []` and one can construct a term of type `Similarity` from it by using the angular bracket notation seen in line 3.

The following lines 7-10 implement the base case for metanodes. Note that even if only one of the metanodes is a leaf and the other has some annotations there exists a simple formula to compute their similarity: the generalizer between $\square$ and $\square(x_1, \ldots, x_n)$ will always be a metanode $\square$ without annotations and its distance is the cost of deleting all $x_1, \ldots, x_n$. This property is used here to implement both the base case of two unannotated metanodes and that where only one metanode is unannotated. Effectively, this creates a shortcut in the computation which reduces the number of recursive calls needed.

Next, we proceed to the implementations of the recursive case for two nodes on lines 12-16, for two metanodes on lines 18-21 and for node and metanode in lines 23 to 29. These make heavy use of (hierarchically layered) auxiliary functions which, however, need to depend on the definition of `compute` itself.

4.3 Structuring the computation through auxiliary functions

A solution to define such functions despite their dependency on `compute` is the use of `let`-expressions. We will describe this for one of the many implemented functions in more detail. When defining `matchBelowOneOf` in the code excerpt below, the keyword `let` is used to state its definition within the body of `compute` itself. This enables (recursively) calling the computation function.

```
1  partial def compute [BEq α] (tree1 : Tree α) (tree2 : Tree α) :
2    Similarity α :=
3
4    let matchBelowOneOf
5      (t : Tree α) (xs : List (Tree α)) : Similarity α :=
6      if xs == [] then
7        ⟨metanode [], t.numberOfNodes⟩
8      else
9        let pairwiseSimilarity := xs.map (fun x => compute t x)
10       let (minimizer, minimizerIdx) :=
11         minimalDistanceSimilarityAndIdx pairwiseSimilarity
12       let generalizer := metanode [minimizer.generalizer]
13       let distance := minimizer.distance
14                       + numberOfNodes (xs.eraseIdx minimizerIdx) + 1
15       ⟨generalizer, distance⟩
16
17   ...
18   -- further auxiliary definitions omitted here,
19   -- some also making use of matchBelowOneOf
20
21   match tree1, tree2 with
22     ...
```

The function `matchBelowOneOf` is defined on line 4, its type signature is given on the following line. The recursive call to `compute` can be found on line 9.

This function implements the calculation that needs to be done in case I.2 of the recursive algorithm. In that situation, the node m matches below the node n. Therefore, the syntactic similarities between m and the children of n need to be calculated. Then, an explicit value for generalizer and distance can be determined. The function `matchBelowOneOf` performs this calculation having received the input m as `t` and the children of n as `xs`.

First, a case distinction is made on whether `xs` is the empty list (line 6). In that case, a closed-form expression exists for the syntactic similarity which is returned in line 7.

When `xs` is nonempty, i.e. n does in fact have children, the `Similarity` between `t` and all of the entries of `xs` get calculated (line 9). The following line 10 then determines the syntactic similarity which has the lowest distance value, and its index in the list. This is used to give concrete values for syntactic generalizer and distance as is done on lines 12-14. The expressions in the code correspond precisely to the formulae derived in item I.2 oof Section 3.2. Finally, in line 15 the thus calculated generalizer and distance are returned as a `Similarity` object by means of the angular bracket notation.

Note that a very similar calculation needs to be done in many other cases that

appear in the recursive algorithm, e.g. case II.2. This justifies the introduction of such an auxiliary function hence preventing code duplication.

4.4 Concluding remarks on the implementation

Giving a detailed description of all further auxiliary functions that are used in the computation would be very lengthy; the reader is invited to have a look at the source code which can be found in `ReferenceImplementation.lean` in the project repository. This implementation is in close correspondence with the derivation of the algorithm in Section 3.2.

Suggestive names and the introduction of several layers of functions have been used to achieve a high level of readability of the Lean code. Moreover, the output of the computation function is unit tested on various examples in the file `ReferenceImplementation.lean` that is located in the folder `tests` of the project repository.

With this we conclude the derivation and implementation of an algorithm that calculates syntactic similarity; that is, the second goal of the three-step research program has been accomplished.

5 A small case study on search

The computation function implemented in the previous section allows us to calculate syntactic similarity for any two given hole trees. In order to use it in the context of search it will be necessary to translate Lean terms into the previously defined hole tree format. It will then be possible to use the notion of syntactic similarity for search in Lean metaprogramming. Concretely, a tactic will be implemented by combining syntactic similarity search and Lean's Aesop proof tactic [LF23]. This can then be used to test the notion of syntactic similarity on simple examples; this corresponds to the third step of the three-step research plan set out in the introduction.

5.1 Translating Lean terms to hole trees

A mathematical term in Lean is represented through the `Lean.Expr` datatype. Functions are implemented through currying and repeated application is used for their evaluation. On the other hand, in the syntactic similarity algorithm we thought of functions in their uncurried form. Thus, it becomes necessary to unfold iterated function applications in a Lean expression into a single node when transforming the expression into a hole tree.

Moreover, the theory on syntactic similarity developed so far does not yet cover variables or lambda abstractions. In this first implementation, we will simply replace any variable with a unique symbol. While this certainly does not reflect a good notion of syntactic similarity for terms that contain variables, it is still a reasonable compromise allowing us to test the algorithm in its current form.

Implementing this behaviour is somewhat cumbersome and technical which is why we will focus on the following excerpt of the Lean code:

```
1  def unfoldArguments : Expr → Expr × List Expr
2    | Expr.app f x => let (function, arguments) := unfoldArguments f
3                      (function, arguments ++ [x])
4    | e => (e, [])
5
6  partial def Lean.Expr.toHoleTree : Expr → MetaM (Tree Expr)
7    | Expr.app f x => do
8        let (function, arguments) := unfoldArguments (Expr.app f x)
9        let functionAsTree ← function.toHoleTree
10       let argumentsAsTrees ← arguments.mapM Expr.toHoleTree
11       pure <| Tree.node (marker "app") (functionAsTree ::
       argumentsAsTrees)
12   | ...
```

First, a function `unfoldArguments` is defined which takes a Lean expression and uncurries it: if the expression is of the form `(f x) y` then the result will be `(f, [x, y])`.

This is used in the definition of the translation function `Lean.Expr.toHoleTree`. Given an application `f x`, this function first unfolds possible iterated applications to receive separate expressions for the function and the arguments it is

applied to. Subsequently, the arguments and `f` are recursively turned into trees. Finally, one can return a tree which has the label `app` and `f` and the arguments as child nodes. This is important in case that `f` is not just a function symbol but itself a complex expression.

The attentive reader might notice that the signature of this function is not `Expr → Tree Expr` as one might expect but `Expr → MetaM (Tree Expr)`. The identifier `MetaM` refers to one of the several monads that Lean uses to structure its various features. Concretely, this monad can give access to information about Lean's meta-variables which is helpful for creating the unique symbols that variables get replaced with as described above.

5.2 Similarity-based Search

The problem of search is to select the fact most helpful for proving given a list of definitions and theorems. Using syntactic similarity in this context hence means to calculate the distance between a given proof goal and a list of library statements. Therefore, it is useful to have a function that given a hole tree `tree1` returns which hole tree from a list has the lowest syntactic distance to `tree1`:

```
1 def indexOfMinimalDistanceTree [BEq α]
2   (tree1 : Tree α) (ts : List (Tree α)) : Nat :=
3   let similarities := ts.map (fun tree2 => compute tree1 tree2)
4   let (_, idx) := minimalDistanceSimilarityAndIdx similarities
5   idx
```

Here we again make use of the function `minimalDistanceSimilarityAndIdx` explained earlier. One can then define a function that calculates the best syntactic match between a given Lean expression and a list of lemmas. The latter are referred to via their Lean `Name`:

```
1 def bestSyntacticLibraryMatch
2   (e : Expr) (libraryLemmas : List Name) : MetaM Name := do
3   let goalAsHoleTree ← e.toHoleTree
4   let lemmasAsHoleTrees ← libraryLemmas.mapM
      createHoleTreeFromLemmaName
5   let indexOfBestMatch := indexOfMinimalDistanceTree goalAsHoleTree
      lemmasAsHoleTrees
6   let bestMatch := libraryLemmas[indexOfBestMatch]!
7   pure bestMatch
```

In lines 3-4 the given expression `e` and the library lemmas are converted into hole trees. The function `createHoleTreeFromLemmaName` does what its name says; first obtaining the lemma as a Lean expression and then converting it to the hole tree format.

In Lean, it is possible for one symbol to have multiple meanings, e.g. $A + B$ can denote addition of natural numbers if $A, B \in \mathbb{N}$ but it can also mean adding the number A to all elements of B if $A \in \mathbb{N}$ and $B \subseteq \mathbb{N}$. The function `createHoleTreeFromLemmaName` also includes preprocessing of the given expression to resolve such type class instances.

The hole tree and list of hole trees thus obtained can be passed to the function

calculating the index of the best syntactic match. Line 6 retrieves the name of the lemma that has the lowest syntactic distance to `e`.

With this, we have programmed a function that can immediately be used in meta-programming. Any Lean tactic can now make use of the notion of best syntactic match.

5.3 Aesop plus Search

We can now continue to write a tactic that tests our notion of syntactic similarity. To do so, we use Lean's Aesop tactic which performs a tree (proof) search based on a set of predefined rules. Therefore, Aesop is inherently limited to proving lemmas that are provable with the given rule set.

This restriction can be overcome by extending it to include syntactic similarity-based search: the proof state is compared with lemmas in a library and the most similar one gets added to Aesop's rule set. A suitable tactic is defined in the following code excerpt.

```
1  def libraryLemmaNames := [``Nat.mul_comm, ``Nat.mul_assoc,
       ``Nat.add_comm, ``Nat.add_assoc]
2
3  elab "aesop_with_search" : tactic =>
4    Lean.Elab.Tactic.withMainContext do
5      let goal ← Lean.Elab.Tactic.getMainGoal
6      let type ← goal.getType
7      let reduced ← withTransparency .instances $ reduceAll type
8
9      let result ← bestSyntacticLibraryMatch reduced libraryLemmaNames
10     let resultIdentifier : Ident := mkIdent result
11
12     Elab.Tactic.evalTactic (← `(tactic| aesop (add unsafe $
       resultIdentifier:ident)))
```

First, a toy library consisting of four lemmas is declared; it includes commutativity and associativity of addition and multiplication of natural numbers.

In line 3, a tactic elaboration function for `aesop_with_search` is introduced. This tactic starts by retrieving the current proof goal (lines 5-6). Line 7 reduces typeclasses in the expression, replacing overloaded symbols with unambiguous representations.

This allows us to then correctly retrieve the name of the lemma that is the best syntactic match (line 9). Consequently, Aesop is invoked on line 12 with the lemma added to the Aesop rule set.

5.4 Testing on toy examples

This tactic can directly be tested on some simple examples. Note that Aesop has in-built knowledge about natural numbers which it can use while proving. However, this inbuilt knowledge alone is not sufficient to solve the following examples directly; (pure) Aesop invoked on these does not succeed in finding a proof.

```
example : ∀a b : Nat, a + (b + 0) = b + a := by
  aesop_with_search

example : ∀a b c : Nat, (a + b) + c = a + (b + c) + 0 := by
  aesop_with_search

example : ∀a b : Nat, ((((a + b) ^ 2) ^ 2) ^ 2) = ((((b + a) ^ 2) ^
    2) ^ 2) := by
  aesop_with_search
```

Because of Lean's excellent support for mathematical notation the mathematical content of these examples is self-explanatory. When running this code, Lean does not show any warnings or errors which indicates that the tactic succeeds in proving all examples. That is, Aesop extended with syntactic similarity search manages to outperform pure Aesop; at least with respect to these examples.

For diagnostic purposes, one can modify the elaboration function to also print the result of `bestSyntacticLibraryMatch` to Lean's output pane. Then it is possible to verify that the lemmas which are calculated to be the best syntactic match are indeed the expected ones; in the first and third example this is commutativity of natural numbers and it is associativity in the second one.

5.5 Towards a faster algorithm

The code presented so far in this section accomplishes the third goal set out in the introduction: to connect the similarity calculation to a metaprogramming framework such that testing can be performed. Still, the above examples are very basic and more thorough testing is desirable for a better evaluation of syntactic similarity.

Unfortunately, the implementation of syntactic similarity presented in Section 4 is slow. For example, running it on the third example above takes several minutes to terminate. Therefore, discussing performance of the algorithm and its implementation become vital for the further development of syntactic similarity based on hole-tree editing.

5.5.1 Theoretical considerations on the algorithm

On the theoretical side, it is conceivable that algorithms exist that show a better asymptotic behaviour as trees get large. In fact, (conventional) tree editing being fast through the use of dynamic programming techniques was one of the reasons for its choice as an inspiration for hole-tree editing.

The recursive algorithm given here creates n^2 recursive calls when comparing two metanodes with n annotations. Comparing trees only consisting of metanodes that all have n children thus means that the number of recursive calls is $n^2 \uparrow\uparrow d$ where d is the depth of the trees. Here, the notation $\uparrow\uparrow$ means repeated exponentiation.

This asymptotic behaviour can certainly become problematic, even for small values of d. However, this worst-case consideration does not take into account that, typically, the trees occurring in hole-tree editing mostly consist of ordinary nodes. A detailed assessment would have to take such factors into account.

However, it is already known that tree editing for trees which have unordered nodes cannot be done efficiently unless P = NP [ZJ94]. Since annotations of metanodes are unordered, this also applies to hole-tree editing; it is hence not to be expected that a polynomial-time algorithm for the calculation of hole-tree edit distances exists.

Nevertheless, this asymptotic behaviour does not necessarily render hole-tree editing useless as a basis for syntactic similarity. Since typical syntactic expressions in human proofs do not exceed a certain length it is sufficient for a similarity calculation algorithm to perform reasonably fast on terms of that size. Improvements could potentially be made by enhancing the way syntactic similarity gets calculated when two (ordinary) nodes with ordered are compared. Still, because of the fundamental limitations to what a new algorithm could achieve we will instead focus on improving implementational aspects.

5.5.2 Implementational aspects

A natural improvement that can be made to accelerate the computation function is to introduce caching. The same recursive calls can be made several times within one calculation and if results get cached this can bring first improvements. Moreover, when comparing a given expression e with a library of results $l_1, l_2, \ldots, l_n$ the cache from calculating $d(e, l_1)$ can be reused when calculating $d(e, l_2)$. This will be particularly helpful if l_1 and l_2 contain similar subexpressions which can be expected to occur frequently in a mathematical library.

Next to this, caching can enable the possibility to reorder the sequence of recursive calls in the calculation. The implementation presented so far always calculates all of $d(e, l_1), \ldots, d(e, l_n)$ and these calculations take particularly long if the value of $d(e, l_i)$ is high. However, as soon as $d(e, l_1)$ is known this gives an upper bound for the distance of the best match and thus the calculation of $d(e, l_i)$ should stop if a value below $d(e, l_1)$ cannot be achieved. Next to this, a clever reordering of the recursive calls can also reduce the number of `compute` invocations in case II.1 of the calculation which is that with the worst blow-up in recursive calls.

This motivates reimplementing the recursive algorithm in a way that uses caching. This implementation can be found in the file `SyntacticSimilarity.lean` of the project repository. A hash function for hole-trees is defined and consequently the results of the syntactic similarity calculation can be saved using a hash map. Moreover, a parameter `maximumDistance` is passed to the calculation function which will abort any calls if the resulting syntactic distance has to exceed `maximumDistance`.

These features are implemented in a Lean-idiomatic way through the creation of a custom monad containing both read-only and modifiable state. The read-only variables are used for setting parameters of the computation, for instance, to pass `maximumDistance` to the computation. Moreover, the modifiable state is used for caching.

Such a setup is especially useful for extending the algorithm in future. For example, when introducing the treatment of variables in the similarity calculation it will be necessary to keep track of what values have been assigned to a certain variable. Such information can be stored in easy-to-add fields of the state monad, permitting a high degree of extensibility of the algorithm.

With these changes the previously defined `aesop_with_search` tactic can

be observed to perform much faster. The calculation that took several minutes before now takes place in less than a second. These changes build a basis enabling further research on hole-tree based syntactic similarity in Lean; with the algorithm now running at an acceptable speed the introduction of new features like variables or binders can be investigated.

5.6 A first evaluation

We conclude this section with a small evaluation that demonstrates the capability of the chosen approach. However, it is important to note that this evaluation only allows us to get a first impression and further empirical validation becomes possible and necessary once additional features are introduced.

Moreover, this case study is limited by the capabilities of Aesop which enter in two ways. Firstly, it is possible that the syntactic similarity algorithm finds a useful lemma for proving the given goal but Aesop does not manage to complete the proof even with this information. Secondly, there are a lot of simple problems which Aesop can solve without additional support. It is possible that syntactic similarity can be helpful in such situations but with the testing setup developed here this cannot be evaluated.

The evaluation consists of 30 test statements which can be found in the file `tests/Search.lean`. They include simple lemmas on natural numbers, integers, real numbers, sets, functions and groups. The tactic combing Aesop and search succeeds on 22 examples. In six cases, an appropriate lemma is found by the search algorithm but Aesop does not succeed in applying it.

In the remaining two tests, the search algorithm does not return an appropriate result. In Lean, universe-polymorphic constants in expressions are qualified with a universe meta-variable if their universe level is unknown. Such internal technical representation can obfuscate syntactic similarity unless unification for meta-variables is part of the similarity algorithm. This is the case in the two examples where the search algorithm failed to return an appropriate result. Likely, an improved search algorithm including unification would succeed on these examples.

To conclude, the small evaluation demonstrates the capability of the chosen approach yet before it has been developed to its full potential. It is only to be expected that an enhanced treatment of variables, binders and alike will improve the results of this case study and bring solving more complicated problems in reach. The search algorithm described in this thesis has been optimized to be used in the development of human-oriented automatic provers. Therefore, future research should not only test it as a component but also evaluate its use when employed as part of a human-oriented prover.

6 Conclusion

This thesis has aimed to explore notions of syntactic similarity in the context of search and human-oriented automatic theorem proving. The primary objective was to develop a quantitative metric for measuring the similarity between syntactic expressions, along with a complementary qualitative description. Additionally, an algorithm was devised to explicitly calculate these similarity measures. An implementation in the Lean proof assistant was given which could further be used for a first evaluation of the notion of similarity.

6.1 Development of a concept of syntactic similarity

The notion of syntactic similarity described in this text is especially useful because of its intuitive definition based on tree-edit distances. This allows other researchers to use it in an appropriate way while not having to understand all the complexity of its implementation. Moreover, the possibility of introducing further edit actions and changing the cost function permits to modify the notion of similarity in an intuitive way.

In its form described in this thesis, the concept of syntactic similarity does not include the treatment of variables or binders. Future research should explore how unification and syntax tree editing can be combined in order to enable proper treatment of variables.

Potentially, one could even extend the notion of syntactic similarity considered here to reflect similarity modulo a theory. This could be achieved by adding an action which enables rewriting equalities $A = B$ from a background theory during the tree editing process. Thereby, the distance between two terms that can be transformed into each other in few (equational) reasoning steps can be improved.

6.2 Derivation and implementation of a recursive algorithm

A recursive algorithm for calculating the above notion of syntactic similarity is derived in this thesis. To that end, mathematical properties of the chosen tree editing actions are investigated. This yields an algorithm that calculates both a metric between syntactic expressions and a common description that can be seen to give a qualitative description of syntactic similarity.

This algorithm is implemented using the proof assistant Lean. The implementation of the many necessary case distinctions is structured through several layers of auxiliary functions. Moreover, it is validated through a set of unit tests.

Further research can benefit from the extensible setup chosen and could potentially investigate what improvements to the algorithm can be made on a theoretical level. However, such research needs to carefully estimate the achievable benefits ahead of time to consider the fundamental limitations on the optimal complexity of this algorithm.

6.3 Building a testing framework

Finally, a small testing framework is created that allows us to evaluate syntactic similarity as a means of search in automated theorem proving. For this purpose,

programming infrastructure is created that connects the previously defined similarity calculation function with Lean's metaprogramming framework.

Using the proof-search tactic Aesop, first tests can be performed which demonstrate the potential of the derived notion of syntactic similarity. The computational performance of the calculation which is initially unsatisfactory is investigated both with respect to theoretical and implementational aspects. Eventually, an improved implementation is found to perform better while also being extensible and usable as a basis for further research.

This enables performing a small case-study based on 30 examples of which the implemented tactic can solve 22 immediately. Most of the observed failures are due to limitations of the testing setup used, concretely, the search algorithm finds an appropriate lemma but Aesop cannot readily apply it. Further, it is to be expected that the implementation of additional features as described above will enable the algorithm to also succeed on the unsuccessful tests.

There are various ways to extend on implementational aspects of search. Firstly, syntactic similarity search can be improved through the introduction of a computationally lean preselection mechanism. This mechanism would filter the most promising results from a potentially large library and then pass them on to the more computationally intense syntactic similarity algorithm.

Alternatively, in the spirit of term-indexing [TB06] it is conceivable to store syntactic similarity information about the library ahead of time. This would entail saving expressions in a compressed format such that a faster retrieval of similarity information can be performed. While storing terms in an index is a classical technique doing so in a way that permits one to read of syntactic similarity as defined here is not obvious.

Ultimately, the setup could be adapted such that syntactic similarity based search is more interactively integrated into Aesop. For example, this could include letting Aesop run the search several times and on different proof goals.

6.4 Outlook

Returning to the overarching goal of this thesis, the advancement of human-oriented automatic theorem proving, it will be most interesting to see what role syntactic similarity can play in human-oriented ATP. The author is excited to find out how syntactic similarity can be integrated in the building of provers that mimic humans.

7 References

[BDP22] Jasmin Blanchette, Martin Desharnais, and Lawrence C. Paulson. *Hammering Away. A User's Guide to Sledgehammer for Isabelle/HOL*. 2022. URL: https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf.

[Bun+93] Alan Bundy et al. "Rippling: A heuristic for guiding inductive proofs". In: *Artificial intelligence* 62.2 (1993), pp. 185–253.

[Bun83] Alan Bundy. *The computer modelling of mathematical reasoning*. Vol. 10. Academic Press London, 1983.

[Des+22] Martin Desharnais et al. "Seventeen provers under the hammer". In: *13th International Conference on Interactive Theorem Proving-ITP 2022*. 2022.

[Gow22] Timothy Gowers. *How can it be feasible to find proofs?* Retrieved from https://gowers.wordpress.com/2022/04/28/announcing-an-automatic-theorem-proving-project/. Apr. 2022.

[Küh+13] Daniel Kühlwein et al. "MaSh: machine learning for Sledgehammer". In: *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4*. Springer. 2013, pp. 35–50.

[Kuh55] H. W. Kuhn. "The Hungarian method for the assignment problem". In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. DOI: https://doi.org/10.1002/nav.3800020109. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nav.3800020109. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109.

[LF23] Jannis Limperg and Asta Halkjær From. "Aesop: White-Box Best-First Proof Search for Lean". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2023. Boston, MA, USA: Association for Computing Machinery, 2023, pp. 253–266. ISBN: 9798400700262. DOI: 10.1145/3573105.3575671. URL: https://doi.org/10.1145/3573105.3575671.

[Mik+23] Maciej Mikuła et al. "Magnushammer: A transformer-based approach to premise selection". In: *arXiv preprint arXiv:2303.04488* (2023).

[PA] Mateusz Pawlik and Nikolaus Augsten. *Tree Edit Distance*. http://tree-edit-distance.dbresearch.uni-salzburg.at/. Accessed: 2023-07-29.

[PA15] Mateusz Pawlik and Nikolaus Augsten. "Efficient computation of the tree edit distance". In: *ACM Transactions on Database Systems (TODS)* 40.1 (2015), pp. 1–40.

[PA16] Mateusz Pawlik and Nikolaus Augsten. "Tree edit distance: Robust and memory-efficient". In: *Information Systems* 56 (2016), pp. 157–173.

[TB06] Frank Theiss and Christoph Benzmüller. "Term Indexing for the LEO-II Prover". In: *IWIL-6 workshop at LPAR 2006: The 6th International Workshop on the Implementation of Logics*. Pnom Penh, Cambodia, 2006.

[Wu+22] Yuhuai Wu et al. "Autoformalization for Neural Theorem Proving". In: *Artificial Intelligence and Theorem Proving Conference 2022*. 2022.

[ZJ94] Kaizhong Zhang and Tao Jiang. "Some MAX SNP-hard results concerning unordered labeled trees". In: *Information Processing Letters* 49.5 (1994), pp. 249–254. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(94)90062-0. URL: https://www.sciencedirect.com/science/article/pii/0020019094900620.