Computing All Minimal Corrections for a Word to Match a Context-Free Description

Bachelor Thesis

Student:	Stefan Kablowski
Number:	35384230
First Assessor:	Prof. Dr. Martin Lange
Second Assessor:	Prof. Dr. Gerd Stumme
Supervisor:	Dr. Florian Bruse
Semester:	WT 22/23
Date:	01.12.2022

University of Kassel - Research Group Theoretical Computer Science/Formal Methods

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined or published before. The electronic version of the thesis matches the printed version.

Kassel, December 1, 2022　　　　Stefan Kablowski

Contents

Chapter 1

Introduction

In tuition of natural sciences pupils learn the scientific method. Part of the scientific method is finding a research question, formulating a hypothesis that can be tested and conducting an experiment to actually test the hypothesis. A hypothesis describes how a measured variable influences another measured variable, e.g. *Light influences plant growth.*, and has to refer to the research question. The pupils receive the predefined research question from the teacher and their task is to find relevant hypotheses relating to the given research question. The teacher then assesses the pupils hypotheses on whether they are grammatically correct and fit in the context of the question. When pupils are assigned with such tasks, their learning process can be assisted by introducing interactive systems for learning and teaching (ITL) as described in [3]. ITLs can provide interactive feedback to the pupils. The research group from [9] has developed an ITL for use in biology lessons. Before the devices with the ITL installed are handed out to the pupils, the pupils will be introduced to a biological phenomenon. Then, using the ITL, they put together a hypothesis from a given set of words and clauses. Clicking on a word or clause appends it to the hypothesis. Whenever a pupil is content with its hypothesis, it can request the ITL to check the hypothesis for grammatical correctness and relevance. The ITL then provides feedback whether the hypothesis is syntactically correct. We aim to extend the functionality of the ITL by improving the feedback. The pupil should receive feedback about which parts of its hypothesis are correct and which parts must be changed in order to form a grammatically correct hypothesis. To provide this kind of feedback we need to generate suggestions that represent simple and minimal ways to correct the hypothesis, so the pupil can choose a suggestion which fits its intended hypothesis best. The ITL contains a representation of all valid hypotheses, which can be specified by the teacher in advance.

1.1 Motivation

Speaking from a mathematical standpoint, the ITL checks the given hypothesis against a context-free grammar. A method to algorithmically correct words to match a context-free grammar is given by a minimum distance parser presented in [1]. If a hypothesis is rejected by the ITL, the minimum distance parser can find correct hypotheses with the lowest edit distance to the pupils input. The edit distance indicates the lowest number of edits required to transform one word into another word and vice versa. An edit is the deletion, insertion or replacement of a symbol. This distance measure is also referred to as Levenshtein distance first presented in [12]. However, we demonstrate that the lowest number of edits may not be the best metric to decide whether a suggestion represents a minimal way of editing a malformed hypothesis. If a student enters the hypothesis

Light influences plant (1)

we can argue that a minimal way of fixing (1) would be to insert the missing word *growth*, which leads us to the hypothesis:

Light influences plant growth. (2)

Let us assume that the pupil tries to form the following hypothesis containing a subordinate clause:

Light influences plant growth, but. (3)

We observe that (3) and (2) are an edit distance of 1 apart, while the correct hypothesis with the full subordinate clause is an edit distance of 7 apart from (3):

Light influences plant growth, but only between 500 nm and 600 nm. (4)

From a didactic perspective (4) is considered to be a good hypothesis, because it is more complex and specific than (2). If we provided all suggestions leading to hypotheses with the lowest edit distance – 1 in this case – to the pupil, we would promote the inferior hypothesis and hide away the superior hypothesis. What makes (4) minimal is the fact that no prefix of the edit sequence that turns (3) into (4) produces a correct hypothesis. This idea of minimality is what we discuss in this thesis. A suggestion should be seen as minimal, if the last edit is the first edit that produces a correct hypothesis.

To further illustrate this idea, consider another example:

Humidity influences plant growth. (5)

When correcting (1), there are two obvious ways of reaching (5). One option is first replacing *Light* against *Humidity* and then inserting *growth*. Another option is to first insert *growth* and then replace *Light* against *Humidity*. According to the notion of minimality we propose, replacing *Light* against *Humidity* and then inserting *growth* should be considered minimal, because the intermediate hypothesis is not correct:

Humidity influences plant. (6)

The aim of this thesis is to present and discuss an alternative notion of minimality which is based on the idea that the sequence of edits correcting a hypothesis does not produce a correct hypothesis before the last edit. We prove that there are finitely many minimal hypotheses according to this notion and present an algorithm to compute these hypotheses.

1.2 Structure of Thesis

We begin by laying down the definition of words and languages in Chapter 2. Thereafter, we introduce regular and context-free languages and their representations. These are, formal grammars, regular expressions and finite automata. In addition we outline how commonly known computational problems such as the word problem and emptiness problem are solved for these types of languages. We then define scattered subwords and Higman's Lemma, which we later utilize to prove the finiteness of the set of minimal corrections. In Chapter 3, we formally define the types and semantics of edit operations that can be applied to a word. We then combine these edit operations into edit sequences, called corrections, and describe how corrections can be reordered and simplified according to a given set of rules. Next up, in Chapter 4 a two-level definition of minimality on edit sequences is presented. We present how minimality behaves for scattered subwords and define a normal form for corrections. After the groundwork is laid out, we prove that for an arbitrary hypothesis and a finite representation of all correct hypotheses there exist only finitely many minimal corrections. From the proof we derive an algorithm to efficiently compute these corrections and discuss the runtime complexity and optimizations of the algorithm in Chapter 5. We close in Chapter 6 with concluding remarks on how suitable the presented algorithm is for deployment in an ITL and what must be improved to enhance the quality of the computed corrections.

Chapter 2

Preliminaries

The definitions in this chapter are mostly taken from the lecture slides of Formal Languages and Logic held by Martin Lange in 2019 [10], [6] and [15].

2.1 Words and Languages

An alphabet Σ is a finite, non-empty set of symbols. A word is a sequence of symbols $w_0 \cdots w_{n-1}$ of length n, where $w_0, \ldots, w_{n-1} \in \Sigma$. The empty word ϵ is the only word where length $n = 0$. With Σ^* we denote the set of all words over the alphabet Σ. The set of all words of length $|w| = n$ is denoted as Σ^n. A language $L \subseteq \Sigma^*$ is a set of words over the alphabet Σ^*. A language class $\mathcal{C} \subseteq 2^{\Sigma^*}$ over Σ is a set of languages.

Concatenation. Let u, v be two words with $u, v \in \Sigma^*$. We define the concatenation of $u = u_0 \cdots u_{n-1}$ and $v = u_0 \cdots u_{m-1}$ via

$$(u \cdot v) := u_0 \cdots u_{n-1} v_0 \cdots v_{m-1}.$$

We abbreviate $u \cdot v$ with uv.

Closure Properties. Let $k \geq 0$ and $f : \underbrace{2^{\Sigma^*} \times \cdots \times 2^{\Sigma^*}}_{k \text{ times}} \to 2^{\Sigma^*}$. A language class $\mathcal{C}$ is closed under f, if $f(L_1, \ldots, L_k) \in \mathcal{C}$ is true for all $L_1, \ldots, L_k \in \mathcal{C}$.

Grammars. A grammar $G = (N, \Sigma, P, S)$ is a 4-tuple where N is a finite set of nonterminal symbols, Σ is an alphabet whose symbols are also called terminals, $P \subseteq (N \cup \Sigma)^+ \times (N \cup \Sigma)^*$ is a set of derivation rules and $S \in N$ is the initial nonterminal. Terminals and nonterminals may not coincide, i.e. $N \cap \Sigma = \emptyset$.

Beginning with the start symbol S we can apply derivation rules to produce sentential forms $s \in (N \cup \Sigma)^*$ and finally words $w \in \Sigma^*$ of the language induced by the grammar G. We define the relation $\Rightarrow_G$ for deriving one step, i.e. applying a single derivation rule to a sentential form. Let $u, v \in (N \cup \Sigma)^*$. Then $u \Rightarrow_G v$ if u and v are of the form $u = xyz$ and $v = xy'z$ such that $x, z \in (N \cup \Sigma)^*$ and $y \to y' \in P$. We say that G derives u into v in one step. Let $\Rightarrow_G^*$ be the reflexive and transitive closure of $\Rightarrow_G$. The language $L(G)$ induced by the grammar G, namely $L(G) = \{ w \in \Sigma^* \mid S \Rightarrow_G^* w \}$, is the set of all words that can be derived in any number of steps beginning at the initial nonterminal. A sequence of sentential forms $(s_0, s_1, \ldots, s_n)$ such that $s_0 = S$, $s_n \in \Sigma^*$ and $s_0 \Rightarrow_G s_1 \Rightarrow_G \ldots \Rightarrow_G s_n$ is a derivation of s_n. In general there may exist multiple derivations for one word, introducing ambiguity into the grammar.

Computational Problems for Languages. Computing whether $w \in L$ if $L \subseteq \Sigma^*$ is a language and $w \in \Sigma^*$ is a word is commonly known as the word problem. The emptiness problem is defined as follows. Let $L \subseteq \Sigma^*$. Compute, whether $L = \emptyset$.

2.2 Regular Languages

Regular languages are languages that can be described by a regular grammar or a regular expression, or that can be recognized by a deterministic or nondeterministic automaton. These formalisms are equally descriptive in the sense that they formalize the same set of languages.

Regular Grammars. A regular grammar G is a grammar where every derivation rule $w_1 \rightarrow w_2 \in P$ satisfies $w_1 \in N$ and $w_2 = \Sigma \cup \Sigma N \cup \{\epsilon\}$, meaning w_2 must consist of a terminal and an optional nonterminal or the empty word. We call a language $L(G)$ induced by a regular grammar G a regular language. The language class of regular languages is denoted as REG. Regular languages can also be described by regular expressions.

Regular Expressions. We use regular expressions to describe regular languages. Firstly, we define the syntax of well-formed regular expressions. Secondly, we define the semantics of such regular expressions. The set of regular expressions over an alphabet Σ is the smallest set *RegEx*, for which the following is true:

- $\emptyset \in \mathit{RegEx}$
- $\Sigma \subseteq \mathit{RegEx}$
- if $\alpha, \beta \in \mathit{RegEx}$, then $\alpha \cup \beta, \alpha \cdot \beta, \alpha^* \in \mathit{RegEx}$

We proceed by introducing the semantics of concatenation and the Kleene closure on languages. Concatenation of two languages L_1, L_2 is defined as $L_1 \cdot L_2 := \{ xy \mid x \in L_1 \text{ and } y \in L_2 \}$, abbreviated with $L_1 L_2$. The Kleene closure L^* of a language $L \subseteq \Sigma^*$ is defined as $L^* := \bigcup_{n \geq 0} L^n$ where $L^0 := \{\epsilon\}$ and $L^{i+1} := LL^i$. The language $L(\alpha)$ of a regular expression α is defined by induction over the structure of the expression via

- $L(\emptyset) := \emptyset$
- $L(\alpha \cup \beta) := L(\alpha) \cup L(\beta)$
- $L(\alpha^*) := L(\alpha)^*$
- $L(a) := \{a\}, a \in \Sigma$

- $L(\alpha \cdot \beta) := L(\alpha)L(\beta)$.

Every language that can be described by a regular expression is a regular language. We abbreviate $L(\alpha)$ with α. The set of languages that can be described by a regular expression is exactly the set of regular languages. For any given regular expression we can construct an NFA describing the same language as explained in [15] and vice versa.

Finite Automata. We define deterministic and nondeterministic finite automata to describe regular languages. A finite automaton operates by reading an input word symbol-wise from left to right. It has a set of states, one of which is an initial state. It also has a designated set of final states. The set of states that it can transition into is defined by the transition rules given in the form of a function, in case of a deterministic finite automaton, or a relation, in case of a nondeterministic finite automaton. Which of these transitions are valid in the current situation depends on the symbol from the input word that is currently read and the state. While a nondeterministic finite automaton possibly has multiple successor states per pair of a symbol and a state, a deterministic finite automaton has exactly one. A finite automaton accepts a word if and only if there exists a path from an initial state to a final state labeled with the input word. An automaton can be visualized as a directed graph where the transition rules correspond to the edges and the vertices correspond to the set of states. An edge connects two vertices, if the state of the destination vertex is a valid successor state of the source vertex state according to the transition rules of the automaton. The edge is labeled with all symbols that can be read to transition from the source vertex state to the destination vertex state. The initial state is labeled with an arrow and the final states are labeled with a double outline stroke.

A nondeterministic finite automaton (NFA) $A = (Q, \Sigma, \delta, I, F)$ is a 5-tuple where Q is a set of states, the input alphabet Σ is an alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $I \subseteq Q$ is a set of initial states and $F \subseteq Q$ is a set of final states. To determine whether an NFA accepts a word we introduce the concept of a run λ. A run λ of an automaton A on a word $w = a_0, \ldots, a_{n-1} \in \Sigma^*$ is a (finite) sequence of alternating states and alphabet symbols of the form

$$\lambda = q_0, a_0, q_1, \ldots, q_{n-1}, a_{n-1}, q_n \in Q(\Sigma Q)^*$$

such that $(q_i, a_i, q_{i+1}) \in \delta$ for all $i \in \{0, \ldots, n-1\}$. Runs use by definition only valid transitions from the transition relation δ. A run $\lambda = q_0, \ldots, q_n$ is called an accepting run of automaton A if $q_0 \in I$ and $q_n \in F$. The language L recognized by an NFA is denoted as $L(A)$ and defined via

$$L(A) := \{\, w \in \Sigma^* \mid \text{there exists an accepting run of } A \text{ on } w \,\}.$$

Note that in general for one word there may exist multiple runs on the same NFA.

A deterministic finite automaton (DFA) $A = (Q, \Sigma, \delta, q_0, F)$ is a 5-tuple where Q is the set of states, the input alphabet Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the initial state and $F \subseteq Q$ is the set of final states. A DFA is similar to an NFA, with the difference that δ is a function instead of a relation and q_0 is a single initial state instead of a set of initial states I. We extend the transition function δ to operate on words rather than symbols. The transition function for words $\hat{\delta} : Q \times \Sigma^* \to Q$ is defined inductively over the input word $w = aw'$ as $\hat{\delta}(q, aw') := \hat{\delta}(\delta(q, a), w')$ where $w \in \Sigma^*$, $a \in \Sigma$ and $q \in Q$. When reading the empty word ϵ, the automaton will remain in the current state, i.e. $\hat{\delta}(q, \epsilon) = q$. The accepted language L of the automaton A is defined as $L(A) := \{ w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F \}$, meaning all words that can be read by the automaton using valid transitions, beginning in the initial state and resulting in the final state.

As shown in [15, Section 1.2.3] for any DFA A we can construct a regular language L described by a regular grammar G such that $L(A) = L$, and thus every language recognized by a DFA is a regular language. The word problem for REG can be solved by conducting a constrained reachability search on the graph of a finite automaton A to determine the acceptance of A on the word w. A finite automaton describes the empty language $L = \emptyset$ if and only if there exists no path from the initial state to a final state. The emptiness problem for REG is solved by a breadth-first search on the graph of a finite automaton beginning at the initial state q_0, terminating with *true* if and only if a final state $q_F \in F$ can be reached. As shown in [7] the emptiness problem for finite automata can be solved in polynomial time.

2.3 Context-Free Languages

A context-free grammar G is a grammar where $P \subseteq N \times (\Sigma \cup N)^*$. We denote the language L generated by the context-free grammar G as $L(G)$. Every regular language is also context-free, since every regular grammar satisfies the constraints of a context-free grammar.

The class of context-free languages CFL is closed under the operations $\cup, \cdot, \cdot^*$, and homomorphisms. Context-free languages are closed under intersection with regular languages, meaning $L_1 \cap L_2 \in$ CFL if $L_1 \in$ CFL and $L_2 \in$ REG. The construction of $L_1 \cap L_2$ is shown in [14] and can be solved in polynomial time [16].

The word problem for CFL is solvable in polynomial time, namely $\mathcal{O}(|G| \cdot n^3)$ where n is the length of the input word, with the CYK algorithm explained in [11] or by using the Earley algorithm proposed in [2].

For a context-free grammar G we find the solution to the emptiness problem by checking whether at least one word can be produced starting from the initial symbol S. We consider those symbols of G to be productive that are terminals or fully derive to productive symbols. Hence, whether $L(G)$ is empty can be solved iteratively by computing the least fixed point of productive symbols and checking whether it contains S. Let X denote the set of productive symbols that have been found so far. We initialize $X := \Sigma$ to start with the terminals. Thereafter, we find nonterminals that derive to symbols in X. That means $X = X \cup \{ A \mid A \Rightarrow_G x \text{ and } x \subseteq X \}$. To find the least fixed point we successively compute X until $|X|$ remains constant for two iterations. Then, $L(G)$ is empty if $S \notin X$. This procedure is presented in [5, Section 7.1.1] and runs in $\mathcal{O}(n^2)$ time where $n = |G|$. In [5, Section 7.4.3] there is presented an even more efficient algorithm with a runtime complexity of $\mathcal{O}(n)$.

2.4 Scattered Subwords

Let $y \in \Sigma^*$ be a word. A scattered subword (also known as subsequence) x of y is obtained by leaving out any number of symbols in the initial word y. Word $x = x_0 \dots x_{n-1}$ is a scattered subword of word y, denoted as $x \leq y$, if y has the form $y = y_0 x_0 y_1 x_1 \dots x_{n-1} y_n$, where $y_i \in \Sigma^*$ and $x_i \in \Sigma$. Every word is a scattered subword of itself. For a pair of a word y and a scattered subword x with different lengths, $|x| < |y|$, we write $x < y$ to denote the strict subsequence relation.

Example 1. Let $x = raspy$ and $y = raspberry$, then x is a scattered subword of y, written $raspy \leq raspberry$.

Lemma 2 (Higman's Lemma [4])**.** *Let Σ be an alphabet and Y be an infinite set of words over Σ. Then, there exists a finite subset $X \subseteq Y$, which contains a scattered subword x for any given word $y \in Y$. Consequently, if $w_0, w_1, \dots$ is an infinite sequence of words, two words w_i, w_j with $i < j$ exist such that $w_i \leq w_j$.*

Thus, if a set S is a set of words over Σ and S does not contain two words x, y such that one is a scattered subword of the other, i.e. $x \leq y$, S must be finite, namely $|S| < \infty$. We will utilize this in the proof of Theorem 19 to prove, that there are finitely many minimal corrections.

Chapter 3

Corrections

In this chapter we introduce the syntax and semantics of the operations we can use to change a word, namely insertions, deletions and replacements. They represent simple instructions that specify how words can be transformed into different words in an atomic step. Then, we chain these instructions together into sequences of edit operations we call corrections. We proceed by describing how these corrections can be simplified and which corrections we consider to be equivalent. We argue why semantic equivalence is not sufficient for our endeavor and introduce the concept of similarity to describe how corrections can be reordered while preserving their effect on all words.

3.1 Edit Operations and Corrections

To start off we formally define the atomic operations that we can apply to a word w. For instance, if we want to turn the word a into ab, we need to insert the symbol b at the second position. Conversely, if we want to turn the word ab into a, we need to delete the symbol at the second position. To turn ab into aa, we would replace b at the second position against an a. We continue by defining the syntax for these kinds of simple instructions.

Definition 3. An edit operation τ is one of $a\uparrow_i$, $b\downarrow_i$ or $b \updownarrow_i a$. These are insertion, deletion and replacement, respectively. The symbols a and b with $a, b \in \Sigma$ denote, which symbol will be inserted and which symbol will be deleted or both in the case of a replacement. The index $i \in \mathbb{N}$ denotes at which location in the word the operation is executed, i.e. at which index a is inserted or b is deleted. Each edit operation τ induces a partial function $f_\tau : \Sigma^* \to \Sigma^*$. When an edit operation τ is applied to a word $w = w_0 \dots w_{n-1}$, the result $\tau(w)$ is a new word w', defined via

$$a{\uparrow_i}\,(w) := \begin{cases} w_0 \cdots w_{i-1} a w_i \cdots w_{n-1} & \text{, if } i \leq n \\ \perp & \text{, otherwise,} \end{cases}$$

$$b{\downarrow_i}\,(w) := \begin{cases} w_0 \cdots w_{i-1} w_{i+1} \cdots w_{n-1} & \text{, if } i < n \text{ and } w_i = b \\ \perp & \text{, otherwise,} \end{cases}$$

$$b{\updownarrow_i}a\,(w) := \begin{cases} w_0 \cdots w_{i-1} a w_{i+1} \cdots w_{n-1} & \text{, if } i < n \text{ and } w_i = b \\ \perp & \text{, otherwise.} \end{cases}$$

When giving feedback to a pupil regarding his hypothesis, suggesting only a single edit operation to modify the hypothesis will not be sufficient in most cases. That is why we introduce corrections as a means to group multiple edit operations together.

Definition 4. A correction $\rho = (\tau_1, \dots, \tau_m)$ is a sequence of edit operations. All edit operations are applied in order of the sequence from left to right. We utilize the definition of the functions induced by the edit operations to define the function $f_\rho : \Sigma^* \to \Sigma^*$ of a correction ρ as

$$f_{(\tau_1,\dots,\tau_m)}(w) := \begin{cases} w & \text{, if } m = 0, \\ f_{(\tau_2,\dots,\tau_m)}(v) & \text{, if } \tau_1(w) = v \text{ and } v \neq \perp, \\ \perp & \text{, otherwise.} \end{cases}$$

A correction ρ can be empty, written $\rho = ()$, if it contains zero edit operations.

As a next step, we need a concept to determine whether two corrections have the same behavior. Two corrections with the same behavior may differ in length, order or in the edit operations they contain. To transcend these concrete features, we need an equivalence on corrections. We discuss two notions of equivalence, namely syntactic and semantic equivalence. We introduce semantic equivalence first and discuss syntactic equivalence (similarity) and the relationship between these notions in Section 3.3.

Definition 5 (Equivalence)**.** We consider two corrections ρ_1, ρ_2 to be (semantically) equivalent, written $\rho_1, \equiv \rho_2$, if $\rho_1(w) = \rho_2(w)$ for every word $w \in \Sigma^*$.

We proceed by giving examples for corrections operating on words over the alphabet $\Sigma = \{a, b\}$. For example, we find that

$$(a{\uparrow_0}, b{\downarrow_1}) \equiv (b{\downarrow_0}, a{\uparrow_0}).$$

That is, because both corrections are only defined on words that start with b. Their application on a word causes the b at the first position of the word to disappear and introduces a at the first position instead. Both corrections differ in the order that the operations are applied.

An example for two corrections that are not equivalent reads as follows:

$$(a{\downarrow_1}) \not\equiv (b{\updownarrow_0}a, b{\uparrow_0}).$$

For instance, the results of applying both corrections to the word aa differ. While $(a\downarrow_1)(aa) = a$, $(b \updownarrow_0 a, b \uparrow_0)(aa) = \perp$. After having defined semantic equivalence we discuss in the following sections how simplifying and reordering corrections affects the preservation of semantic equivalence.

3.2 Simplification

In this section we discuss obvious cases in which corrections can be simplified based on their syntactical features. There exist combinations of insertions, deletions and replacements which are unnecessarily complex, i.e. the result of applying both to a word could be achieved with fewer operations. Deleting and inserting the same symbol at the same position over the course of a correction would be nonsensical. Inserting a symbol and replacing that symbol with another symbol later on can also be achieved by directly inserting the desired symbol. There are even more cases in which we can simplify combinations of edit operations. To formally define these trivial ways of simplifying corrections containing such combinations we introduce the concept of a simplification. We begin by defining how pairs of operations can be simplified and then extend the definition onto corrections of arbitrary length. This definition is a prerequisite to define an order on corrections which we can utilize to determine whether a correction is minimal regarding its syntactical features.

Example 6. Consider the hypothesis $w =$ *Heat influences yeast growth* over the alphabet $\Sigma = \{\textit{Heat,influences,yeast,growth,linearly}\}$ and the correction

$$\rho = (\textit{linearly}{\uparrow_4}, \textit{linearly}{\downarrow_4}).$$

We want to simplify ρ by substituting ρ with $()$. Intuitively, one would assume that ρ is semantically equivalent to $()$, because their effect on the given hypothesis w from above would be equal. That is, $(\textit{linearly}{\uparrow_4}, \textit{linearly}{\downarrow_4})(w) = w$ and $()(w) = w$. However, $\rho \not\equiv ()$. The empty correction $()$ is defined on all words over Σ and its application on a word yields that word without any modification. In contrast, ρ is defined on words of length greater than three. We conclude that $\rho \not\equiv ()$, because $\rho(w') = ()(w')$ only holds for words w' with $|w'| > 3$.

As a second example we consider $\rho_{\text{swapped}} = (\textit{linearly}{\downarrow_4}, \textit{linearly}{\uparrow_4})$ which is undefined on words that do not include the symbol *linearly* at index 4.

We conclude that there are two reasons why simplification cannot always preserve equivalence. The first reason is that an index occurring in an edit operation requires the word it is applied to to have a minimum length. This could be defined away by introducing a padding for words, which modifies all words to be sufficiently long. The second reason is a character mismatch. Deletions and insertions expect a specific symbol at a specific index in a word. We decided that in the context of the ITL generating helpful corrections for pupils we do not require a simplification to preserve semantic equivalence. That is, because in this context we are interested in determining whether a correction applied to a single input word is minimal or not. Instead of preserving semantic equivalence, we only demand that the simplification of a correction yields the same result on all words on which the original correction is defined.

Example 6 demonstrates that we cannot always call two corrections equivalent, even though the presented substitution of the correction $(\textit{linearly}{\uparrow_4}, \textit{linearly}{\downarrow_4})$ to $()$ seems to represent a more direct way of achieving the same result. Similarly, the correction $(\textit{linearly}{\uparrow_4}, \textit{linearly}{\updownarrow_4}\textit{nonlinearly})$ can be simplified to $(\textit{nonlinearly}{\uparrow_4})$. Even though we cannot arbitrarily interchange a part of a correction with such a simplification and vice versa, a simplified correction will be defined (not evaluate to $\perp$) and its result be equal for all words that the

β α	$a{\downarrow}_i$	$a{\uparrow}_i$	$a{\updownarrow}_i c$	$a{\downarrow}_{i+1}$	$a{\downarrow}_{i-1}$	$b{\uparrow}_i$	$b{\uparrow}_{i+1}$
$a{\downarrow}_i$	(α, β)	$()$	(α, β)	(α, β)	(α, β)	$a{\updownarrow}_i b$	(α, β)
$a{\uparrow}_i$	$()$	(α, β)	$(c{\uparrow}_i)$	(α, β)	(α, β)	(α, β)	(α, β)
$b{\updownarrow}_i a$	$(b{\downarrow}_i)$	(α, β)	$(b{\updownarrow}_i c)$	$b{\downarrow}_i$	$b{\downarrow}_i$	$a{\uparrow}_{i+1}$	$a{\uparrow}_i$

Table 1: The pair (α, β) can be simplified according to the table.

initial correction is defined on, which we denote in Corollary 10. Simplifying corrections in such a way is one aspect of finding minimal corrections. We will utilize this concept in Chapter 4 to define *minimality on corrections*.

Analogous to Example 6 we define a map of simplification rules given in Table 1, which maps a pair of operations (α, β) to one of two cases:

(i) a correction γ which may be empty, written $\gamma = ()$,
or contain a single edit operation, written $\gamma = (\tau)$, or

(ii) not simplifiable: (α, β).

If we can find a pair of adjacent edit operations (as a part of a correction) that matches a simplification rule, we can apply that rule to obtain a simpler correction. We do so by substituting the pair of edit operations with its simplification whilst preserving the rest of the correction. Next, we formally define this concept.

Definition 7. Let ρ, ρ' be two corrections where $\rho = (\tau_1, \ldots, \tau_n)$. Correction ρ' is simpler than ρ, denoted by $\rho' \lhd \rho$, if the pair (τ_i, τ_{i+1}) where $i \in \{1, \ldots, n-1\}$ matches a simplification rule from Table 1 and can be simplified to

- $()$ and ρ' has the form $(\tau_1, \ldots, \tau_{i-1}, \tau_{i+1}, \ldots, \tau_n)$, or
- (τ) and ρ' has the form $(\tau_1, \ldots, \tau_{i-1}, \tau, \tau_{i+2}, \ldots, \tau_n)$.

Note that in cases where the simplification of (τ_i, τ_{i+1}) is (τ_i, τ_{i+1}) as well, $\rho' = \rho$ and thus ρ' is not effectively a simpler correction.

After having defined what we mean by calling a correction simpler than another correction, we need to extend Definition 7 to relate corrections that are farther than one step of simplifying away. For instance, we can express that $(\textit{linearly}{\uparrow}_4, \textit{linearly}{\downarrow}_4) \rhd ()$, by applying $(a{\uparrow}_i, a{\downarrow}_i) \rhd ()$. However, we cannot yet express

$$(\textit{linearly}{\uparrow}_4, \textit{linearly}{\updownarrow}_4\textit{nonlinearly}, \textit{nonlinearly}{\downarrow}_4) \rhd (),$$

even though the sequence of simplifications

$$\begin{aligned} &(\textit{linearly}{\uparrow}_4, \textit{linearly}{\updownarrow}_4\textit{nonlinearly}, \textit{nonlinearly}{\downarrow}_4) \\ \rhd\ &(\textit{linearly}{\uparrow}_4, \textit{linearly}{\downarrow}_4) \\ \rhd\ &() \end{aligned}$$

which at first simplifies the pair of the second and third operation, and thereafter simplifies the pair of the first and second operation, holds. That is why we introduce transitivity to the definition of simplicity.

Definition 8. $\blacktriangleleft$ is the reflexive and transitive closure of $\lhd$.

After having defined $\blacktriangleleft$, we can express

$$(\mathit{linearly}{\uparrow_4}, \mathit{linearly}{\updownarrow_4}\mathit{nonlinearly}, \mathit{nonlinearly}{\downarrow_4}) \blacktriangleright ().$$

In order to make use of the standard notion of minimality on partial orders we need to prove that $\blacktriangleleft$ is a partial order.

Lemma 9. $\blacktriangleleft$ *is a (non-strict) partial order.*

Proof. Reflexivity and transitivity hold by construction of $\blacktriangleleft$. We prove anti-symmetry by contradiction. Note that $|\rho| < |\rho'|$ for two corrections ρ, ρ' where one is a (direct) simplification of the other, written $\rho \lhd \rho'$. Every applicable rule from Table 1 decreases the length of ρ by at least one. Assume that anti-symmetry does not hold. Let ρ, ρ' be corrections such that $\rho \blacktriangleleft \rho'$ and $\rho' \blacktriangleleft \rho$ and $\rho \neq \rho'$. Let $|\rho| = n$ and $|\rho'| = m$. Because $m < n$ and $n < m$ cannot be true simultaneously, only one of $\rho \blacktriangleleft \rho'$ and $\rho' \blacktriangleleft \rho$ can be true if $\rho \neq \rho'$. Simultaneous relationships $\rho \blacktriangleleft \rho'$ and $\rho \blacktriangleleft \rho'$ only exist when derived from reflexivity. In those cases $|\rho| = |\rho'|$ and thus $\rho = \rho'$. □

To find minimal (fully simplified) corrections amongst a set of corrections we can utilize the standard notion of minimality on partial orders, meaning a correction ρ is minimal regarding $\blacktriangleleft$ if there exists no correction ρ' with $\rho \neq \rho'$ such that $\rho' \blacktriangleleft \rho$. For instance, the corrections

$$(\mathit{linearly}{\uparrow_4}),\ (\mathit{Heat}{\updownarrow_0}\mathit{Air}, \mathit{pressure}{\uparrow_1})$$

are minimal regarding $\blacktriangleleft$.

Corollary 10. *Let ρ, ρ' be two corrections such that $\rho' \blacktriangleleft \rho$. If $\rho(w) \neq \perp$ with $w \in \Sigma^*$, then $\rho(w) = \rho'(w)$.*

After having defined $\blacktriangleleft$ as a concept for syntactic minimality, we now look at how we can group together corrections that have the same effect on a word based on their syntactical features. In doing so we abstract from the specific instance of a correction.

3.3 Similarity

Corrections should be seen as being equivalent, if they contain the same operations, but in a different order. It is not sufficient to check whether two corrections are permutations of each other to classify them as equivalent, since we need to account for the change of indices. For instance, if we conduct an insertion in the beginning of a correction, all subsequent operations operating on greater indices than the inserted symbol have to factor in the fact that the word they are operating on is extended by one symbol. In contrast, if we conduct that insertion at the end, all previous operations need to have their indices reduced by one. We may encounter that some operations cannot be permuted past each other, because they depend on each other. We consider two corrections to be similar if their length is equal, they contain the same edit operations in a different order without conflicts and the indices have been shifted accordingly.

(α,β)	$j<i$	$j=i$	$j>i$
$(a{\downarrow}_i, b{\downarrow}_j)$	$(\beta, a{\downarrow}_{i-1})$	$(b{\downarrow}_{j+1}, \alpha)$	$(b{\downarrow}_{j+1}, \alpha)$
$(a{\uparrow}_i, b{\downarrow}_j)$	$(\beta, a{\uparrow}_{i-1})$	$\perp$	$(b{\downarrow}_{j-1}, \alpha)$
$(a{\updownarrow}_i c,\ b{\downarrow}_j)$	$(\beta, a{\updownarrow}_{i-1} c)$	$\perp$	(β, α)

(α,β)	$j \le i$	$j>i$
$(a{\downarrow}_i, b{\uparrow}_j)$	$(\beta, a{\downarrow}_{i+1})$	$(b{\uparrow}_{j+1}, \alpha)$
$(a{\uparrow}_i, b{\uparrow}_j)$	$(\beta, a{\uparrow}_{i+1})$	$(b{\uparrow}_{j-1}, \alpha)$
$(a{\updownarrow}_i c,\ b{\uparrow}_j)$	$(\beta, a{\updownarrow}_{i+1} c)$	(β, α)

(α,β)	$j<i$	$j=i$	$j>i$
$(a{\downarrow}_i, b{\updownarrow}_j d)$	(β, α)	$(b{\updownarrow}_{i+1} d, \alpha)$	$(b{\updownarrow}_{j+1} d, \alpha)$
$(a{\uparrow}_i, b{\updownarrow}_j d)$	(β, α)	$\perp$	$(b{\updownarrow}_{j-1} d, \alpha)$
$(a{\updownarrow}_i c,\ b{\updownarrow}_j d)$	(β, α)	$\perp$	(β, α)

Table 2: Taking two adjacent edit operations α and β from a correction ρ and swapping them results in a reordered correction ρ'. $a, b, c, d \in \Sigma$ and $i, j \in \mathbb{N}$.

Definition 11. Let the (α, β) be a pair of edit operations. According to Table 2 we define the swapping of that pair as a partial map. The result of swapping (α, β) is a new pair (β', α'), or $\perp$ in some cases. Let ρ, ρ' be two corrections, where $\rho = (\tau_1, \ldots, \tau_n)$. Correction ρ' is a reordering of ρ, written $\rho \rightsquigarrow \rho'$, if

- the pair $(\tau_\beta, \tau_\alpha) \neq \perp$ is a swapping of (τ_i, τ_{i+1}) where $i \in \{1, \ldots, n-1\}$ and
- $\rho' = (\tau_1, \ldots, \tau_{i-1}, \tau_\beta, \tau_\alpha, \tau_{i+2}, \ldots, \tau_n)$.

Definition 12. $\simeq$ is the reflexive, symmetric and transitive closure of $\rightsquigarrow$.

If a successive series of reorderings in the form of $\rho \rightsquigarrow \cdots \rightsquigarrow \rho'$ is possible, we call ρ and ρ' similar. All reorderings that can be reached from ρ are contained in the equivalence class $[\rho]_\simeq$.

We proceed by discussing which corrections will always yield $\perp$. These are corrections which can be formed but are not useful because they contain operations that are logically inconsistent. Namely, if a correction can be reordered in a way that two adjacent operations match one of these cases:

$$(a{\uparrow}_i, b{\downarrow}_i), (a{\updownarrow}_i c, b{\downarrow}_i), (a{\uparrow}_i, b{\updownarrow}_i d), (a{\updownarrow}_i c, b{\updownarrow}_i d)$$

with $a, b, c, d \in \Sigma$ being pairwise unequal and $i \in \mathbb{N}$.

In contrast, there are corrections containing at least two operations which edit the same symbol and therefore both operations are dependent on one another. That is why swapping these operations does not preserve equivalence:

$$(a{\uparrow}_i, a{\downarrow}_i), (b{\updownarrow}_i a, a{\downarrow}_i), (a{\uparrow}_i, a{\updownarrow}_i c), (b{\updownarrow}_i a, a{\updownarrow}_i c).$$

As discussed in Section 3.3, corrections containing a pair matching one of the cases above can be simplified.

Theorem 13. *If $\rho \simeq \rho'$, then $\rho \equiv \rho'$.*

Proof. We prove Theorem 13 by induction on the number of reorderings n required to turn ρ into ρ'. For the base case $n = 0$ we know that $\rho \equiv \rho'$, because $\rho = \rho'$.

For $n > 0$ we assume that for the sequence of $n-1$ reorderings $\rho \rightsquigarrow \cdots \rightsquigarrow \rho''$ we know that $\rho'' \equiv \rho$. We show that the sequence $\rho \rightsquigarrow \cdots \rightsquigarrow \rho'' \rightsquigarrow \rho'''$ of n reorderings preserves equivalence, such that $\rho \equiv \rho'''$. In $\rho'' = (\tau_1, \ldots, \tau_m)$ exists a pair (τ_k, τ_{k+1}) for some $k \in \{1, \ldots, m-1\}$ with τ_k being the k-th and τ_{k+1} being the $k+1$-th operation in ρ''. The swapping of (τ_k, τ_{k+1}) into (τ'_k, τ'_{k+1}) according to Table 2 leads to $\rho''' = (\tau_1, \ldots, \tau'_k, \tau'_{k+1}, \ldots, \tau_m)$. By construction, $\rho''(w) = \rho'''(w)$ for every $w \in \Sigma^*$.

For instance, consider the first case in Table 2 $(a\downarrow_i, b\downarrow_j) \simeq (b\downarrow_j, a\downarrow_{i-1})$ where $i, j \in \mathbb{N}$ with $j < i$ and $a, b \in \Sigma$. Let $(\tau_1, \ldots, \tau_{k-1})(w) = w'$. The equation

$$(a\downarrow_i, b\downarrow_j)(w') = (b\downarrow_j, a\downarrow_{i-1})(w')$$

must hold for any $w' \in \Sigma^*$. If $(a\downarrow_i, b\downarrow_j)(w') =\perp$, then $(b\downarrow_j, a\downarrow_{i-1})(w') =\perp$. Otherwise, $w' = w_0 \cdots w_j \cdots w_i \cdots w_{n-1}$. Accordingly,

$$\begin{aligned}(a\downarrow_i, b\downarrow_j)(w') &= w_0 \cdots w_{j-1} w_{j+1} \cdots w_{i-1} w_{i+1} \cdots w_{n-1} \\ &= (b\downarrow_j, a\downarrow_{i-1})(w').\end{aligned}$$

We conclude that

$$\begin{aligned}\rho'''(w) &= (\tau_{k+2}, \ldots, \tau_m)((b\downarrow_j, a\downarrow_{i-1})((\tau_1, \ldots, \tau_{k-1})(w))) \\ &= (\tau_1, \ldots, \tau_{k-1}, b\downarrow_j, a\downarrow_{i-1}, \tau_{k+2}, \ldots, \tau_m)(w) \\ &= \rho(w).\end{aligned}$$

for any $w \in \Sigma^*$. The remaining cases from Table 2, where the swapping of (τ_k, τ_{k+1}) is not $\perp$, follow the same pattern. However, if the swapping of (τ_k, τ_{k+1}) is $\perp$ we do not consider ρ to be a useful correction as described earlier and cannot reorder ρ whilst preserving equivalence. $\square$

Chapter 4

Minimality

After having defined similarity and simplicity for corrections, which determine whether two corrections are equivalent or simplifications of each other respectively, we bring together these concepts with the notion of minimality introduced in the introduction of this thesis. Namely, that a correction is minimal regarding a word and a language if no proper prefix of that correction produces a word matching the language. Therefore, we define a necessary condition for minimality. This condition singles out corrections that are potentially minimal. Then, we demonstrate that the necessary condition is not sufficient, because it classifies corrections as minimal that are unnecessarily long. In doing so, we also discover that equivalence classes can be partly minimal, meaning that some potentially minimal corrections can be reordered into similar corrections which are not considered potentially minimal, which is undesirable. To ensure that every operation in a correction poses a meaningful step of correcting a word, we define a sufficient condition which we call actual minimality. Thereafter, we discuss how minimal corrections behave if the word they correct is a scattered subword of their resulting words. As a prerequisite for the proof in Section 4.3 we prove that corrections can be reordered into a normal form in which the operations are ordered by their type. Then, we prove that for a given word and a given language there exist only finitely many minimal corrections leading into the language.

4.1 Minimal Corrections

The notion of minimality we discuss in this thesis is based on the idea that a correction should be seen as minimal with regard to a word and a language if it does not contain a prefix which already produces a word of the language. Obviously, a correction should not be considered minimal if it can be simplified based on its syntactical features. The following definition of minimality connects these ideas.

Definition 14 (p-Minimality). Let L be a language, w a word and ρ a correction. A correction ρ is potentially minimal *(p-minimal)*, if there is no proper prefix ρ' of ρ, so that applying ρ' to w produces a word $w' \in L$ and ρ cannot be simplified further, i.e. there exists no ρ'' with $\rho'' \neq \rho'$ such that $\rho'' \blacktriangleleft \rho$.

Thus, p-minimality serves as a necessary condition. Every correction that matches our intuitive notion of minimality must satisfy the constraints defined by p-minimality. However, p-minimality does not accurately model our intuitive notion of minimality. Note, that a correction can be p-minimal even if it does not result in a word of the language L. The following example demonstrates why similarity is no congruence with regard to p-minimality:

Example 15. Let $L_{\text{ex}} = a^*b$ be a language induced by a regular expression and $w = a$ be a word. Consider the the following corrections for some $k \in \mathbb{N}$ denoting the count of inserted a symbols:

$$\rho = (a{\uparrow}_1, \dots, a{\uparrow}_1, b{\uparrow}_{k+1})$$

$$\sigma = (b{\uparrow}_1, \underbrace{a{\uparrow}_1, \dots, a{\uparrow}_1}_{k \text{ times}})$$

$$\sigma' = (b{\uparrow}_1)$$

When applying the correction ρ, the symbol a is appended k times, eventually appending the symbol b. This correction is *p-minimal* because we reach $\rho(w) \in L$ only after appending the symbol b in the last step $(a{\uparrow}_{k+1})$.

When applying the correction σ, initially symbol b will be inserted at position 1. The successive insertion of the symbol a at index 1 continually keeps symbol b at the end of the word. Thus, "pushing" all previously inserted symbols to the right. We can see that σ is not *p-minimal* , because we can find a prefix $\sigma' = (b{\uparrow}_1)$, so that $\sigma'(a) = ab \in L_{\text{ex}}$.

The correction σ' is *p-minimal* regarding w and L_{ex}, because there is no proper prefix σ'', so that $\sigma''(a) \in L_{\text{ex}}$.

On application of corrections $\rho(a)$ and $\sigma(a)$, both result in the word $a^{k+1}b$. The sequences ρ and σ are similar, because one can successively swap the insertion of b $(b{\uparrow}_{k+1})$ forward, until the correction starts with the insertion of b. We start with swapping the last two operations according to the rule $(b{\uparrow}_{j-1}, \alpha)$ from Table 2, so that $(a{\uparrow}_1, b{\uparrow}_{k+1})$ becomes $(b{\uparrow}_k, a{\uparrow}_1)$. We successively apply this rule $\rho \rightsquigarrow (a{\uparrow}_1, \dots, a{\uparrow}_k, a{\uparrow}_1) \rightsquigarrow \dots \rightsquigarrow \sigma$ and thus, $\rho \simeq \sigma$.

Observation 1. We find that corrections ρ and σ are similar $\rho \simeq \sigma$. Correction ρ is *p-minimal* , while correction σ is not. However, both belong to the same equivalence class $[\rho]_{\simeq}$. As a consequence, similarity is not congruent with regard to *p-minimality.*

We argue that ρ should not be considered minimal according to our intuitive notion of minimality, because the arbitrarily long prefix $(a{\uparrow}_1, \dots, a{\uparrow}_1)$ is not necessary to correct w from above so that the result matches L_{ex}. Leaving the prefix out wholly would represent a more minimal way of editing the word. To prevent such corrections from receiving the status of being minimal, we introduce the second definition of (actual) minimality.

Definition 16 (Minimality)**.** A correction ρ is (actually) minimal, if ρ is *p-minimal* and there exists no similar correction that is not p-minimal.

4.2 Observations on Minimality

In this chapter we examine how a minimal correction must be built if the word that is to be corrected is a scattered subword of the resulting word of the correction. Afterwards, we introduce a normal form for corrections as a prerequisite for the proof in the following section.

We demonstrate that the only minimal way to transform a scattered subword into a superword is to insert the symbols missing in the subword. That is, because any replacement or deletion operation present in a correction representing such a transformation can be simplified into a direct insertion.

Lemma 17 (Minimality and Scattered Subwords)**.** *Let w, w' be two words, where $w \leq w'$ and $\rho(w) = w'$ where ρ contains deletion or replacement operations. Then ρ cannot be minimal.*

Proof. Because $w \leq w'$, all symbols $a_0, \dots, a_{n-1}$ occurring in $w = a_0 \cdots a_{n-1}$ must also be present in w' in that order. ρ must contain exactly $|w'| - |w|$ insertions. If ρ contains fewer insertions, $\rho(w) \neq w'$. If ρ contains more insertions or insertions $s{\uparrow}_i$ with s being a symbol not occurring in w', ρ can be simplified. We consider three cases, of which at least one applies to ρ:

(i) ρ contains a deletion $s{\downarrow}_i$ with $s \in \Sigma, i \in \mathbb{N}$. Then, for ρ to reach w', ρ must also contain an insertion $s{\uparrow}_j$ with $j \leq |w'|$ to reinsert that symbol s later. Then, ρ can be reordered so that $s{\uparrow}_j$ and $s{\downarrow}_i$ are adjacent with $i = j$ and can be simplified by $(s{\downarrow}_i, s{\uparrow}_j) \rhd ()$.

(ii) ρ contains a replacement $s{\updownarrow}_i s'$ and an insertion $s{\uparrow}_j$ of the prior replaced symbol s, which effectively is a deletion of s'. Then, we can reorder ρ and simplify $(s{\updownarrow}_i s', s{\uparrow}_j) \rhd (s'{\downarrow}_{i+1})$ if $i = j$, or $(s{\updownarrow}_i s', s{\uparrow}_j) \rhd (s'{\uparrow}_i)$ if $j = i + 1$.

(iii) ρ contains a replacement $s{\updownarrow}_i s'$. Then, for ρ to reach w', ρ must contain the symbol with $s'{\updownarrow}_i s$ to bring back s, so $(s{\updownarrow}_i s', s'{\updownarrow}_i s) \rhd ()$ can be simplified when reordering both replacements to be adjacent.

Note that for (iii), if we try to swap $(s{\updownarrow}_i s', s'{\updownarrow}_i s)$ then $\rho(w) = \perp$. Hence, we do not consider ρ to be useful, and thus, not minimal as pointed out in Section 3.3, since a symbol is edited twice.

□

The type of edit operation affects the length of the resulting word differently, when the edit operation is applied to a word. A deletion decreases the word length by one, a replacement does not change the word length and an insertion increases the word length by one. To make corrections easier to handle we introduce a normal form for a correction. This normal form states that a correction in normal consists of three segments such that each segment contains operations of one type only. The first segment may contain only deletions, while the second and third segments contain replacements and insertions respectively. If the correction does not contain an operation of a type, the segment corresponding to that type is empty. If a correction in normal form is applied to a word there is a point at the end of the second segment such that all operations prior to that point are deletions and replacements which do not increase the word length and all operations after that point are insertions. In particular, this means that after applying all operations of the first and second segment the resulting word is a scattered subword of all words that result from applying insertions of the third segment.

We can bring every correction into that normal form by swapping every deletion to the beginning and swapping every insertion to the end of the correction. If at some point we encounter the scenario that two operations cannot be swapped according to the rules given by Table 2 we can disregard the correction we are trying to normalize, because it is logically inconsistent or can be simplified.

Lemma 18 (Equivalent Normal Form)**.** *Let $L \subseteq \Sigma^*$ be a language and ρ be a correction with $\rho(w) \neq \perp$ for all $w \in L$. Then there exists a similar correction ρ' such that $\rho' \in [\rho]_{\simeq}$ and ρ is ordered in a normal form where all deletions come first, followed by all replacements and finally all insertions.*

Proof. Let $\rho = (\tau_1, \ldots, \tau_n)$ and d, r, i be the number of deletions, replacements and insertions in ρ, respectively. Then there exists a similar reordered correction ρ' in normal form such that

$$\rho' = (\tau_1, \ldots, \tau_d, \tau_{d+1}, \ldots, \tau_{d+r}, \tau_{d+r+1}, \ldots, \tau_{d+r+i})$$

with $\tau_1 \cdots \tau_d$ being deletions, $\tau_{d+1} \cdots, \tau_{d+r}$ being replacements and $\tau_{d+r+1} \cdots \tau_{d+r+i}$ being insertions.

We start by propagating all deletions to the beginning of the correction. If there exists a deletion τ_i and a replacement or insertion τ_j in ρ with $i = j + 1$, we swap τ_j and τ_i, adjusting the operating indices according to Table 2, until the first d operations in ρ are deletions.

Then, we propagate all replacements into the middle segment. If there exists a replacement τ_i and an insertion τ_j with $i = j + 1$, we swap τ_i and τ_j, also adjusting their operating indices, until $\tau_{d+1} \cdots \tau_{d+r}$ are replacements.

If, at some point we cannot swap τ_i and τ_j according to Table 2, because the swapping is $\perp$ we proved that ρ is inconsistent and τ_i and τ_j depend on each other. Therefore, we omit ρ. □

4.3 Finitely Many Minimal Corrections

After having introduced a normal form for corrections we utilize the observations presented in the previous section to prove that there are finitely many minimal corrections leading into the target language. Our definition of minimality depends on a given word and a given (target) language – an incorrect hypothesis and a finite representation of all correct hypotheses, respectively. Obviously, we are only interested in corrections that actually correct a hypothesis in a way that the result is a correct hypothesis. Because our definition of minimality does not include that a minimal correction must produce a word of the language in the following proof we cover corrections that are minimal and result in a word of the target language.

Theorem 19. *For any given word and language there are finitely many minimal corrections leading into the given language. Moreover, there is a finite number of words that can be produced by applying all minimal corrections on the given word.*

We outline the structure of the following proof. As described in the previous section we can reorder every correction into a normal form and assume, without loss of generality, that all corrections are given in normal form. Therefore, we can partly apply each minimal correction leading into the target language to the point where all deletions and replacements are applied. We observe that there are finitely many ways to correct a word in a minimal way by only using deletions and insertions. The partial application described above leads to an intermediate word. If that word is in the target language, according to the definition of minimality there cannot exist more minimal corrections having the operations as a prefix that lead to the intermediate word. If the intermediate word is not in the target language, the correction must contain one or more insertions. As a next step we prove that there exist finitely many minimal corrections consisting of insertions that turn the intermediate word into a word of the target language. Therefore, we use the fact that the intermediate word is a scattered subword of every result produced by applying insertions. If we join each minimal correction consisting of deletions or replacements leading to an intermediate word with each minimal correction consisting of insertions leading from the intermediate word into the target language, we obtain a finite upper bound for all minimal corrections leading into the target language.

Proof. Let Σ be an alphabet. Let $L \subseteq \Sigma^*$ be a language and $w \in \Sigma^*$ a word over the alphabet. We define the set of all minimal corrections for w leading to L via

$$M_L(w) := \{\, \rho \mid \rho \text{ is minimal and } \rho(w) \in L \,\}.$$

We prove that $M_L(w)$ is finite. We start by observing that the number of minimal corrections which preserve or decrease the length of the word in each operation is finite.

Corrections without Insertions These corrections result in words of shorter or equal length when compared to $|w|$. We define them via

$$\begin{aligned} K_L(w) := \{\, \rho \mid \rho \text{ is minimal and } \rho \text{ consists only of} \\ \text{replacement or deletion operations} \,\}. \end{aligned}$$

We describe how the corrections in $K_L(w)$ are built. When building a minimal correction for w with replacements and deletions, every symbol in w can be edited at most once. When deleting a symbol, it is not present in the word afterwards. As a consequence, all subsequent edit operations may not refer to this symbol anymore, but refer to all remaining symbols. If the correction contained a number of deletions greater than $|w|$, the correction would evaluate to $\perp$ when applied to w. It would also evaluate to $\perp$ if a symbol in w was deleted twice. Note that the correction may contain two deletions having the same index or symbol, due to the index shift happening to the remaining symbols after applying a deletion to a word. Deleting any number of symbols yields 2^d possible options to build deletions, if d is the number of deleted symbols and $0 \leq d \leq |w|$. Additionally, only those symbols in w may be replaced which are not deleted during the whole correction. Each symbol w_i in w with $i \in \{0, \ldots, n-1\}$ that is not deleted may be replaced once against any symbol $s \in \Sigma$ with $s \neq w_i$. A correction containing a replacement $s \updownarrow_i s$ with $i \in \{0, \ldots, n-1\}$ and $s \in \Sigma$ is never minimal, because it can be simplified. Replacing any number of remaining symbols yields $|\Sigma|^{|w|-d}$ options. Hence, for any given word w, $|K_L(w)|$ is at most $(|\Sigma|+1)^d$ and thus, finite.

Note that in general $K_L(w)$ contains corrections leading to words that are part of the language L and corrections leading to words that are not in L. That is, because for a correction to be minimal, we do not demand its result to be in L. Instead, we demand that regardless of how the correction is reordered, the correction cannot be simplified and no proper prefix produces a word in L.

We denote the set of all words that can be reached by applying all corrections in a set S to a word w as $W_w(S) := \{\, w' \mid \rho(w) = w' \text{ with } \rho \in S \,\}$. Every minimal correction in $M_L(w)$ will produce a word $w' \in W_w(K_L(w))$ when all deletions and replacements are applied. In the following, we prove that for any word w' there exist finitely many minimal corrections consisting solely of insertions and resulting into a word in L.

Corrections with Insertions. Next, we consider corrections that solely consist of insertions and, thus, increase the length of the word in each step. We define this set of corrections via

$$\begin{aligned} E_L(w) := \{\, \rho \mid \rho &\text{ is minimal and } \rho \text{ consists only of} \\ &\text{insertion operations and } \rho(w) \in L \,\}. \end{aligned}$$

We prove that for any given word w and any given language L the set $E_L(w)$ is finite.

Let $E_L(w)$ be infinite. Hence, $W_w(E_L(w))$ is also infinite, because for a word there exist only finitely many corrections containing only insertions leading to that word. According to Higman's Lemma, there must be two words x and y with $x, y \in W_w(E_L(w))$, so that $x \leq y$ with $|x| < |y|$. There exist two minimal corrections ρ_x and ρ_y with $\rho_x, \rho_y \in E_L(w)$ consisting only of insertion operations, so that $\rho_x(w) = x$ and $\rho_y(w) = y$. According to the definition of $E_L(w)$, $w \leq x$ and $x, y \in L$. Because $w \leq x$ and $x \leq y$, a correction leading from w to y only by using insertion operations will inevitably pass through x. This means ρ_x or a reordering of ρ_x is a proper prefix of ρ_y. Thus, ρ_y cannot be minimal because $\rho_x(w) \in L$. However, ρ_y has to be minimal by definition of $E_L(w)$. This poses a contradiction to the assumption that $E_L(w)$ is infinite. We conclude that $E_L(w)$ must be finite.

All Corrections. Because of Lemma 18 we can assume, without loss of generality, that all corrections in $M_L(w)$ follow the normal form where all deletions come first, followed by all replacements and finally, all insertions. Thus, a correction $\rho_{\min} \in M_L(w)$ can be represented as $\rho_{\min} = \rho_1\, \rho_2$ such that ρ_1 contains the deletions and replacements and ρ_2 contains the insertions. Let $\rho_1(w) = w'$. The application of the deletions and replacements leads to the intermediate word w'. The correction ρ_1 is in $K_L(w)$ which we proved finite above. Hence, there are finitely many intermediate words w'. The second part of $\rho_{\min}$ is ρ_2 which is in $E_L(w')$. And as we proved above $E_L(w')$ is also finite for every w'.

We obtain an upper bound for all minimal corrections leading to a word of the language L, namely $M_L(w) \subseteq \bigcup_{\rho_1 \in K_L(w)} \bigcup_{\rho_2 \in E_L(\rho_1(w))} \rho_1 \rho_2$ where the index sets of both unions are finite. Thus, we proved that $M_L(w)$ is also finite. $\square$

Chapter 5

Computation of Minimal Corrections

In this chapter we present an algorithm that follows the structure of the proof from the previous chapter. We begin by describing how the corrections without insertions $K_L(w)$ can be computed and checked for minimality. Thereafter, we present how based on the words resulting from the application of all corrections in $E_L(w)$, the corrections can be extended by insertions. In doing so we compute $E_L(w)$. That computation is closely related to the termination condition of the algorithm, which is thematized simultaneously. As a next step, we point out optimizations that we have implemented to avoid computing all shortening and length preserving corrections naively. That can be achieved by starting with short corrections and consecutively appending operations whilst testing for p-minimality between each step. Moreover, we utilize a sorted normal form which builds on our definition of a normal form for corrections set out earlier. This sorted normal form can also be used to speed up checking whether a correction is minimal. Thereafter follows a description of our implementation and a discussion of the findings from running the implementation on an exemplary grammar. We then examine the runtime complexity of each step in the algorithm. We close this chapter by proposing how the implementation can be improved further.

5.1 The Algorithm

From the proof of Theorem 19 we can derive an algorithm to compute all minimal corrections. We divide the algorithm in four steps. Steps one to three compute $K_L(w)$, i.e. corrections without insertions, while step four checks whether the algorithm can terminate and consecutively extends the corrections found in step three by insertions, thus, computing $E_L(w)$. We implemented the computation of $K_L(w)$ and enumerate the insertions from step four without checking the termination condition. Instead, our implementation terminates after a fixed iteration count received as a parameter.

1. **Corrections without Insertions**. Enumerate all corrections $(\tau_1 \cdots \tau_n)$, for $n = 1, \ldots, |w|$ where τ_i is the $i-$th operation and τ_i is a deletion or

replacement.

2. **Filter for p-Minimality**. Filter the corrections from step one for p-minimality by checking two conditions. For each corrections $\rho = (\tau_1, \dots \tau_n)$:

   (i) For all (τ_i, τ_{i+1}) with $i \in \{1, \dots, n-1\}$ we check if (τ_i, τ_{i+1}) can be simplified to (τ) or $()$. If so, we filter out ρ.

   (ii) Solve the word problem mentioned in Section 2.3 for every proper prefix $\rho' < \rho$ with the CYK algorithm as described in [11] or the Earley algorithm as described in [2]. If $\rho'(w) \in L$, filter out ρ.

3. **Filter for a-Minimality**. Filter corrections from step two for a-minimality. For every correction ρ calculate all reorderings $\rho' \in [\rho]_{\simeq}$ with $\rho' \neq \rho$ and execute step two for ρ'.

4. **Extend by Insertions**. After $K_L(w)$ has been computed we need to find out which corrections from the previous steps can be extended by insertions, such that the extended correction is minimal and leads into the language. We take the resulting words from $K_L(w)$ as basis to find these insertions. The idea is that we construct a regular language $L_{\text{missing}}(w)$ which describes for a resulting word all words that can be produced by inserting symbols into the resulting word. Hence, we utilize the language $L_{\text{missing}}(w)$ to find the missing symbols that need to be inserted, and with that the corresponding insertion.

   First, we compute $\rho(w) = w'$ for every ρ from step three. Let $w' = b_0 \cdots b_{n-1}$. We define the language of all superwords of w' that have at least one symbol preceding w' via $L_{\text{missing}}(w') = \Sigma^+ b_0 \Sigma^* b_1 \Sigma^* \cdots \Sigma^* b_{n-1} \Sigma^*$. Now we can intersect this language $L_{\text{missing}}(w')$ with the target language L with the construction mentioned in Section 2.3. Then we solve the emptiness problem for the intersection which also is described in Section 2.3. If the intersection is not empty, we know that there may be a minimal correction leading to that word in the intersection. To find out which symbol the insertion must insert, we replace Σ^+ against each symbol $s \in \Sigma$ in $L_{\text{missing}}(w')$. If we again intersect $s b_0 \Sigma^* b_1 \Sigma^* \cdots \Sigma^* b_{n-1} \Sigma^*$ with L and find that the intersection is not empty for s, we extend all corrections leading to w' by an insertion $s\uparrow_0$. We also apply the insertion $s\uparrow_0$ to w' and add the result of the application to a list of words for which we need to find the next insertion by repeating the described procedure. However, we do also need to check for possible insertions between every symbol. That means, Σ^+ must be moved between every gap in $L_{\text{missing}}(w')$, e.g. $\Sigma^* b_0 \Sigma^+ b_1 \Sigma^* \cdots \Sigma^* b_{n-1} \Sigma^*$ and if we find out through the intersection with L that a symbol can possibly be inserted in that gap, we try $\Sigma^* b_0 s b_1 \Sigma^* \cdots \Sigma^* b_{n-1} \Sigma^*$ for every $s \in \Sigma$. So:

   - If $L \cap L_{\text{missing}}(w') = \emptyset$, we know that we cannot insert any symbol into the gap in w' and match the target language L with the result. We move Σ^+ to the next gap.
   - If $L \cap L_{\text{missing}}(w') \neq \emptyset$ we know that we can extend ρ by an insertion. To find out, by which insertion $s\uparrow_i$ we can extend ρ, we check $L_s(w') = s b_1 \Sigma^* b_2 \Sigma^* \cdots \Sigma^* b_{n-1} \Sigma^*$ for every $s \in \Sigma$.

We save for every correction ρ by which insertions $s{\uparrow}_i$ it can be extended and add new corrections $\rho' = \rho s{\uparrow}_i$. Of course, these corrections ρ' must be checked for minimality as described in step three. We can terminate if there is no $\rho'(w)$ that we can insert a symbol into. Hence, repeat step four for all new corrections ρ' from step four, until no new corrections are found. Since the algorithm is derived from the proof of Theorem 19 we know that at some point the algorithm will terminate when all minimal corrections have been computed. Step four may produce insertions that insert symbols that have been deleted previously. Such corrections can be simplified and must be filtered.

5.2 Implementation and Optimizations

For steps one to three of the algorithm we found some optimizations that will be presented in this section. Steps one, two and three can be combined by incrementing the maximum length of the correction by one progressively. Instead of enumerating all reorderings of all corrections consisting of deletions and replacements, one can generate all corrections of length one and filter all corrections leading to a word in L. Any extensions of such corrections cannot be minimal by definition of minimality. All remaining corrections can then be extended by all possible deletions and replacements that can be applied after the first operation has been applied to the input word w.

By tracking which symbols in the input word already have a corresponding operation in the correction, one can avoid deleting or replacing a symbol twice. Hence, we only generate useful corrections. For instance, if we are correcting the word *abc* and have computed the correction $(b{\updownarrow}_1 d, a{\downarrow}_0)$, any extension of that correction that deletes or replaces a symbol at index 0 should be omitted, because the correction would edit the symbol b twice.

In addition to the normal form presented in Section 4.2, which is not unique, one can demand that the operations of each type are ordered by their index as second constraint. Namely, that deletions are sorted non-descending by the indices they are operating on and replacements are sorted ascending. This strategy further reduces the number of generated corrections.

Generating corrections in their normal form straightaway has a second advantage. Checking whether a correction can be simplified is less costly, because the rules in column four and five in Table 1 can be checked by propagating each pair of deletion and replacement to the transition border between the deletion and replacement segment, so that they are adjacent. For example in the correction

$$(\underline{b{\downarrow}_1}, c{\downarrow}_5, c{\updownarrow}_0 d, \underline{a{\updownarrow}_1 b})$$

we can propagate the first operation to the second position and the fourth operation to the third position resulting in

$$(c{\downarrow}_6, \underline{b{\downarrow}_1}, \underline{a{\updownarrow}_1 b}, c{\updownarrow}_0 d) \rhd (c{\downarrow}_6, a{\downarrow}_2, c{\updownarrow}_0 d)$$

and see that the simplification rule $(b{\downarrow}_i, a{\updownarrow}_i b) \rhd (a{\downarrow}_{i+1})$ matches the second and third operation. Note that in Table 1 the rule is transposed for better readability. This method discovers corrections which cannot be minimal because they can be simplified early in the computation. Without using this method we would

$$\begin{aligned}
\textit{hypothesis} &\rightarrow \langle\textit{simple}\rangle \mid \langle\textit{complex}\rangle\\
\textit{simple} &\rightarrow \langle\textit{independentVariable}\rangle \text{ `\textit{influences}'}\\
&\ \langle\textit{dependentVariable}\rangle\\
\textit{complex} &\rightarrow \langle\textit{simple}\rangle\ \langle\textit{modal}\rangle\\
\textit{modal} &\rightarrow \text{`\textit{, but only}'}\ \langle\textit{valueRange}\rangle\\
\textit{independentVariable} &\rightarrow \text{`\textit{Light}'} \mid \text{`\textit{Intensity}'} \mid \text{`\textit{Wavelength}'}\\
\textit{dependentVariable} &\rightarrow \text{`\textit{plant growth}'} \mid \text{`\textit{stem length}'} \mid \text{`\textit{photosynthesis rate}'} \mid\\
&\ \text{`\textit{leaf color}'}\\
\textit{valueRange} &\rightarrow \text{`\textit{up to 500nm}'} \mid \text{`\textit{from 500nm}'} \mid \text{`\textit{between 500 and 600}}\\
&\ \text{\textit{nm}'}
\end{aligned}$$

Figure 1: A simple grammar which describes syntactically correct hypotheses. Its initial symbol is *hypothesis*.

have discovered the fact that the above correction is not minimal in step three of the algorithm when computing all reorderings. However, with this method we do not need to compute all permutations of the correction, and we can also exclude every correction from the computation that is an extension of the above correction.

To prevent computing a word problem twice, the solutions of all previously calculated word problems can be cached using the memoization technique.

When programming step three, we need to generate all reorderings for a given correction. Utilizing the Steinhaus-Johnson-Trotter algorithm as described in [8] is the obvious solution. The algorithm generates all permutations of a sequence by swapping two elements of the sequence in each step. This mode of operation aligns nicely with the definition of similarity, since every reordering of a correction needs to account for index modifications of the affected operations.

Our implementation expects a parameter *iterations* which specifies the number of deletions or replacements and insertions. If *iterations* is set to two, the computed corrections may contain at most two corrections of type deletion or replacement and in addition at most two insertions. We ran the algorithm on the exemplary grammar from Figure 1.

5.3 Experimental Results and Findings

For the hypothesis *Light influences*, the algorithm computed the following corrections:

$$(\textit{plant growth}{\uparrow_2}), (\textit{stem length}{\uparrow_2}), (\textit{photosynthesis rate}{\uparrow_2}), (\textit{leaf color}{\uparrow_2}).$$

This example demonstrates that the algorithm does compute helpful corrections.

We proceed by presenting an example for a correction that is rather undesirable even though we consider it to be minimal according to the definition set out in this thesis. The following correction proposes to turn *Light influences* into *Intensity influences plant growth*:

$$(\textit{Light}{\updownarrow_0}\textit{influences}, \textit{influences}{\updownarrow_1}\textit{plant growth}, \textit{Intensity}{\uparrow_0}).$$

The correction above demonstrates that our definition of minimality is still insufficient, because it represents an odd way of correcting the given hypothesis. Instead, the correction

$$(\textit{Light}\updownarrow_0\textit{Intensity}, \textit{plant growth}\uparrow_2)$$

would be a better correction to achieve the same result. However, the latter correction is not minimal because it can be reordered so that *plant growth* is inserted first. That is why it is not amongst the computed minimal corrections. We observe that undesirable minimal corrections edit symbols that are present at the same position of the hypothesis before and after their application. The same behavior can be observed when considering

$$(\textit{influences}\downarrow_1, \textit{Light}\updownarrow_0\textit{plant growth}, \textit{Intensity}\uparrow_0, \textit{influences}\uparrow_1).$$

The correction mentioned above deletes and inserts *influences* and therefore poses another example of correcting the hypothesis *Light influences* in an odd way. Further research is required to exclude corrections like those presented above.

When running the algorithm on the hypothesis *Light influences Wavelength, but only* we see that some of them are symmetrical and semantically equivalent but not similar, such as

$$(\textit{, but only}\downarrow_3, \textit{Wavelength}\updownarrow_2\textit{stem length}) \text{ and}$$
$$(\textit{Wavelength}\downarrow_2, \textit{, but only}\updownarrow_2\textit{stem length}).$$

Obviously, when displaying them to a pupil both corrections should be combined and displayed as one, because they describe the same way of editing the hypothesis. Therefore, we require a concept to determine whether two corrections are symmetrical like the two corrections above.

5.4 Runtime and Complexity

We ran the algorithm on a machine with an 11th Gen Intel(R) Core(TM) i7-11800H processor with a base clock speed of 2.30 GHz and 32 GB of RAM.

Computing the results for the first hypothesis in the previous section took 50 ms with *iterations* set to one. When running the algorithm with *iterations* set to two, we obtained 28 corrections in 1015 ms. For *iterations* set to values greater than two, our implementation of the algorithm did not yield a full a result, because the heap was full. For the hypothesis in the second example we obtained 12 minimal corrections in 70 ms with *iterations* set to one. When setting *iterations* to two the algorithm computed 63 minimal corrections in 14918 ms. When setting *iterations* to three, the algorithm did not return a full result. However, all corrections containing deletions and insertions were returned after 1077 ms. Before any correction including insertions was found, the heap was full.

At the end of this chapter we discuss which actions can be taken to improve the implementation such that the algorithm yields results for minimal corrections of greater length.

When comparing the computation time for *iterations* set to one (corrections with a maximum length of two) and two (corrections with a maximum length of four) we can see that computing took 20 times longer for the first hypothesis and 213 times longer for the second hypothesis. This huge increase can be explained by analyzing the asymptotic runtime complexity of the algorithm.

One should mention that since the proof of Theorem 19 yields an upper bound for $M_L(w)$ the algorithm does not compute corrections in a target-oriented way. Many corrections are computed that turn out to be not minimal when the corrections without insertions are joined with the corrections with insertions. Hence, a superset of $M_L(w)$ is computed which is then reduced. We do not have a method to compute $M_L(w)$ directly. We proceed by analyzing each step of the algorithm.

In the first step there are at most $(|\Sigma| + 1)^w$ corrections in $K_L(w)$, because each symbol in the word can either be deleted, replaced against any other symbol from the alphabet, or ignored by a correction. The first part of the second step, namely checking adjacent operations for simplification, takes $\mathcal{O}(|\rho|)$ time for each ρ in $K_L(w)$. All remaining corrections get checked against the condition of the second part of step two, namely whether a prefix exists that produces a word of the language. Therefore, for each correction $|\rho|$ word problems must be solved with an algorithm mentioned in Section 2.3. Hence, step two runs in $\mathcal{O}(|\rho| \cdot |G| \cdot |w|^3)$ for each correction ρ. The third step computes all reorderings of each correction, which can be done in $\mathcal{O}(\rho!)$. Step three is a very costly computation, since each reordering of each correction ρ must be checked against the conditions of step two. The fourth step of the algorithm is even more costly, because for each word w' resulting from each correction an intersection between a regular and a context-free language is computed $|w'|$ times, which can be done in polynomial time, and checked for emptiness, which can be done in $\mathcal{O}(|G|)$ as pointed out in Section 2.3. The correction leading to w' is then extended by insertions and is checked for minimality again, which adds the complexity of steps two and three for each appended insertion. One can not determine a priori how many iterations are necessary until all minimal corrections have been found. Our implementation brute forces step four, which implies an exponential blow-up of the search space for insertions.

After having analyzed the runtime complexity and discussed computation results worth remarking we proceed by mentioning aspects in which our implementation can be improved. The implementation should produce less intermediate data so that less heap space is required. One approach is to use iterator functions instead of arrays. In doing so we would eliminate intermediate data after each iteration. Additionally, we can use index structures to prevent saving parts of memoized hypotheses twice. Also, instead of memoizing the solution to the word problem for each correction, we can use the non-descending sorted normal form mentioned previously. Before looking up whether a solution to the prefix of a correction has been calculated, we can normalize the prefix and then do the lookup. Similarly, if the lookup fails and the word problem has been solved for that prefix, we save the solution together with the normalized version of the prefix. This method requires insignificantly more time but saves space. The normalization of a correction ρ can be done in $\mathcal{O}(|\rho|^2)$ as described in the previous chapter.

Chapter 6

Conclusion

Through the course of the thesis we have defined a system for editing words and a notion of minimality to determine which constraints a minimal edit sequence must satisfy. We then proved that for any given word and language there is a finite number of minimal edit sequences. On the basis of that proof we described an algorithm which can compute this finite set of edit sequences. We presented how we implemented the first part of the algorithm and found a few optimizations which arise as a result from the definitions of simplicity and similarity.

6.1 Results

The quality of results that we were able to compute were partly suitable to be used in an ITL as a means to provide feedback to pupils. However, due to the runtime complexity of the algorithm we were only able to compute short corrections even for small grammars and hypotheses.

6.2 Outlook

If we assume that in practice hypotheses have a length of five to ten words then the implementation needs to be capable of computing minimal corrections of length ten within a few seconds, so that the ITL provides feedback to a pupil in reasonable time. Our implementation is not sufficient to meet this requirement. The inherent complexity of the algorithm constraints a fast computation for larger hypotheses and larger grammars. However, for smaller inputs there are many optimizations to be made. Our implementation is written in TypeScript which is a just-in-time compiled high-level programming language [13]. Using an embedded programming language and choosing a more efficient encoding for corrections may speed up the computation significantly.

Alternatively, because of the exponential blow-up of the search space spanned by possible indices, operation types and symbols, using a declarative programming language may be more suitable to solve the computation problem presented in this thesis. Optimizations as presented in the previous chapter are necessary to render the algorithm suitable for practical use in an ITL.

Even though we tried to define away as many nonsensical ways to correct a word, some computed corrections still seem unnecessarily long. That is because the algorithm presented does not make use of the inner structure of the given language – a context-free grammar in this case. If information about which terminal is derived from which nonterminal was incorporated into the algorithm, the quality of the computed results could be improved further. For the specific use case of correcting hypotheses of pupils in an ITL application, we propose that an algorithm computing suggestions should be more custom-tailored to the given language of valid hypotheses.

Bibliography

[1] Alfred V. Aho and Thomas G. Peterson. “A Minimum Distance Error-Correcting Parser for Context-Free Languages”. In: *SIAM Journal on Computing* 1.4 (1972), pp. 305–312. DOI: `10.1137/0201022`.

[2] Jay Earley. “An efficient context-free parsing algorithm”. In: *Communications of the ACM* 13.2 (1970), pp. 94–102.

[3] Wendy Hall, Su White, and Beverly Woolf. “Interactive Systems for Learning and Teaching”. In: Jan. 1998.

[4] Graham Higman. “Ordering by divisibility in abstract algebras”. In: *Proceedings of the London Mathematical Society* 3.1 (1952), pp. 326–336.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007. ISBN: 978-0-321-47617-3.

[6] Juraj Hromkovič. *Theoretical computer science: introduction to Automata, computability, complexity, algorithmics, randomization, communication, and cryptography*. Springer Science & Business Media, 2003.

[7] Norbert Hundeshagen. *Berechenbarkeit und Komplexität*. Lecture Notes. University of Kassel, 2019/2020.

[8] Selmer M. Johnson. “Generation of permutations by adjacent transposition”. In: *Mathematics of computation* 17.83 (1963), pp. 282–285.

[9] Marit Kastaun et al. “ProfiLL – Professionalisierung durch intelligente Lehr-Lernsysteme”. In: Nov. 2020, pp. 357–363. ISBN: 978-3-8309-9246-2.

[10] Martin Lange. *Formale Sprachen und Logik*. Lecture Notes. University of Kassel, 2019.

[11] Martin Lange and Hans Leiß. “To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm”. In: *Informatica Didactica* 8.2009 (2009), pp. 1–21.

[12] Vladimir I. Levenshtein et al. “Binary codes capable of correcting deletions, insertions, and reversals”. In: *Soviet physics doklady*. Vol. 10. 8. Soviet Union. 1966, pp. 707–710.

[13] Ross McIlroy. *Firing up the Ignition interpreter*. 2016. URL: `https://v8.dev/blog/ignition-interpreter` (visited on 11/29/2022).

[14] Clemente Pasti et al. “On the Intersection of Context-Free and Regular Languages”. In: *arXiv preprint arXiv:2209.06809* (2022).

[15] Uwe Schöning. *Theoretische Informatik - kurz gefasst.* Spektrum Akademischer Verlag, 2008. ISBN: 9783827418241.

[16] Joseph Swernofsky and Michael Wehar. "On the Complexity of Intersecting Regular, Context-Free, and Tree Languages". In: *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II.* Ed. by Magnús M. Halldórsson et al. Vol. 9135. Lecture Notes in Computer Science. Springer, 2015, pp. 414–426. DOI: `10.1007/978-3-662-47666-6\_33`.