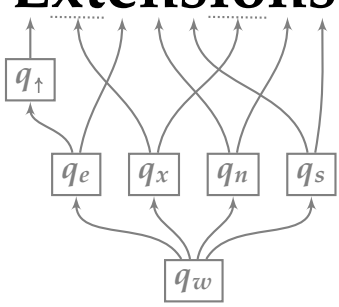
Complexities of Order-Related Formal Language Extensions

Martin Berglund



Department of Computing Science
Umeå 2014

Complexities of Order-Related Formal Language Extensions

Martin Berglund

PhD Thesis, May 2014
Department of Computing Science
Umeå University
Sweden

Department of Computing Science
Umeå University
SE-901 87 Umeå, Sweden

mbe@cs.umu.se

ISBN 978-91-7601-047-1
ISSN 0348-0542
UMINF 14.13

Cover photo by Tc Morgan (used under Creative Commons license BY-NC-SA 2.0).
Printed by Print & Media, Umeå University, 2014.

Abstract

The work presented in this thesis discusses various formal language formalisms that extend classical formalisms like regular expressions and context-free grammars with additional abilities, most relating to order. This is done while focusing on the impact these extensions have on the efficiency of parsing the languages generated. That is, rather than taking a step up on the Chomsky hierarchy to the context-sensitive languages, which makes parsing very difficult, a smaller step is taken, adding some mechanisms which permit interesting spatial (in)dependencies to be modeled.

The most immediate example is shuffle formalisms, where existing language formalisms are extended by introducing operators which generate arbitrary interleavings of argument languages. For example, introducing a shuffle operator to the regular expressions does not make it possible to recognize context-free languages like $a^n b^n$, but it does capture some non-context-free languages like the language of all strings containing the same number of as, bs and cs. The impact these additions have on parsing has many facets. Other than shuffle operators we also consider formalisms enforcing repeating substrings, formalisms moving substrings around, and formalisms that restrict which substrings may be concatenated. The formalisms studied here all have a number of properties in common.

1. They are closely related to existing regular and context-free formalisms. They operate in a step-wise fashion, deriving strings by sequences of rule applications of individually limited power.

2. Each step generates a constant number of symbols and does not modify parts that have already been generated. That is, strings are built in an additive fashion that does not explode in size (in contrast to e.g. Lindenmayer systems). All languages here will have a semi-linear Parikh image.

3. They feature some interesting characteristic involving order or other spatial constraints. In the example of the shuffle multiple derivations are in a sense interspersed in a way that each is unaware of.

4. All of the formalisms are intended to be limited enough to make an efficient parsing algorithm at least for some cases a reasonable goal.

This thesis will give intuitive explanations of a number of formalisms fulfilling these requirements, and will sketch some results relating to the parsing problem for them. This should all be viewed as preparation for the more complete results and explanations featured in the papers given in the appendices.

Sammanfattning

Denna avhandling diskuterar utökningar av klassiska formalismer inom formella språk, till exempel reguljära uttryck och kontextfria grammatiker. Utökningarna handlar på ett eller annat sätt om ordning, och ett särskilt fokus ligger på att göra utökningarna på ett sätt som dels har intressanta spatiala/ordningsrelaterade effekter och som dels bevarar den effektiva parsningen som är möjlig för de ursprungliga klassiska formalismerna. Detta står i kontrast till att ta det större steget upp i Chomsky-hierarkin till de kontextkänsliga språken, vilket medför ett svårt parsningsproblem.

Ett omedelbart exempel på en sådan utökning är s.k. *shuffle*-formalismer. Dessa utökar existerande formalismer genom att introducera operatorer som godtyckligt sammanflätar strängar från argumentspråk. Om shuffle-operator introduceras till de reguljära uttrycken ger det inte förmågan att känna igen t.ex. det kontextfria språket $a^n b^n$, men det fångar istället vissa språk som inte är kontextfria, till exempel språket som består av alla strängar som innehåller lika många a:n, b:n och c:n. Sättet på vilket dessa utökningar påverkar parsningsproblemet är mångfacetterat. Utöver dessa shuffle-operatorer tas också formalismer där delsträngar kan upprepas, formalismer där delsträngar flyttas runt, och formalismer som begränsar hur delsträngar får konkateneras upp. Formalismerna som tas upp här har dock vissa egenskaper gemensamma.

1. De är nära besläktade med de klassiska reguljära och kontextfria formalismerna. De arbetar stegvis, och konstruerar strängar genom successiva applikationer av individuellt enkla regler.

2. Varje steg genererar ett konstant antal symboler och modifierar inte det som redan genererats. Det vill säga, strängar byggs additivt och längden på dem kan inte explodera (i kontrast till t.ex. Lindenmayer-system). Alla språk som tar upp kommer att ha en semi-linjär Parikh-avbildning.

3. De har någon instressant spatial/ordningsrelaterad egenskap. Exempelvis sättet på vilket shuffle-operatorer sammanflätar annars oberoende deriveringar.

4. Alla formalismera är tänkta att vara begränsade nog att det är resonabelt att ha effektiv parsning som mål.

Denna avhandling kommer att ge intuitiva förklaring av ett antal formalismer som uppfyller ovanstående krav, och kommer att skissa en blandning av resultat relaterade till parsningsproblemet för dem. Detta bör ses som förberedande inför läsning av de mer djupgående och komplexa resultaten och förklaringarna i de artiklar som finns inkluderade som appendix.

Preface

This thesis consists of an introduction which discusses some different language formalisms in the field of formal languages, touches upon some of their properties and their relations to each other, and gives a short overview of relevant research. In the appendix the following six articles, relating to the subjects discussed in the introduction, are included.

Paper I Martin Berglund, Henrik Björklund, and Johanna Björklund. Shuffled languages – representation and recognition. *Theoretical Computer Science*, 489-490:1–20, 2013.

Paper II Martin Berglund, Henrik Björklund, and Frank Drewes. On the parameterized complexity of Linear Context-Free Rewriting Systems. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 21–29, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

Paper III Martin Berglund, Henrik Björklund, Frank Drewes, Brink van der Merwe, and Bruce Watson. Cuts in regular expressions. In Marie-Pierre Béal and Olivier Carton, editors, *Proceeding of the 17th International Conference on Developments in Language Theory (DLT 2013)*, pages 70–81, 2013.

Paper IV Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catatrophic backtracking behavior in practical regular expression matching. Submitted to the *14th International Conference on Automata and Formal Languages (AFL 2014)*, 2014.

Paper V Martin Berglund. Characterizing non-regularity. Technical Report UMINF 14.12, Computing Science, Umeå University, `http://www8.cs.umu.se/research/uminf/`, 2014. In collaboration with Henrik Björklund and Frank Drewes.

Paper VI Martin Berglund. Analyzing edit distance on trees: Tree swap distance is intractable. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 59–73. Prague Stringology Club, Czech Technical University, 2011.

Acknowledgments

I must firstly thank my primary advisor, Frank Drewes, who made all this both possible, enjoyable and inspiring. In much the same vein I thank my co-advisor, Henrik Björklund, who knows many things and throws a good dinner party, as well as my unofficial co-advisor Johanna Björklund, who organizes many things and makes people have fun when they otherwise would not. I must also thank the rest of my university colleagues, in the Natural and Formal Languages Group (thanks to Niklas, Petter and Suna) and many others in many other places. A special thank you to all the support and administrative staff at the department and university, who have helped me out with countless things on countless occasions, a fact too easily forgotten. I also owe a great debt to all my research collaborators outside of this university, including but not limited to Brink van der Merwe and Bruce Watson. I thank those who have given me useful research advice along the way, like Michael Minock and Stephen Hegner.

On the slightly less professional front I thank my family for their support, in particular in offering places and moments of calm when things were hectic. I thank my friends who have helped both distract from and inspire my work as appropriate, thanks to, among many others, Gustaf, Sandra, Josefin, Sigge, Mårten, John, a Magnus or two, some Tommy, perhaps a Johan and a Maria, and many many more.

I wish to dedicate this work to the memory of Holger Berglund and Bertil Larsson, both of my grandfathers, who passed away during my studies leading up to this thesis.

Contents

Chapter 1
Introduction

This thesis studies extensions of some classical formal languages formalisms, notably for the regular and context-free languages. The extensions center primarily around additions of operations or mechanism that constrain or loosen order, with a special focus on parsing in the presence of such ordering loosening or constraints. This statement is, of course, quite vague. The extensions take such a form that they modify the way in which a grammar or automaton generates a string. "Order" here refers to a spatial view of this generation.

Very informally, imagine a person with finite memory (a natural assumption) who is tasked to write down certain types of strings of symbols on paper. The ways in which he or she is allowed to move around the paper will impact the types of strings they can write. If they are required to start at the left (i.e., start with the first, leftmost, symbol) and work their way through the string in a left-to-right fashion they can easily write the string $abcabcabc\ldots$, but the strings $\{ab, aabb, aaabbb, \ldots\}$ (i.e. as followed by an equal number of bs) require them to remember the number of as written if it is done in a left-to-right fashion, which is arbitrarily much information to remember. If the person is permitted to keep track of the middle of the string, adding symbols on the right and left side simultaneously, they can easily write strings of the second type by simply in each step writing one a and one b, never having to remember how many steps have been made. The first variant, where the person has to work left-to-right and cannot remember arbitrarily much is an informal description of finite automata, a characterization of the very important class of regular languages. The case where the person keeps track of the middle and writes on both the left and the right corresponds to the class of linear context-free languages, another very classical concept. From this perspective it is easy to imagine additional extensions of the formalisms, a notable example is that the writer may remember *multiple* positions, and add symbols to them interchangeably, which corresponds to a more complex language class.

Among the variety of formalisms one can imagine that modify the way in which generation happens it is important to remain true to the spirit of classical mechanisms. This tends to return to the idea that only finite memory is required when viewed from the correct perspective. Consider for example the following trivial formalism.

Example 1.1 (Mappings of copy-languages) Given two mappings σ_1, σ_2 from $\{a, b\}$ to arbitrary strings and a string w decide whether there exist some $\alpha_1, \ldots, \alpha_n \in \{a, b\}$ such that $\sigma_1(\alpha_1)\cdots\sigma_1(\alpha_n) \cdot \sigma_2(\alpha_1)\cdots\sigma_2(\alpha_n) = w$. ◇

This particular example is simplified quite a bit, but there are popular formalisms exhibiting this exact behavior, where some underlying "decision" is made in one derivation step, and the result gets reflected in multiple (but normally constant number of) places in the output string. The mapping may make it difficult to actually recognize the decision after the fact, but the problem is very related to parsing for some language classes with similar spatial dependencies.

Not all formalisms are concerned with instilling this extra level of order on the string, we also consider cases where separate "underlying decisions" may become intertwined or otherwise not get spatially separated in the way we are used to. Consider the following example of a fairly important real-world problem where difficulties arise from insufficient order.

Example 1.2 (Parallel program verification) Let P be a computer program which when run produces some output string. Assume we have a context-free grammar G which is such that if a string w can be output by a correct run of P then w can be derived in G. Then, whenever P produces output that is not accepted by G we know that P is not functioning properly.

Now run n copies of the program P, in parallel, all producing output simultaneously into the same string w. In w the outputs of the different instances of P will be arbitrarily interleaved. Now we wish to use G to determine whether this w is consistent with n copies of P running correctly. ◇

The *lack* of order makes this problem difficult, to answer the question we need to somehow track how single decisions in single instances of the program may have been spread out across the resulting string. As these artifacts may be arbitrarily far apart this problem becomes rather difficult, and the unfortunate reality is that the string w may *appear* consistent despite a program failing to run in accordance with G, due to some other part of the string masking the fault.

The cases in Example 1.1 and Example 1.2 are almost each others opposites, but are connected in that they are both possible to describe by a spatial dependence in the strings. A simple block-wise dependence in Example 1.1, and an entirely scattered dependence in Example 1.2.

Earlier Work This work is deeply related to the preceding licentiate thesis [Ber12] by the same author. While this thesis is intended to replace this earlier work it may for some readers be of interest to refer back to [Ber12] for further examples and explanations of many of the same concepts.

1.1 Formal Languages

Formal languages is a vast area of study, it covers both a lot of practical algorithmic work with numerous application areas, as well as more theoretically founded mathematical study. The original subject of study in formal languages are string languages. These are concerned with sequences of symbols from a finite alphabet, which is usually denoted Σ. Going forward we will usually simply assume that Σ is the latin alpha-

bet, $\Sigma = \{a, b, c, \ldots, z\}$, meaning that usual words like "cat" and "biscuit" are strings in this formal sense. We let ε denote the empty string. A *language* is a, potentially infinite, set of strings. One trivial example is the empty set, $\varnothing$, the language that contains no strings, and the set of *all* strings, which we denote Σ^*. Other examples include finite languages like $\{cat\}$ and $\{cat, biscuit\}$, infinite languages like the set of all strings *except* "cat", the language $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$, and, over the alphabet $\{0, \ldots, 9\}$, the language $\{3, 31, 314, 3141, 31415, 314159, \ldots\}$.

The most immediate subject of study in formal languages is representing them. Finite languages like $\varnothing$ and $\{cat, biscuit\}$ are easy to describe by exhaustively enumerating the strings they contain. Some infinite languages are also trivial, the language containing all strings except "cat" can be described by enumerating the strings it does not contain. However, languages like $\{ab, aabb, aaabbb, aaaabbbb, \ldots\}$ and $\{3, 31, 314, 3141, 31415, \ldots\}$ are more complex. Certainly the "dots"-notation used here to describe them is flawed, as the generalization intended is ambiguous at best.

This question of representation for languages is the core of formal language theory, arbitrary languages can of course represent almost arbitrary computational problems, but the question of how the language can be *finitely represented* restricts matters. Specifically what is studied is *classes* of languages defined by the type of descriptional mechanism capable of capturing them. Most trivially, the finite languages is a language class, defined by being describable through simply enumerating the strings.

While language classes are typically defined using the formalism that can describe them it is important to remember that languages are abstract entities that exist in and of themselves. In most formalisms a given language can be represented by many different grammars or automata, and few of the usual formalisms have unique normal forms that can be computed.

1.2 An Example Representation

To make the previous more concrete let us establish a representation for formal language formalisms as rather visual grammars. We call these instances of formalisms "grammars" here, but the sketches used here intentionally straddle the boundary of what is traditionally called "grammars" and what is called "automata".

1.2.1 Our Grammar Sketch

Essentially the grammars will consist of two parts; "memory", or state, and rules. *States*, or *non-terminals*, represent what the formalism is remembering about the string it is generating. They are simply symbols attached to the intermediary output. The grammars always start out in the state *S*, the *initial non-terminal* in an otherwise empty string. The rules specify which state can generate what in the string. We write the rules down as shown in Figure 1.3, where three rules are given which generate the language $\{a, aba, ababa, abababa, \ldots\}$ using two non-terminals. The left-hand side shows the state which the rule applies to. The little dot below the *S* represents the position in the string the *S* is keeping track of. On the right-hand side is shown what the formalism generates, in the case of the first rule it outputs the symbol "a", followed by a position

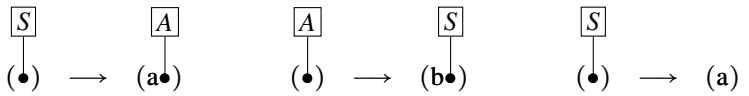
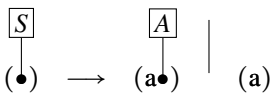


Figure 1.3: A regular grammar generating the language $\{a, aba, ababa, abababa, \ldots\}$ using three rules. S is the initial non-terminal.

which is kept track of by the second non-terminal A. In effect S "remembers" that the next symbol should be an "a", and the second non-terminal A remembers that the next symbol should be a "b" (and we then go back to S. The third rule allows the S to generate a final "a" and ending the generation by producing no new non-terminal. Since the first and third rule have the same left-hand side the abbreviation



is sometimes used in place of writing both out in full. We write the generation of strings in the way shown in Figure 1.4, where a derivation is performed using the grammar from Figure 1.3 to generate the string "ababa". Notice that, as usual, none

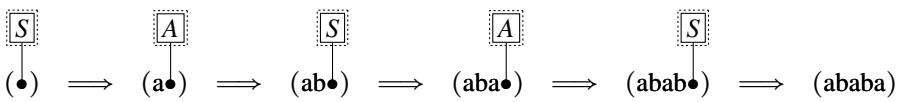


Figure 1.4: A derivation of the string "ababa" using the grammar Figure 1.3. The derivation starts with the initial non-terminal S, applies the first rule, this produces the non-terminal A, making the second rule the only possible one. This is then repeated, and finally the third rule is used to get rid of the non-terminal S entirely. As there is no more state left the derivation is finished, and the string "ababa" has been generated. The dotted outline around non-terminals show which non-terminal is used in the next rule application, but as there is only one to choose from in each step it is not very informative here.

of the intermediary strings are "generated", all states must be gone before generation is finished. The black bullets, or "positions" act as the points of the string tracked by attached non-terminals. Their role will become slightly more complex later on.

1.2.2 Generating Regular Languages

A simple and important class of languages that we can generate with grammars of the type we have sketched are the regular languages. Specifically the regular languages are precisely the following.

Definition 1.5 (Regular Grammars) A grammar of the form sketched in Figure 1.3 is regular if

- It is finite.
- Each right-hand side contains zero or one symbol from Σ and zero or one non-terminal attached to the position (bullet).
- The position is to the right of the symbol if one exists.

Every regular language can be represented by a grammar of this form. ◇

A grammar G then generates exactly the strings one can produce by starting from S attached to the initial position, and then repeatedly picking a rule, and replacing an instance of the non-terminal on the left-hand side of the rule (this is then only possible if that non-terminal exists in the string) by the new substring on the right-hand side of the rule. If a point is reached where no non-terminal exists in the string the generated string w is in the language, denoted $w \in \mathcal{L}(G)$. That is, $\mathcal{L}(G)$ is a set consisting of exactly these strings.

1.2.3 Regular Expressions as an Alternative

A regular expression is another way of expressing a language, which is equivalent to the description of a regular grammar in Definition 1.5, but which is often more compact and convenient, as well as being very popular in practical use.

Definition 1.6 (Regular Expressions) A regular expression over the alphabet Σ is, inductively, the following. For each $\alpha \in \Sigma$ and regular expressions R and T:

- ε is a regular expression with $\mathcal{L}(\varepsilon) = \{\varepsilon\}$.
- α is a regular expression with $\mathcal{L}(\alpha) = \{\alpha\}$.
- $R \cdot T$ is a regular expression with $\mathcal{L}(R \cdot T) = \{wv \mid w \in \mathcal{L}(R), v \in \mathcal{L}(T)\}$ (i.e. the concatenation of the strings in the languages of the subexpressions). We often write RT as an abbreviation.
- $R|T$ is a regular expression, with $\mathcal{L}(R|T) = \mathcal{L}(R) \cup \mathcal{L}(T)$.
- R^* is a regular expression, with $\mathcal{L}(R^*) = \{\varepsilon\} \cup \{wv \mid w \in \mathcal{L}(R), v \in \mathcal{L}(R^*)\}$ inductively. That is, the concatenation of arbitrarily many strings from R. ◇

1.3 Computational Problems in Formal Languages

With formalisms for representing formal languages in hand it is time to consider the various questions that can be asked about them. An immediate example is the emptiness problem; given a grammar G, does it generate the language $\varnothing$? Computing the answer to this problem is easy for context-free languages[1], but it is undecidable to determine if a context-free language generates Σ^*, the language of *all* strings.

[1] We have not defined the context-free languages properly, but all regular languages are context-free, and some context-free languages are not regular, so it can serve as an unspecific more powerful example.

Many problems also deal with languages themselves, being somewhat independent of representation. For example, given two context-free languages (i.e., two languages that can be generated by some context-free grammar) L and L', is the language $L \cup L'$ also context-free? It, in fact, is, and given any context-free grammar for L and L' a grammar for $L \cup L'$ can easily be constructed. The same does *not* hold for the language $L \cap L'$, some context-free languages have an intersection that is not context-free. The regular languages, however, as closed under intersection, so for all regular languages L and L' the language $L \cap L'$ is regular as well, a fact we will make use of later.

It is important to remember that while grammars may determine languages the grammar is not necessarily always in the most convenient form. Given a regular grammar G it is easy to determine if it generates Σ^*, but it is hard to determine if a context-free grammar generates Σ^*. However, context-free grammars can generate all the regular languages as well, but even if a context-free grammar generates a regular language it is *still* hard to tell if it generates Σ^* (in fact, as Σ^* *is* regular this is a part of the general problem).

The problem we are primarily concerned with in this work, however, is the *membership problem*. This is the problem of determining whether a string belongs to a given language or not. There are at least three different variations of the membership problem of interest here.

Definition 1.7 (The Uniform Membership Problem) Let $\mathcal{G}$ be a class of grammars (e.g. context-free grammars) such that each $G \in \mathcal{G}$ defines a formal language. *The uniform membership problem for* $\mathcal{G}$ is "Given a string w and some $G \in \mathcal{G}$ as input, is w in the language generated by G?" ◇

This case is certainly of interest at times, but fairly often the details of the formalism $\mathcal{G}$ are irrelevant to the practical problem. The most notable example is in instances where the language is known in advance and can be coded into the most efficient representation imaginable. A second type of membership problem accounts for this case, by simply considering *only* the string part of the input.

Definition 1.8 (The Non-Uniform Membership Problem) Let L be any language. Then the *non-uniform membership problem for* L is "Given a string w as input, is w in L?" ◇

There is a third approach, called fixed-parameter analysis, which provides more nuance in the complexity analysis of the membership problems. In this approach any part of the problem may be designated the "parameter", and is considered secondary in complexity concerns. This is treated in Section 3.5.1.

The final, and perhaps most practically interesting case, is *parsing*. In parsing we no longer expect to get just a "yes" or "no" as an answer to the question whether the string belongs to the language, we expect a *description* of *why* the string belongs to the language. For example, when asking whether the string "ababa" can be generated by the grammar in Figure 1.3 the answer should not be "yes", it should be some description of the generation procedure in Figure 1.4. In most practical cases any solution to the membership problems in Definition 1.7 and 1.8 will construct some representation of this answer *anyway* (the case of Definition 1.8 becomes more complicated,

however, as the internal representation of the language may be hard to practically decipher). Thanks to this fact this thesis will primarily refer to and work on membership problems, despite it being understood that parsing is the real goal.

1.4 Outline of Introduction

In the following chapters we will look at some formalisms that are of interest for this thesis (and are studied in the papers included). We will start out using variations on the informal notation demonstrated above (as in Figure 1.3), modifying it to illustrate the general idea of how the formalisms differ. More formalized, and deeper, matters are then considered for each.

For the most part each chapter starts out with a self-contained informal introduction, with a more formal treatment being undertaken at the end. This is intended to cater to multiple types of readers. A casual reader may be most interested in reading every chapter only up until the section marked by a star, *, and then skipping to the next. The non-starred portion of the introduction is self-contained. For a deeper treatment the entirety of the introduction may be read, but, of course, in the end most of the material is in the accompanying papers, and readers familiar with the area may be best served only skimming the introduction in favor of proceeding to the papers.

Chapter 2 gives a light introduction to *shuffle* formalisms, which are related to Example 1.2, extending regular expressions with an operator that interleaves strings. This sets the scene for a short summary of the contents of Paper I, with some words on Paper V in addition. Chapter 3 discusses synchronized substrings, similar to Example 1.1, going into a summary of Paper II. Chapter 4 discusses some extensions of regular expressions, primarily dealing with the *cut* operator, which provides a more limited string concatenation, but also giving an overview of some of the details of real-world matching engines. Papers III and IV are then discussed in brief in this context. Chapter 5 discusses distance measures on languages for handling errors. This yields a short discussion of grammar-instructed block movements, where substrings may be moved around in the string depending on how they were generated by a grammar, leading into Paper VI. Finally, Chapter 6 provides a short summary.

CHAPTER 2

Shuffle-Like Behaviors in Languages

Shuffle in the title of this chapter refers to shuffling a deck of cards, specifically to the riffle shuffle, where the deck is separated into two halves, which are then interleaved. This idea, transferred to formal languages, is intended to capture situations such as the one illustrated in Example 1.2, where multiple mostly independent generations are performed in an interleaved fashion.

2.1 The Binary Shuffle Operator

We specifically transfer the riffle shuffle to the case of strings in the following way. Starting with the strings "ab" and "cd", the *shuffle* of "ab" and "cd" is denoted $ab \odot cd$, and results in the language $\{abcd, acbd, cabd, acdb, cadb, cdab\}$, that is, all ways to interleave "ab" with "cd" while not affecting the internal order of the strings. Let us make this point slightly more formal with a definition.

Definition 2.1 (Shuffle Operator) Let w and v be two arbitrary strings. Then $w \odot \varepsilon = \varepsilon \odot w = \{w\}$. Recall that ε denotes the empty string.

If both w and v are non-empty let $w = \alpha w'$ and $v = \beta v'$ (for strings w' and v', single symbols α and β). Then $w \odot v = \alpha(w' \odot v) \cup \beta(w \odot v')$. ◇

This is then generalized to the shuffle of two languages in a straightforward way, for two languages L and L' we let the shuffle $L \odot L'$ be the language of shuffles of strings in L with strings in L', or $\bigcup\{w \odot w' \mid w \in L, w' \in L\}$.

Example 2.2 (The shuffle of two languages) Let $\mathcal{L} = \{ab, abab, ababab, \ldots\}$ and $\mathcal{L}' = \{bc, bcbc, bcbcbc, \ldots\}$. Then the shuffle $\mathcal{L} \odot \mathcal{L}'$ contains, for example, *abbc* (all of "ab" which is in $\mathcal{L}$ occurring before "bc" which is in $\mathcal{L}'$), *babc* (same strings interleaved differently), and *abbabcbcabab*. ◇

2.2 Sketching Grammars Capturing Shuffle

Without further ado we can fairly easily modify the graphical grammars we previously introduced to generate shuffles of this kind. We for the moment stick to the regular

languages, such as in Figure 1.3, and then extend the formalism to combine them. There are a number of restrictions on the shape of the grammars in this formalism:

1. There may be at most one non-terminal position marker (black dot) on the right-hand side of a rule.

2. The right-hand side of a rule may contain at most one generated symbol (from Σ), and the non-terminal position marker, if there is one, must be to the right of the symbol.

These two requirements together in effect require the grammar to work from left to right, generating one symbol at a time. We now, on the other hand, permit more than one non-terminal to attach itself to the same "position" (we will also in the next section outline how a non-terminal may be attached to another). In this way (with the correct precise semantics) we arrive at shuffle formalisms of various kinds. Consider for example the grammar in Figure 2.3. Effectively this grammar will generate the

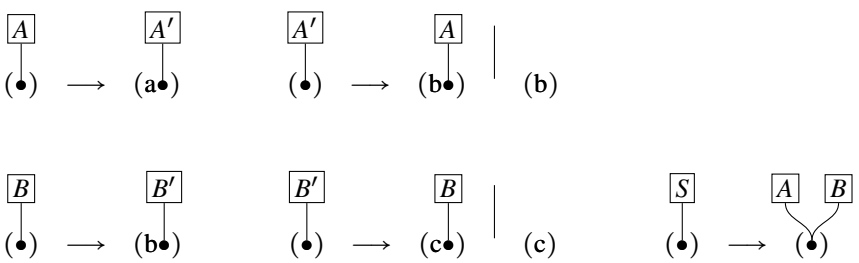


Figure 2.3: A grammar generating a language exhibiting a shuffling behavior.

shuffle $L_A \odot L_B$, if we let L_A and L_B denote the language the grammar would generate if we started with the non-terminal A and B respectively. The way the grammar works is that it starts out (since there is only one rule for the initial state) by attaching two states, A and B, to the same position. The intended semantics of this is that all non-terminals attached to the same position can generate symbols simultaneously, while the others are unaware. A derivation of the string "bacbbc" is shown in Figure 2.4.

The languages that these grammars express are closely related to the languages generated by (or, rather, denoted by) regular expressions extended with the shuffle operator. For example, the grammar in Figure 2.3 corresponds to the expression $(ab)^* \odot (bc)^*$. These expressions form a part of what is known as "shuffle expressions". This is not all there is to the grammars or to shuffle expressions. Consider the grammar in Figure 2.5. This grammar is able to keep attaching arbitrarily many additional instances of the non-terminal S to the initial position, each S can produce one "a" to transition into the non-terminal B, which simply produces a "b" and disappears. An example derivation is shown in Figure 2.6. The language generated by this grammar is, obviously, $ab \odot ab \odot ab \odot \cdots$ (the language which is such that in every prefix the number of "a"s is greater or equal to the number of "b"s, and the entire string has the same number of "a"s and "b"s). This language is not expressed by any regular

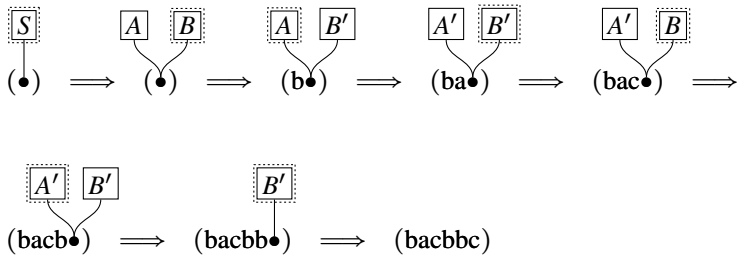


Figure 2.4: A derivation of the string "bacbbc" in the grammar from Figure 2.3. Notice that there are multiple ways this string could be derived, here the last "b" "belongs" to the string "ab" generated by the A non-terminal, but the second to last could be used instead.

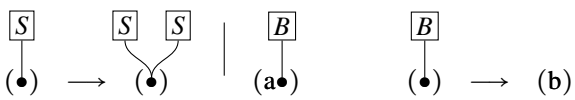


Figure 2.5: A grammar that showcases the ability to shuffle arbitrarily many strings.

expression extended by the shuffle operator, but general shuffle expressions have an additional operator for this purpose.

2.3 The Shuffle Closure

To complete the picture, shuffle expressions are regular expressions (regular expressions are introduced in short in Definition 1.6, for a more complete introduction see e.g. [HMU03]) extended with the binary shuffle operator from Definition 2.1 and the unary shuffle closure operator, denoted $\mathcal{L}^{\odot}$ (for some expression or language $\mathcal{L}$). The shuffle closure captures exactly languages of the type illustrated in Figure 2.5, where

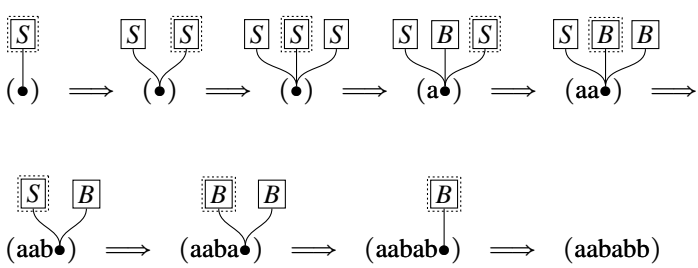


Figure 2.6: An example derivation using the grammar from Figure 2.5.

arbitrarily many strings from a language are shuffled together. Recall that $\mathcal{L}(E)$ denotes the language generated/denoted by a grammar/expression E.

Definition 2.7 (Shuffle Closure) For a language $\mathcal{L}$ the shuffle closure of $\mathcal{L}$, denoted $\mathcal{L}^{\odot}$ is $\{\varepsilon\} \cup \{w \odot \mathcal{L}^{\odot} \mid w \in \mathcal{L}\}$. For an expression E of course $\mathcal{L}(E^{\odot}) = \mathcal{L}(E)^{\odot}$. ◇

The language generated by the grammar in Figure 2.5 is then simply $(ab)^{\odot}$.

The grammatical formalism we have so far sketched can represent simple shuffles, but it is not yet complete. The shuffle expression $(ab)^{\odot}c$ causes trouble. If we start out with the grammar in Figure 2.5 (and we more or less have to) we somehow have to designate a non-terminal to generate the final c, but we have no way of ensuring that all the *other* non-terminals finish generating first. As such further extensions to the grammars are required. To leap straight to the illustrative example, see Figure 2.8. Here the first rule generates two non-terminals, one A and one C, where the C is

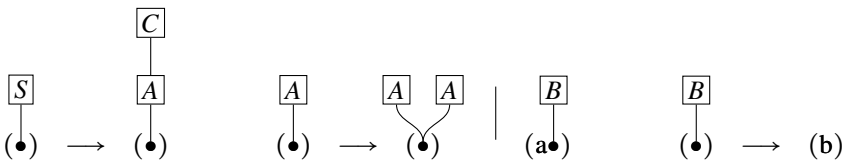


Figure 2.8: This grammar illustrates an extension which enables the combination of shuffling with sequential behavior. Specifically this grammar generates the language $(ab)^{\odot}c$.

no longer connected to the position tracked, but is rather connected to the A. We say that C *depends* on A. The semantics is that rules may only be applied to non-terminals attached only to the position, all non-terminals that depend on another must be left alone. If new non-terminals are created from the one on which C depends then C will depend on all the new non-terminals. If all non-terminals on which C depends are removed (i.e. they finish generating) then C gets attached to the position. See the example run in Figure 2.9. Notice how the C is generated with the first rule application, but then no rule can be applied to it until all the non-terminals it depends on have disappeared, meaning, in this case, that it will generate the last symbol in the string, since all the As (and subsequent Bs) much first finish.

2.4 Shuffle Operators and the Regular Languages

It may be interesting to note that a shuffle expression which uses *only* the binary shuffle operator, $\odot$, still denotes a regular language (i.e. any regular formalism, such as finite automata or regular expressions, can represent the same language). That is, we do not *need* to generate multiple non-terminals to construct a shuffle language of this kind. This is fairly easy to see, recall the simple shuffle grammar in Figure 2.3, and then consider a new grammar with non-terminals containing multiple symbols. Consider specifically the two left-most rules in that figure, and then consider the new rules in

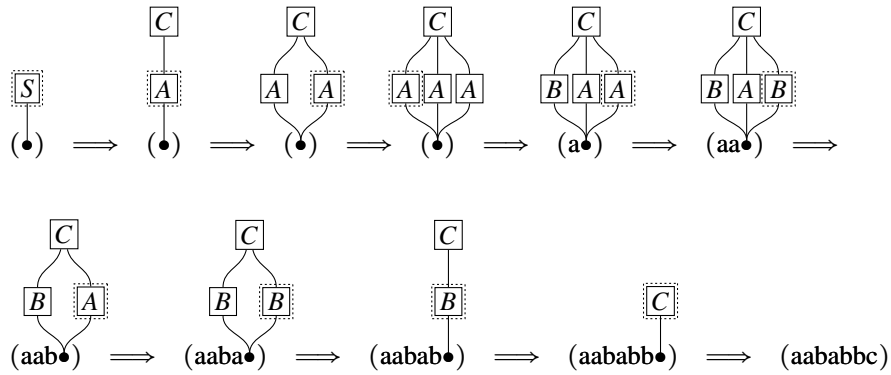


Figure 2.9: Generation of the string "aababbc" using the grammar from Figure 2.8.

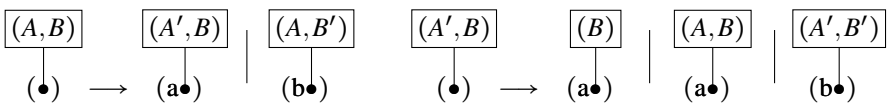


Figure 2.10: Some example rules from a regular grammar for the shuffle grammar in Figure 2.3.

Figure 2.10. That is, we create non-terminals which contain all the non-terminals of a certain step of the generation for the original grammar. The first left-hand side, with the nonterminal (A,B), corresponds to the situation created immediately after the first rule applied in Figure 2.4, and the two possible right-hand sides correspond to either applying a rule to the A or to the B. Similarly the second left-hand side corresponds to when A' and B are tracking the position, and either A' is chosen to disappear generating a, or just produce a and generate a new A, or B generates a b turning into B'. Instead of the grammar in Figure 2.3 we get a grammar with the non-terminals (S), (A,B), (A',B), (A,B'), (A',B'), (A), (A'), (B), (B'), quite a number, but this grammar only has a single non-terminal tracking the point at any point of a generation. This procedure demonstrates that only one non-terminal is necessary, so the language generated is regular. However, a potentially exponential number of non-terminals may be generated performing the construction, so this *cannot* be combined with the efficient parsing for regular languages to produce an efficient uniform parsing algorithm. This construction works for any expression with arbitrarily many binary shuffle operators, as they still only give rise to a constant number of possible sets of non-terminals attached to the tracked position, making this product construction generate a finite regular grammar.

Applying the shuffle closure, however, does not necessarily preserve regularity. Recall that the language $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular, as reading it from left to right arbitrarily much information (the number of as) must be remembered. Regular lan-

guages are also closed under intersection, so if R_1 and R_2 are regular then so is $R_1 \cap R_2$. Consider the language $\mathcal{L}(a^*b^*)$, which contains all strings consisting of some number of as followed by some number of bs. This is clearly regular. However,

$$\mathcal{L}((ab)^{\odot}) \cap \mathcal{L}(a^*b^*) = \{a^n b^n \mid n \in \mathbb{N}\}$$

since the language $\mathcal{L}((ab)^{\odot})$ only matches strings with equally many as and bs. As such, since $\{a^n b^n \mid n \in \mathbb{N}\}$ is not regular it follows that $(ab)^{\odot}$ cannot be regular either. Notice that in terms of the sketched grammars above this corresponds to the case where arbitrarily many non-terminals may be attached to the tracked position, which would create an infinite grammar if the product construction above was attempted.

2.5 Shuffle Expressions and Concurrent Finite State Automata

The formalism that these sketched grammars are trying to imitate is Concurrent Finite State Automata, one of the main subjects of Paper I. These can represent all the languages that can be represented by shuffle expressions, in the way the previous sections sketched. They can, however, represent even more languages using one special trick: as was shown in the grammar in Figure 2.8 they are able to build "stacks" of non-terminals, where only the bottom one can be used to apply rules. By building these stacks arbitrarily high, by having rules that add more and more non-terminal on top, they are able to represent arbitrarily amounts of state (i.e. arbitrarily much information). In this way they are able to represent context-free languages, as well as the shuffle of context-free languages.

However, when this particular trick is removed we reach one of the important milestones. Understanding that the formalism is vaguely sketched so far (next chapter formalizes things further), let us nevertheless call it CFSA and make the following statement.

Theorem 2.11 (Fragment of Theorem 2 in Paper I) A language $\mathcal{L}$ is accepted by some shuffle expression if and only if it is accepted by some CFSA for which there exists a constant k such that no derivation in the CFSA has a stack of non-terminals higher than k. ◇

As such, CFSA capture both the well-known class of shuffle languages (the languages recognized by shuffle expressions), and permit additional language classes based on (possibly fragments of) context-free languages. This opens up questions about membership problems.

2.6 Overview of Relevant Literature

These types of languages featuring shuffle, and many questions relating to them, have been studied in depth and over quite some time. Arguably they started with a definition by S. Ginsburg and E. Spanier in 1965 [GS65]. The shuffle expressions, and the shuffle languages they generate have been the primary focus of this section so far. This is the

name given to regular expressions extended with the binary shuffle operator and unary shuffle closure, a formalism introduced by Gischer [Gis81]. These were in turn based on an 1978 article by Shaw [Sha78] on flow expressions, which were used to model concurrency. The proof that the membership problem for shuffle expressions is NP-complete in general is due to [Bar85, MS94], whereas the proof that the non-uniform case is decidable in polynomial time is due to [JS01].

Shuffle expressions are nowhere near the end of interesting aspects of the shuffle however, even if we restrict ourselves to the focus on membership problems. A very notable example is Warmuth and Hausslers 1984 paper [WH84]. This paper for example demonstrates that the uniform membership problem for the iterated shuffle of a single string is NP-complete. That is, given two strings, w and v, decide whether or not $w \in v \odot v \odot \cdots \odot v$. A precursor to one of the results in Paper I is due to Ogden, Riddle and Rounds, who in a 1978 paper [ORR78] showed that the non-uniform membership problem for the shuffle of two deterministic context-free languages is NP-complete (extended to linear deterministic context-free languages in Paper I).

Some additional examples of interesting literature on shuffle includes a deep study on what is known as shuffle on trajectories [MRS98], where the way the shuffle may happen is in itself controlled by a language, and axiomatization of shuffle [EB98]. For a longer list of references, see the introduction of Paper I.

2.7 CFSA and Context-Free Languages

As noted in Section 2.5 part of the purpose of concurrent finite-state automata is that they permit the modeling of context-free languages, for example the language $\{a^n b^n \mid n \in \mathbb{N}\}$ (i.e. the language where some number of as are followed by the same number of bs), something that is not captured by shuffle expressions. A grammar for this language is shown in Figure 2.12. A derivation in this grammar will simply gen-

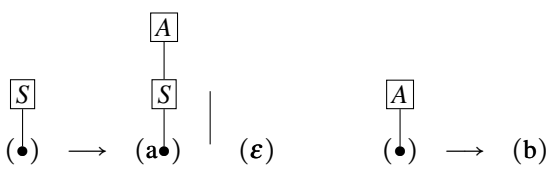


Figure 2.12: A grammar in the CFSA style for the language $\{a^n b^n \mid n \in \mathbb{N}\}$.

erate some number of as while stacking up equally many A non-terminals, then when the S is finally replaced by ε the A non-terminals drop down and each successively generates a b. In this way the (non-shuffle) language is generated. Effectively the CFSA simulates a push-down automaton.

We can easily shuffle two context-free languages in this way, by simply taking grammars of the style of Figure 2.12 and generating their initial non-terminal (now suitably renamed) attached to the same position using a new initial non-terminal rule. This type of language, mixing context-free languages and shuffle, are of some practi-

cal interest, so Paper I studies this type of situation in some depth.

In fact, where shuffle expressions are regular expressions with the two shuffle operators added, it is instructive to view general CFSA as context-free languages with the addition of the binary shuffle operator. This part requires knowledge of context-free grammars, see e.g. [HMU03]. Consider the right-most rule in Figure 2.13, which showcases all the features of CFSA. Then consider the context-free grammar which

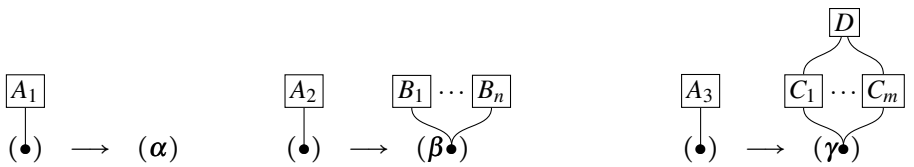


Figure 2.13: The three possible types of rules in our sketched variation of CFSA where $\alpha, \beta, \gamma \in \Sigma \cup \{\varepsilon\}$. The right-most exhibits all features, where the two first are only differentiated in that some parts don't exist.

produces strings over the alphabet $\Sigma \cup \{\odot,), (\}$ by rewriting the CFSA rules in the way shown in Table 2.14. Constructing a context-free grammar in this way, starting from a

Table 2.14: Context-free rules for the CFSA rule in Figure 2.13.

First rule	$A_1 \to \alpha$
Second rule	$A_2 \to \beta(B_1 \odot \cdots \odot B_n)$
Third rule	$A_3 \to \gamma(C_1 \odot \cdots \odot C_m)D$

CFSA A, one gets a context-free language L containing shuffle expressions which are such that $\mathcal{L}(A) = \bigcup\{\mathcal{L}(e) \mid e \in L\}$. That is, when the result of evaluating all the shuffle expressions in L are unioned together we arrive at the language generated by A.

This should serve to illustrate that all languages generated by CFSA can be viewed as "disordered" context-free languages. The above procedure generates a characterizing context-free language, which specifies which strings are to be shuffled together to produce strings in the original CFSA. As such, for example the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ cannot be generated by a CFSA, as it is not context-free, nor can one arrive at it by relaxing the order of substrings in a context-free language.

2.8 Membership Problems

The membership problem for these shuffle formalisms should be divided into two parts; the membership problem for shuffle expressions, which do not feature the context-free abilities of full CFSA, and the one for full CFSA.

2.8.1 The Membership Problems for Shuffle Expressions

The membership problem for shuffle expressions is already a fairly complex question. There is a sizable body of literature, and Paper I studies one fragment of the problem.

- The non-uniform membership problem is decidable in polynomial time [JS01]. The algorithm relies on permitting each symbol read (or generated) to produce some large number of potential states, which limits the complexity in terms of the length of the string but explodes the complexity in terms of the size of the expression.
- Unsurprisingly, in view of the above, the general uniform membership problem is NP-complete [Bar85, MS94].

These two pieces paint a fairly clear picture; if we wish to check membership (or parse) a string with respect to a shuffle expression it can be done reasonably efficiently if the string is much larger than the shuffle expression. However, this does not reveal the exact way in which the complexity depends on the expression. Notably, regular expressions are (trivially) shuffle expressions, and for regular expressions the uniform membership problem is not very difficult. Paper I explores how the structure of the expression affects the complexity of the problem. See Section 2.9.

2.8.2 The Membership Problems for General CFSA

The membership problem for CFSA is NP-hard even in very restrictive cases, such as where at most two non-terminals are ever attached to a position. It may therefore be surprising that the problem is *in* NP. The overall construction hinges on limiting the size of the trees of non-terminals generated by parsing a certain string, which relies on a careful case-by-case analysis of symmetries in how non-terminals may be generated. This means that even if far more (seemingly) complex CFSA are considered the problem does not become substantially harder. All of this is treated in Paper I, which Section 2.9 now takes a deeper look into.

2.9 Contributions In the Area of Shuffle☆

This section provides, as denoted by the star, a slightly more formal treatment of the contributions to the area of shuffle that have been made in (the papers included in) this work. We need some additional definitions to start with.

2.9.1 Definitions and Notation

Let $\mathbb{N}_+$ denote $\mathbb{N} \setminus \{0\}$. A *tree* with labels from an alphabet Σ is a function $t{:}N \to \Sigma$, where $N \subseteq \mathbb{N}_+^*$ is a set of nodes which are such that

- N is prefix-closed, i.e., for every $v \in N$ and $i \in \mathbb{N}_+$, $vi \in N$ implies that $v \in N$, and
- N is closed under less-than, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $v(i+1) \in N$ implies $vi \in N$.

Let $N(t)$ denote the set of nodes in the tree t. The root of the tree is the node ε, and vi is the ith child of the node v. t/v denotes the tree with $N(t/v) = \{w \in \mathbb{N}_+^* \mid vw \in N(t)\}$ and $(t/v)(w) = t(vw)$ for all $w \in N(t/v)$. The empty tree, denoted t_ε, is a special case, since $N(t_\varepsilon) = \varnothing$ it cannot be a subtree of another tree. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$. The tree $\alpha[\,]$ may be abbreviated by α. Given an alphabet Σ, the set of all trees of the form $t : N \to \Sigma$ is denoted by T_Σ. For trees t, t' and $v \in N(t)$ let $t_{v \mapsto t'}$ be the tree resulting from replacing the node at v by t' in t. That is, $t_{\varepsilon \mapsto t'} = t'$, and $t_{iv \mapsto t'} = t(\varepsilon)[t/1, \ldots, (t/i-1), (t/i)_{v \mapsto t'}, (t/i+1), \ldots, t/n]$ for $iv \in N(t)$ and $i \in \mathbb{N}_+$. For $t_{v \mapsto t_\varepsilon}$ the subtree at v is deleted (e.g. $\alpha[t_1, t_2, t_3]_{2 \mapsto t_\varepsilon} = \alpha[t_1, t_3]$).

2.9.2 Concurrent Finite State Automata

With this we can make a formal definition of the concurrent finite state automata already sketched. These automata are the subject at the heart of Paper I.

Definition 2.15 A *concurrent finite state automaton* is a tuple $A = (Q, \Sigma, S, \delta)$ where Q is a finite set of states, Σ is the input alphabet, $S \in Q$ is the initial state, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times T_Q$ are the rules.

A derivation in A is a sequence $t_1, \ldots, t_n \in T_Q$ such that $t_1 = S[\,]$ and $t_n = t_\varepsilon$. For each $i < n$ the step from $t = t_i$ to $t' = t_{i+1}$ is such that there is some $(q, \alpha, t'') \in \delta$ and $v \in N(t_i)$ such that $t/v = q[\,]$ and $t' = t_{v \mapsto t''}$. Applying this rule reads the symbol α (nothing if $\alpha = \varepsilon$). $\mathcal{L}(A)$ is the set of all strings that can be read this way.

We only permit four types of rules in δ. Deleting rules of the form $(q, \varepsilon, t_\varepsilon) \in \delta$. Horizontal rules of the form $(q, \alpha, q'[\,]) \in \delta$. Vertical rules of the forms $(q, \alpha, q'[p_1]) \in \delta$ and $(q, \alpha, q'[p_1, p_2]) \in \delta$. Finally the closure rules, where $(q, \alpha, q'[p_1, \ldots, p_1]) \in \delta$ for every number of repetitions of p_1s, greater or equal to zero. ◇

We treat the in practice infinite set of rules for the closure rules as a schema (i.e. they count as a constant number of rules for the purposes of defining the size of the automaton).

Using this definition it should be easy to see how the rules in Figure 2.13 can be constructed. The graphical rules cheat by ignoring the possibility that $\alpha = \varepsilon$, while permitting e.g. generating siblings without a root (effectively having rules $(q, \alpha, p_1 p_2)$), but it is trivial to add an additional state that serves as root for the subtree with only a deleting rule defined.

Notice that the rules overlap a bit, in that the closure schema is unnecessary if we are allowed to replace $(q, \alpha, q'[p_1, \ldots, p_1])$ with $(q, \alpha, q'[q'', p_1])$ where q'' is a new state with only two rules, $(q'', \varepsilon, t_\varepsilon)$ and $(q'', \varepsilon, q''[q'', p_1])$. However, the context-free languages are precisely those that can be recognized by a CFSA where every $(q, \alpha, t) \in \delta$ has no node with more than one child in t, and we often wish to syntactically restrict CFSA to not permit context-free languages, recreating the shuffle languages. We do this as follows: a configuration is acyclic if for every $v \in N(t)$ it holds that $t(v)$ does not occur in t/vi for any i, the shuffle languages are then precisely the CFSA where all configurations are acyclic. The closure-free shuffle languages are those recognizable by a CFSA with a finite (schema-free) δ and all reachable configurations acyclic.

2.9.3 Properties of CFSA

Paper I proves a number of relevant properties about CFSA. Notably they are closed under union, concatenation, Kleene closure, shuffle, and shuffle closure (i.e., if A and A' are CFSA then there exists a CFSA A'' such that e.g. $\mathcal{L}(A'') = \mathcal{L}(A) \odot \mathcal{L}(A')$), but not under complementation or intersection (so there exists some A and A' such that there exists no CFSA recognizing the language e.g. $\mathcal{L}(A) \cap \mathcal{L}(A')$). Emptiness of CFSA is decidable in polynomial time, and the CFSA generate only context-sensitive languages.

2.9.4 Membership Testing CFSA

Membership in general CFSA. With this done we can consider uniform membership testing for general CFSA, one of the core results of Paper I. Since even a severely restricted case of CFSA already have a NP-complete uniform membership problem [Bar85, MS94], which serves as a lower bound, it is a pleasant surprise that the general problem is *in* NP, as the restricted cases appear so relatively restrictive. A non-deterministic polynomial time algorithm can simply guess which rules to apply to accept a string, as long as the number of rules necessary (i.e. the sequence $t_1, \ldots, t_n$ in Definition 2.15) is polynomial in the length of the string. The only way this might *not* happen is if a lot of ε-rules are required. A simple polynomial rewriting procedure on A solves this, based on statements such as "if rules from δ can rewrite $q[\,]$ into $q'[\,]$ without reading a symbol, include $(q, \varepsilon, q'[\,])$ in δ." This ensures that if a derivation of a string exists in A then a *short* one exists.

Membership in the shuffle of shuffle languages and context-free languages. The CFSA model goes on to be used to prove a number of other membership problem results. One interesting case is the shuffle of a shuffle language and a context-free language, i.e., membership for the CFSA where every configuration tree (except the first one and the last one where things are getting set up and dismantled) is of the form $q[t_1, t_2]$ where t_1 is acyclic and $N(t_2) \subset 1^*$ (that is, no node in t_2 has more than one child). This proof is rather more involved, and relies on finding a number of symmetries in the way the tree corresponding to the shuffle language (i.e. t_1 here) can behave. Notably it relies on defining an equivalence relation on nodes in the tree, i.e., if we have $t(v) = t(v')$ what we do to v and v' is interchangeable. Most notably, if we in two places apply a rule schema $(q, \alpha, q'[p_1, \ldots, p_1])$ there is *no point* in generating p_1 instances in both places, we might as well pick one of the places and generate *all* the instances of p_1 necessary. In fact, in the procedure we can just remember "as long as this node is still here we can assume we have any necessary number of p_1 instances". In this way the number of possibilities are limited in such a way that a Cocke-Younger-Kasami-style table can be established for parsing. While polynomial the degree of the polynomial is very substantial, an efficient algorithm is left as future work.

The hardness of context-free shuffles. Another of the core results of Paper I is a proof that there exist two deterministic linear context-free (DLCF) languages L and L'

such that the membership problem for $L \odot L'$ is NP-complete. That is, the non-uniform membership problem for the shuffle of two DLCF languages is NP-complete. The proof relies on the following. We can construct a DLCF language L which consists of strings of the following form:

$$\underbrace{[0][1]\cdots[1][1]}_{C_1}\$\underbrace{[0][1]\cdots[1][1]}_{C_2}\$\cdots\$\underbrace{[0][1]\cdots[1][0]}_{C_2'}\$\underbrace{[0][1]\cdots[1][1]}_{C_1'}$$

where each bit-string is a polynomial-length Turing machine configuration, and C_1' is the (reversed) configuration the Turing machine reaches taking *one step* from C_1, and similarly C_2' is one step from C_2 (and so on nested inwards). The rules of the Turing machine are encoded in L. The language class is not powerful enough to relate C_1 and C_2, all it can do by itself is take a single step. We can however also construct a DLCF language L' which recognizes all strings

$$\$\underbrace{[0][1]\cdots[0][1]}_{P_1}\$\underbrace{[1][1]\cdots[1][1]}_{P_2}\$\cdots\$\underbrace{[1][1]\cdots[1][1]}_{P_2'}\$\underbrace{[1][0]\cdots[1][0]}_{P_1'}$$

which are such that P_1' is P_1 reversed, and P_2' is P_2 reversed, and so on inwards. At the center there is one extra string of the form $([0]|[1])^*$, entirely arbitrary. Now construct the string

$$\underbrace{[0]\cdots[0]}_{I}\$[[01]]\cdots[[01]]\$\$\cdots\$\$[[01]]\cdots[[01]]$$

where I is filled with the initial Turing machine configuration we are interested in. Then check if this string is in $L \odot L'$. What will happen is that L and L' will have to "share" every $[[01]]\cdots[[01]]$ substring (since neither can by itself produce e.g. $[[$), each producing half the brackets and binary digits, forcing the other to produce its *complement*. The initial I must be produced by L, as L' requires a leading \$, which makes L produce the result of taking the first step of the Turing machine in the last $[[01]]\cdots[[01]]$ section, which leaves the complement for L' to produce in the last section, which will make it produce the complement in the first $[[01]]\cdots[[01]]$ section, forcing L to produce the *same* configuration in that first sectiont that it produced in the last section. This makes it produce the result of taking another computation step in the second-to-last $[[01]]\cdots[[01]]$ section, which L' then copies, and so on. In this way the shuffle will cooperate to perform an arbitrary (non-deterministic) Turing machine computation for polynomially many steps, making the membership problem NP-hard. This is non-uniform as the Turing machine coded in L may be one of the universal machines, which reads its program from the input I.

2.9.5 The rest of Paper I.

Paper I has a number of further results, including a fixed parameter analysis of parsing shuffle expressions with the number of shuffle operators which is discussed in brief in Section 3.5.1. In addition the paper discusses the uniform membership problem for

the shuffle of a context-free language and a regular language. That is, a context-free grammar G, a finite automaton A and a string w are given as input, and the decision problem is checking whether $w \in \mathcal{L}(G) \odot \mathcal{L}(A)$). An important point in this context is that $\mathcal{L}(G) \odot \mathcal{L}(A)$ is a context-free language for all G and A. This can be shown by a simple product construction. This, however, raises a question discussed in another paper.

2.9.6 Language Class Impact of Shuffle

Paper V also considers shuffle, but here the question is of a more abstract nature. The claim studied is, for two context-free languages $L \subseteq \Sigma^*$ and $L' \subseteq \Gamma^*$ (with $\Sigma \cap \Gamma = \varnothing$) is $L \odot L' \notin \mathcal{CF}$ unless either $L \in \mathcal{Reg}$ or $L' \in \mathcal{Reg}$? That is, if the shuffle of two context-free languages is context-free must one of the languages be regular? The author conjectures that this is indeed the case, but Paper V gives only a conditional and partial proof.

CHAPTER 3

Synchronized Substrings in Languages

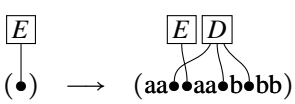
In this chapter we take a look at what can be described as formalisms with synchronized substrings. A single sequence of derivation decisions which (may) have effects in several places of a string. This is most easily illustrated by extending our running sketched formalism to generate such languages.

3.1 Sketching a Synchronized Substrings Formalism

3.1.1 The Graphical Intuition

In this section the grammars introduced in Figure 1.3 will be extended in a *different way* from the preceding shuffle chapter. In this new grammatical formalism there may *not* be more than one non-terminal attached to a position (i.e. to a bullet), *nor* may we have non-terminals depend on each other. That is, the "stacking" of non-terminals of Figure 2.8 is no longer permitted.

The new grammatical formalism for this chapter instead generalize the regular grammars in some new ways, which will pave the way to rules of the following form.



- Positions (i.e. bullets) may now occur anywhere in the string, not just at the end. There may be any number of positions on the right-hand side of rules.
- Each non-terminal may be attached to multiple positions. We say that the non-terminal tracks, or controls, those positions. This in turn means that the left-hand sides may also contain multiple positions (the number controlled by the non-terminal being replaced).

We assume that each non-terminal always tracks the same number of positions (so if A tracks 3 positions in one rule it will always track 3 positions). See Figure 3.1 for a first example of a grammar of this new kind. An example derivation using this grammar is shown in Figure 3.2.

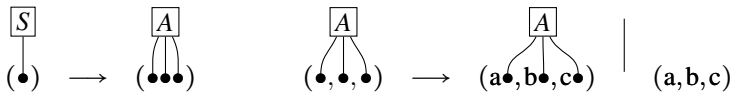


Figure 3.1: An example of a grammar of the synchronized substring variety. The initial non-terminal *S*, which tracks a single position, generates an instance of the non-terminal *A*, which tracks three positions, inserted as a string at the position which *S* was previously tracking (notice that this is *not* the same as attaching them all to that position, they are ordered and distinct in the resulting string). *A* has two rules, the first generates an *a* in the first position, a *b* in the second and a *c* in the third, while generating a new *A* tracking the positions just after each of the newly generated symbols. The last rule generates the same symbols but creates no new *A*.

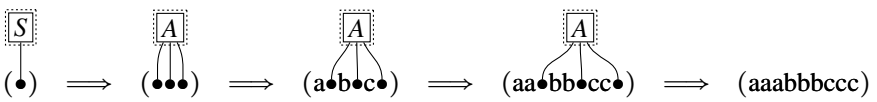
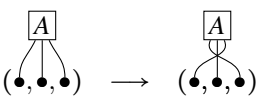


Figure 3.2: A derivation of the string "aaabbbccc" using the grammar in Figure 3.1. Notice that even though *A* tracks multiple positions there will never be commas in the derivation like there is in the grammar, the positions will instead be interspersed with real symbols in a contiguous string. Applying a rule places new substrings in some positions, and these substrings may themselves contain positions.

Notice that this formalism features ordering in the positions that the non-terminals track. Consider for example adding the following rule to the grammar in Figure 3.1.



This then permits derivations like the one shown in Figure 3.3, and more generally it permits deriving strings of the form "aacacbbbbbccaca", containing the same number of "a"s, "b"s and "c"s, but the first and last section are the same sequence with "a"s replaced with "c"s and vice versa.

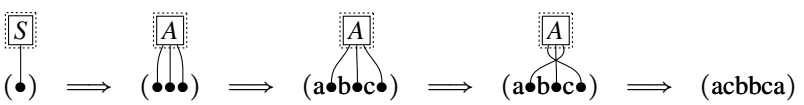


Figure 3.3: A derivation of the string "acbbca" using the grammar in Figure 3.1 extended with a rule which switches the first and third position tracked by the *A*.

3.1.2 Revisiting the Mapped Copies of Example 1.1

Example 1.1 illustrates a trivial case of synchronized substrings formalisms, where a sequence of symbols is chosen, and two different symbol mappings create two different strings, which are concatenated to produce an output string. Let us recall it here.

Example 3.4 (Mappings of copy-languages) Given two mappings σ_1, σ_2 from $\{a,b\}$ to arbitrary strings and a string w decide whether there exists some $\alpha_1, \ldots, \alpha_n \in \{a,b\}$ such that $\sigma_1(\alpha_1)\cdots\sigma_1(\alpha_n)\cdot\sigma_2(\alpha_1)\cdots\sigma_2(\alpha_n) = w$. ◇

Let us look at how

1. we can model this type of language by a grammar, and,
2. parsing may be performed, in both the uniform and non-uniform case.

3.1.3 Grammars for the Mapped Copy Languages

Here we have two alphabets, the "internal" alphabet $\Gamma = \{a,b\}$ as well as the usual Σ. In addition we have two mappings from Γ to strings in Σ. Let $w_a = \sigma_1(a)$, $w_b = \sigma_1(b)$, $v_a = \sigma_2(a)$ and $v_b = \sigma_2(b)$. Then the grammar in Figure 3.5 generates the language of the strings that the procedure in Example 1.1 yields.

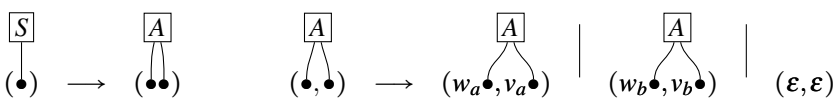


Figure 3.5: A synchronized substring-type grammar for the language that the procedure sketched in Example 1.1 can produce. Notice that w_a, v_a, w_b and v_b are strings derived from the mappings σ_1 and σ_2, rather than symbols in their own right.

3.1.4 Parsing for the Mapped Copy Languages

Let us consider the uniform parsing problem for this class of grammars (i.e., those that can be generated by some choice of σ_1 and σ_2 in the above construction). We can divide the parsing problem into two parts:

1. We need to find the position at which the concatenation happens. That is, let G be the grammar constructed as in Figure 3.5 for some σ_1 and σ_2, then, to decide if some w belongs to $\mathcal{L}(G)$ we need to tell if there is some way to divide w into two, $w = xy$, such that $\sigma_1(v) = x$ and $\sigma_2(v) = y$ for some $v \in \{a,b\}^*$.
2. The second part is finding the actual $v \in \{a,b\}^*$.

Solving the second part effectively solves the first, in the sense that if we are *given* v we will be able to tell where the concatenation happens by simply computing $\sigma_1(v)$ and $\sigma_2(v)$.

However, it might be easier to find v if the point of concatenation is found. We are in fact primarily concerned about whether parsing can be done in polynomial time or not, and if we can compute v in polynomial time given the point of concatenation we can solve the whole problem in polynomial time, as there are only linearly many positions at which the concatenation may occur. That is, we can simply for each possible way of dividing w into xy try to compute a v for this x and y. This exhaustive search at most makes the membership problem linearly more expensive.

The full algorithm for this toy example is in fact fairly simple. It will, however, serve to illustrate the more general algorithms later, where the directed graph construction will be replaced by something similar but more advanced.

Algorithm 3.6 (Parsing for Example 1.1)

```
1:  function ParseCopyMap(string w ∈ Σ*, σ1 : {a,b} → Σ*, σ2 : {a,b} → Σ*)
2:      let α1···αn = w (i.e., each αi is a symbol in Σ).
3:      let G be a digraph with nodes {(p,q) | p,q ∈ {0,...,n}} and no edges.
4:      for p,q,p′,q′ ∈ {0,...,n} do
5:          if (σ1(a) = αp+1···αp′ and σ2(a) = αq+1···αq′) or
6:             (σ1(b) = αp+1···αp′ and σ2(b) = αq+1···αq′) then
7:              add an edge from (p,q) to (p′,q′) to G
8:          end if
9:      end for
10:     for i ∈ {0,...,n} do
11:         if Reachable(G, (0,i), (i,n)) = True then
12:             return True
13:         end if
14:     end for
15:     return False
16: end function
```

Reachable is a function which takes a graph G and two nodes v and w and checks if w can be reached from v following the edges. Notice that we abuse the subscripts in the algorithm, so $\alpha_{p+1}\cdots\alpha_{p'}$ for $p \geq p'$ will be an empty string.

To quickly outline the algorithm, in lines 4–9 the graph G is constructed in such a way that a node (p',q') is only reachable from (p,q) if the substrings $\alpha_{p+1}\cdots\alpha_{p'}$ can be generated by $\sigma_1(v)$ and $\alpha_{q+1}\cdots\alpha_{q'}$ can be generated by $\sigma_2(v)$ for some common v. Once this graph is constructed lines 10–14 simply test all ways to cut the initial string into two pieces and checks on the graph if the resulting two strings can correspond to a common original string mapped through σ_1 and σ_2.

Notice that the graph will be polynomial in the size of the string to be parsed, and computing reachability on a directed graph is very cheap. Also notice that this algorithm as written is just a membership test, but making it parsing amounts to simply outputting the path in G found when line 11 succeeds.

3.2 The Broader World of Mildly Context-Sensitive Languages

The above may seem like trivialities, but it appears to be at the core of the difficulty in deciding membership for the general class of formalisms along these lines. The formalism sketched here (exemplified by the grammar in Figure 3.5) is intended to imitate a hyperedge replacement grammar (see e.g. [DHK97]) generating a string, but that formalism is equivalent to a large class of other formalisms.

3.2.1 The Mildly Context-Sensitive Category

All the formalisms discussed in this chapter fall within the category "mildly context-sensitive", defined by Aravind Joshi in [Jos85]. A language class $\mathcal{L}$ is defined by Joshi to be mildly context-sensitive if and only if all the following hold.

1. At least one language in $\mathcal{L}$ features *limited cross-serial dependencies*.
2. All languages L in $\mathcal{L}$ have a semi-linear set $\{|w| \mid w \in L\}$. That is, the lengths of strings in the language form a union of a finite number of linear sequences, $\{s_1 + ik_1 \mid i \in \mathbb{N}\} \cup \cdots \cup \{s_n + ik_n \mid i \in \mathbb{N}\}$.

In addition the following two requirements are implicit but clearly required in [Jos85]

3. All $L \in \mathcal{CF}$ are in $\mathcal{L}$, that is, a mildly context-sensitive formalism must be able to represent all context-free languages (recall Section 2.7).
4. The non-uniform membership problem for languages in $\mathcal{L}$ is decidable in polynomial time.

Requirement 1 needs some further explanation. It refers to a certain type of substring synchronization that Joshi derives from the tree-adjoining grammar formalism that he uses in that paper. The description is fairly involved, but one key detail is that languages of the form $a^n b^n c^n$ may be in such a class, but $a^n b^n a^n b^n a^n b^n \cdots$ may not. This statement may be transferred to the formalism we have sketched by noting that for every such grammar there exists some constant k such that no non-terminal tracks more than k positions, which makes it impossible to generate e.g.

$$\underbrace{a^n b^n \cdots a^n b^n}_{k+1 \text{ copies}}.$$

3.2.2 The Mildly Context-Sensitive Classes

There are at least two different classes of languages with published formalisms that fit clearly into the mildly context-sensitive definition.

1. The first is the motivating class, into which tree-adjoining grammars [JLT75] which Joshi used when first defining the category, fall. All the following formalisms are equivalent [JSW90] (that is, they define the same language class): linear indexed grammars [Gaz88], combinatorial categorial grammars [Ste87] and head grammars [Pol84].

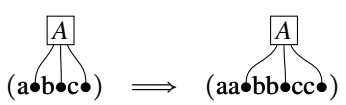
2. The second class still fulfills all the requirements outlined by Joshi, but is strictly more powerful (i.e. every language that can be generated by e.g. a head grammar can be generated by any of these formalisms). These formalisms include linear context-free rewriting systems [Wei92][1], deterministic tree-walking transducers [Wei92], multicomponent tree-adjoining grammars [Jos85, Wei88], multiple context-free grammars [SMFK91, Göt08], simple range concatenation grammars [Bou98, Bou04, BN01, VdlC02] and string-generating hyperedge replacement grammars [Hab92, DHK97].

It is interesting to note that while the mildly context-sensitive definition requires a non-uniform membership problem (i.e. the membership problem where the string, but not the grammar/automaton, is included in the input, recall Definitions 1.7 and 1.8) that is solvable in polynomial time, all the listed formalisms above have an NP-hard uniform membership problem. The way that the grammars perform concatenation, notably how many positions each non-terminal keeps track of (or, in Joshis terminology, the extent of the cross-serial dependencies), play a big part in how difficult the uniform membership problem becomes.

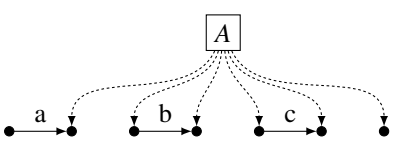
Going through all of these formalisms is not time well spent for an introductory text like this, but in the next section we will make the connection between the sketched formalism of Figure 3.1 and string-generating hyperedge replacement grammars.

3.3 String-Generating Hyperedge Replacement Grammars

The formalism sketched in Figures 3.1–3.5 is more or less a direct copy of hyperedge replacement grammars tuned for string generation. This formalism constructs a directed graph by having *hyperedges* (that is, edges that connect an arbitrary number of nodes) labeled with non-terminal symbols, and having rules that replace these by new subgraphs connected to the nodes the hyperedge was connected to. So, the rule application (using a rule from Figure 3.1)

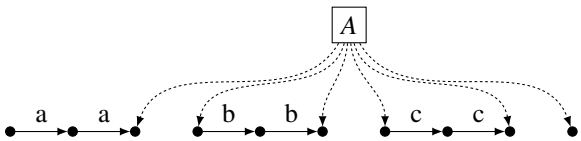


actually corresponds to rewriting the directed graph



where the box labeled by A now represents a hyperedge which is connected to 6 nodes, into this graph

[1] References are for the most part not to the original definitions, but rather to sources where they are described and related to the broader class at hand.



by the hyperedge labeled A being replaced by attaching new nodes and edges to the positions the original hyperedge was attached to, and attaching a new hyperedge (also labeled A). To make this perfectly formal we also need to number the nodes, or otherwise somehow distinguish between them, which we manage to avoid graphically in the string case by just keeping track of things left-to-right.

Notice that while the non-uniform membership problem is polynomial for *string-generating* hyperedge replacement grammars it very quickly becomes NP-hard when the grammar are allowed to generate graphs even a little bit more complex than these string-representing directed chains. In fact, if the grammar is allowed to make *multiple* chains, that is, creating a graph consisting of the union of directed chains, by simply having the hyperedge replacement rules leaving pieces unconnected, the non-uniform membership problem becomes NP-complete [LW87].

In addition note that for a hyperedge replacement grammar to generate a string it will necessarily have to keep track of both sides of each "gap" corresponding to a position, as is sketched in the above figures. If it loses track of a node that is supposed to be internal to the string it becomes impossible to later join it up to the other parts generated, and the graph becomes a set of multiple chains.

We will next take a look at a general non-uniform parsing algorithm for string-generating hyperedge replacement grammars (the informal flavor used here). This construction is from 2001 by Bertsch and Nederhof [BN01] (this is not the earliest or most efficient parsing algorithm of this type, but a straightforward and clear one).

3.4 Deciding the Membership Problem

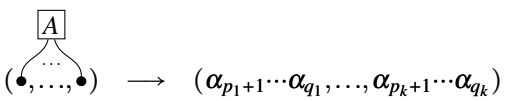
3.4.1 Deciding Non-Uniform Membership

Origins of the Algorithm The construction from [BN01] is here modified a bit for clarity and to fit into the approach we use. In its original form the algorithm takes the grammar G (for which membership should be decided) and a string w, and from these two constructs a new grammar G', which is empty if and only if $w \notin \mathcal{L}(G)$. In this way it reduces the problem of deciding membership for one grammar to the problem of deciding emptiness for another. More specifically the grammar G is one of the mildly context-sensitive formalisms (the algorithm is originally defined in terms of *Range Concatenation Grammars*, but here we opt to continue with the equivalent string-generating hyperedge replacement grammars) and the constructed grammar G' is a context-free grammar, for which emptiness testing (i.e. computing whether $\mathcal{L}(G') = \varnothing$) is very easy. However, as context-free grammars are a subset of the hyperedge replacement grammars (and emptiness testing is just as easy for hyperedge replacement grammars) we will not differentiate between the formalisms.

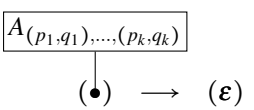
Deciding Emptiness Emptiness-checking a hyperedge replacement grammar is very easy. Start by letting all non-terminals be *unmarked*. For each rule, if the left-hand is unmarked but the right-hand side contains no unmarked non-terminals mark the non-terminal on the left-hand side and start over the looping through the rules from the first one. If we make it through all the rules without marking a non-terminal we are done, and the language generated by the grammar is empty if and only if S is still unmarked. This algorithm is clearly in $\mathcal{O}(n^2)$ where n is the number of rules, as it is a loop that is restarted at most n times (sooner or later all the non-terminals have been marked).

Reducing Membership to Emptiness It is time to solve the membership problem for string-generating hyperedge replacement grammars. Let G be the grammar, and $\alpha_1 \cdots \alpha_n$ ($\alpha_i \in \Sigma$ for each i) the string for which we wish to check whether $\alpha_1 \cdots \alpha_n \in \mathcal{L}(G)$. To decide this we will construct a new grammar G' such that $\mathcal{L}(G') \neq \emptyset$ if and only if $\alpha_1 \cdots \alpha_n \in \mathcal{L}(G)$ (specifically, $\mathcal{L}(G') = \{\varepsilon\}$ otherwise). The construction of G' should be reminiscent of the construction of the graph in Algorithm 3.6.

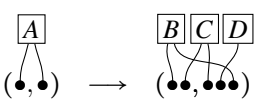
G' will be constructed in such a way that if G contains a non-terminal A which controls k positions (i.e. there are k positions on the left hand side of rules for A) then G' contains one non-terminal $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ for each $p_1, q_1, \ldots\ldots, p_k, q_k \in \{0, \ldots, n\}$, such that $p_1 \leq q_1, \ldots, p_k \leq q_k$. The logic will be, if the non-terminal $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ can generate *any* strings, then the non-terminal A in the original grammar G is able to generate the strings $\alpha_{p_1} \cdots \alpha_{q_1}, \ldots, \alpha_{p_k} \cdots \alpha_{q_k}$. The most direct rule to include in G' then becomes that if



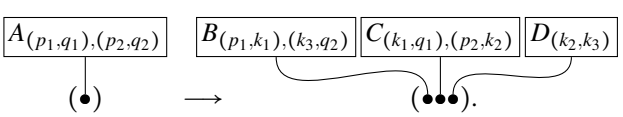
is a rule in G then



is a rule in G'. That is; if A can generate the substrings $\alpha_{p_1+1} \cdots \alpha_{q_1}$ through $\alpha_{p_k+1} \cdots \alpha_{q_k}$, then $A_{(p_1,q_1),\ldots,(p_k,q_k)}$ can generate the empty string (we could select any string as only emptiness is of interest). Similarly, for example, if there is a rule in G of the form

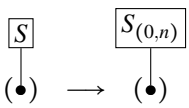


then, for all $p_1, q_1, p_2, q_2 \in \{0, n\}$, and $k_1, k_2, k_3 \in \{0, n\}$, such that $p_1 \leq k_1 \leq q_1$, and $p_2 \leq k_2, k_3 \leq q_2$ there is a rule



That is, the p_1, q_1, p_2, q_2 decide the substrings checked by the left-hand side, and k_1 is the position between p_1 and q_1 where the substring goes from being generated by the B and the C, and similarly for k_2 and k_3 with the two concatenation points in the second substring. This generalizes to arbitrary rules in the obvious way, when non-terminals and symbols are mixed one additionally needs to check that the symbols generated correspond correctly to the symbols in $\alpha_1 \cdots \alpha_n$.

As such, if each of the B, C and D non-terminals can generate their respective substrings then the right-hand side can generate the empty string, meaning that the A can generate the whole. Finally, add the rule



to G', that is, the initial non-terminal goes to the non-terminal that checks if the non-terminal S in G can generate the substring $\alpha_{0+1} \cdots \alpha_n$, i.e., the whole string.

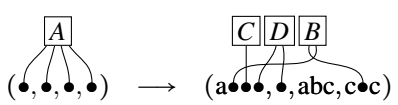
Following this procedure G' will be able to generate the empty string if and only if G can generate the string $\alpha_1 \cdots \alpha_n$. It should be clear that the size of G' is on the order of $\mathcal{O}(n^c)$ where c is polynomial in the size of the grammar G. Notably c increases with the number of positions tracked by non-terminals in G.

However, we were considering the non-uniform membership problem, and as such we view G as a constant, which in turn means that c is viewed as a constant. It follows that $\mathcal{O}(n^c)$ is a polynomial, and, as established above, deciding emptiness is polynomial, making for a membership algorithm that is polynomial.

3.4.2 Deciding Uniform Membership

Deciding the uniform membership problem for our sketched grammatical formalism appears to be extremely hard, a proof that LCFRS parsing is PSPACE-hard is given in [KNSK92]. This makes it extremely unlikely that an efficient algorithm exists. The best known algorithms (see e.g. [SMFK91] and Paper II for more references) for the problem are in $\mathcal{O}(mn^{f(r+1)})$ where m is the size of the grammar, n the size of the input string, and f and r are two very specific properties of the grammar, the *fan-out* and the *rank* of the grammar respectively. Before we get further into the explanation it is important to note that since f and r are values depending on the grammar the algorithm is in $\mathcal{O}(n^m)$ as well, that is, the length of the string raised to the size of the grammar. However, it turns out that in practice the fan-out and rank tend to be small compared to the size of the full grammar, so it is a useful distinction to separate them out.

In short, the rank of a grammar is the maximum number of non-terminals occurring on the right hand side of a rule. The fan-out is the maximum number of positions a non-terminal tracks. So for example a grammar G containing the rule



implies that it has rank at least 3, since the right-hand side has three non-terminals, and fan-out at least 4, since A controls four positions. These are "at least" since other rules may contain more non-terminals or more positions tracked.

3.4.3 On the Edge Between Non-Uniform and Uniform

So far we have seen that the problem is polynomial when the grammar is left out of the input entirely, and an algorithm that is unquestionably exponential when the grammar is included. However, this distinction, where some aspects of the grammar is separated out to illustrate that it is not exponential in the "worst" way, is a bit blunt and imprecise a way of viewing the complexity. For example, if the total number of rules were in the exponent many practical grammars would have very problematic running times, whereas if the number of symbols in the alphabet is the part in the exponent things may not be nearly as problematic.

For a deeper look in this direction, we now give a slightly deeper summary of the results in Paper II, with explanations of some of the supporting theory.

3.5 Contributions in Fixed Parameter Analysis of Mildly Context-Sensitive Languages☆

We take this opportunity to briefly explain fixed parameter complexity, as it is necessary to appreciate the contents of Paper II, and may not be common knowledge. See e.g. [FG06] for a full introduction.

3.5.1 Preliminaries in Fixed Parameter Tractability

In classical complexity theory a problem may be viewed as a set of all *positive instances* $P \subseteq I$, where I is the set of all *possible instances*. For example, we may have I be all graphs and P be the set of all graphs which have a Hamiltonian path. A decision procedure for the problem is then a program that computes a function $a : I \to \{yes, no\}$ such that $P = \{p \in I \mid a(p) = yes\}$. The running time of the program is then analyzed as a function in $|p|$. For two problems $P \subseteq I$ and $P' \subseteq I'$ a polynomial time reduction is a function $r : I \to I'$, computable in polynomial time such that $p \in P \Leftrightarrow r(p) \in P'$. A polynomial time reduction r from P to P' implies that the fastest decision procedure for P cannot be more than polynomially slower than the fastest for P'.

A parameterized problem is viewed as a set $P \subseteq I \times \mathbb{N}$, where I is again the set of all problem instances and the integer is what is called the *parameter*. A decision procedure for R again computes a function $a : I \times \mathbb{N} \to \{yes, no\}$, but now the time of deciding $a(p,k)$ is described in both $|p|$ and k. If a runs in time $f(k) \cdot |p|^{\mathcal{O}(1)}$ for *any computable function* $f : \mathbb{N} \to \mathbb{N}$ the problem is said to be fixed-parameter tractable (FPT); that is, intuitively, for a small parameter the problem is basically polynomial. The way the parameter is chosen has a large impact on how the analysis behaves. Notably, taking any problem $P \subseteq I$ and constructing the parameterized problem $\{(p, |p|) \mid p \in P\} \subseteq I \times \mathbb{N}$ yields a fixed-parameter tractable problem for every decidable P. A FPT reduction from $P \subseteq I \times \mathbb{N}$ to $P' \subseteq I' \times \mathbb{N}$ is a program which

computes a function $r : (I \times \mathbb{N}) \to (I' \times \mathbb{N})$ such that, (i) $r(p,k)$ is computable in time $f(k) \cdot |p|^{\mathcal{O}(1)}$ for some computable function f; (ii) $(p,k) \in P \Leftrightarrow r(p,k) \in P'$; and; (iii) there exists a computable function g such that for all $(p',k') = r(p,k)$ it holds that $k' \leq g(k)$.

The parameter is normally chosen as some minor aspect of the problem. A classic case is the vertex cover problem for graphs, which is NP-complete in general, but if one looks for *small* covers (i.e. does this graph of a million vertices have a cover of size five?) it turns out that the problem is easy. Vertex cover is, in fact, a classic problem in the class FPT. That is, the parameterized problem is $P \subseteq \mathbb{G} \times \mathbb{N}$, where $\mathbb{G}$ is the set of all graphs, such that $(G,k) \in P$ if and only if G has a cover of size k. This problem is decidable in time $\mathcal{O}(k|G| + 1.3^k)$, which is excellent for small k. Not all problems work out this well however, and there is a hierarchy of parameterized complexity classes:

$$\mathrm{FPT} \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots \subseteq \mathrm{W}[\mathrm{SAT}] \subseteq \mathrm{W}[\mathrm{P}] \subseteq \mathrm{XP}, \quad \text{where } FPT \subsetneq XP,$$

each of which has some complete (characterizing) problem.

To make a quick revisit to Paper I and Chapter 2 note that there a proof is given which shows that the uniform membership problem for shuffle expressions is $\mathrm{W}[1]$-hard when the parameter is the number of shuffle operators used in the expression. One example of an instance is $(((ab)^* \odot a^*, abb), 1)$, and deciding it involves checking whether $abb \in \mathcal{L}((ab)^* \odot a^*)$ (and checking that the expression has precisely one shuffle operator, agreeing with the parameter). It is proven that this is $\mathrm{W}[1]$-hard using a reduction from k-clique. k-clique is the set $P \subseteq \mathbb{G} \times \mathbb{N}$ where $(G,k) \in P$ if and only if G has a clique of size k. k-clique is known to be $\mathrm{W}[1]$-complete.

3.5.2 The Membership Problems of Paper II

The graphical formalism sketched in this chapter is again slightly unspecific on some details, but is close enough to hyperedge replacement string grammars that we can restate all the results in Paper II in terms of it, although care should be taken and the paper read whenever vagueness makes any statement feel unclear.

Recall the definition of *rank* and *fan-out* from Section 3.4.2. Then Paper II considers the following four parameterized membership problems, all having the set of all instances $I \times \mathbb{N}$ where $I = \{(G,w) \mid G \text{ a grammar}, w \in \Sigma^*\}$. The grammars are of the LCFRS type in Paper II, but considering the sketched hyperedge replacement grammar case is illustrative enough. In each case the decision problem is to check whether $w \in \mathcal{L}(G)$, but the parameter k differs.

1. The problem where the fan-out is constant (i.e. fixed, not part of the problem) and the rank is the parameter. That is, deciding $P \subseteq I \times \mathbb{N}$ such that $((G,w),k) \in P$ if and only if $w \in L(G)$, G has rank at most k, and G has fan-out less than or equal to a constant c.

2. The problem where the rank is constant (less than or equal to a constant c) and the fan-out is the parameter.

3. The problem where the parameter contains both the rank and the fan-out. That is, the tuples in $P \subseteq I \times \mathbb{N}$ are still $((G,w),k)$ but G has rank at most k and fan-out at most k.[2]

4. The problem where the parameter contains the rank, the fan-out, and the length of the derivation. That is, we again have $((G,w),k) \in P$ if and only if $w \in \mathcal{L}(G)$, the derivation in w takes less than k steps, and both the rank and fan-out of G are less than k.

The first problem is proven to be W[1]-hard already for $c = 2$, again by a reduction from the k-clique problem. There is currently nothing to suggest that this problem is *in* W[1], and unfortunately W[1] is likely already rather hard.

The second problem is proven to be W[SAT]-hard already for rank 1, by reduction from a type of satisfiability problem that is W[SAT]-complete. This is (most likely) an even harder class than the previous parameterization, and mostly serves to illustrate the need for a better choice of parameter. This also makes the third problem W[SAT]-hard, since if fixing the rank to one gives W[SAT]-hardness it cannot help us to include it in the parameter, as it then only goes up by a constant in infinitely many hard cases.

Finally, consider the fourth problem, where the rank, the fan-out, and the derivation length are all included in the parameter. As we keep adding more and more of the problem to the parameter the complexity of the resulting parameterized problem should hopefully fall (up until the $(p,|p|)$ case discussed above), the limitation on the derivation length limits the length of the strings possible, but still does nothing to control the overall size of the grammar. However, the proof of W[1]-hardness for the first problem type is here reapplied, as the reduction incidentally also constructs a grammar where all derivations are short (in the overall size of the grammar). Luckily it can be proven that this problem is *in* W[1]. This is proven by using a special case of parsing short derivations in context-sensitive grammars, which is known to be W[1]-complete, and applying this to our short LCFRS derivations through a careful FPT reduction.

[2] It may seem more logical to make k the sum or product of the rank and fan-out, but since all treatment of the parameter is through arbitrary computable functions this is unnecessary, as for example squaring the maximum of the rank and the fan-out is necessarily greater or equal to the product of the two.

CHAPTER 4

Constraining Language Concatenation

In this chapter we consider another operator which in some ways operates in the opposite way of the binary shuffle operator. The binary shuffle operator for two languages L and L' constructs a language $L \odot L'$ which is a superset of the concatenation $L \cdot L'$. This superset is created by, in a sense, softening the requirement of the concatenation point, and letting strings interleave into each other. Here we will introduce the *cut* operator, which creates a *subset* of $L \cdot L'$, which contains all of the concatenations for which it is not in any way *ambiguous* where one string ends and the next starts. This is quickly clarified once we leap into the definition.

We will in addition compare and contrast this type of operator with a number of features and properties of real-world regular expression engines. This chapter, since it deals with a somewhat singular practical regular expression feature, does not have a *-marked section, and instead has a slightly more technical slant in various parts. Familiarity with regular expressions is important for understanding the material here presented.

4.1 The Binary Cut Operator

The cut operator is a kind of concatenation of languages. To state this definition we need some additional notation. For any string $\alpha_1 \cdots \alpha_n \in \Sigma^*$ (i.e., $\alpha_i \in \Sigma$ for each i) let $\mathit{prefix}(\alpha_1 \cdots \alpha_n) = \{\alpha_1, \alpha_1\alpha_2, \ldots, \alpha_1 \cdots \alpha_n\}$, that is, all *non-empty* prefixes of $\alpha_1 \cdots \alpha_n$. Let $\mathcal{P}(S)$ denote the power-set of a set S. Then the definition of the cut is as follows.

Definition 4.1 (Binary Cut Operator) Let $! \colon \mathcal{P}(\Sigma^*) \times \mathcal{P}(\Sigma^*) \to \mathcal{P}(\Sigma^*)$ be a binary operator such that

$$\mathcal{L} \,!\, \mathcal{L}' = \{uv \mid u \in \mathcal{L}, v \in \mathcal{L}', uv' \notin \mathcal{L} \text{ for all } v' \in \mathit{prefix}(v)\}$$

for any languages $\mathcal{L}, \mathcal{L}' \in \Sigma^*$. ◇

Notice that this definition ensures that $\mathcal{L} \,!\, \mathcal{L}' \subseteq \mathcal{L} \cdot \mathcal{L}'$, that is, the cut is a subset of the concatenation. The inclusion is not necessarily strict, for example if $\mathcal{L}' = \{\varepsilon\}$ it necessarily follows that $\mathcal{L} \,!\, \mathcal{L}' = \mathcal{L} \cdot \mathcal{L}'$.

Let us look at a series of examples (partially borrowed from Paper III, see Section 2 of that paper for further examples).

Example 4.2 (Empty Cuts) Let $\mathcal{L} = \{ab, abb, abbb, \ldots\}$ and $\mathcal{L}' = \{bc, bbc, bbbc, \ldots\}$ (that is, $\mathcal{L} = abb^*$ and $\mathcal{L}' = bb^*c$). Then $\mathcal{L} \,!\, \mathcal{L}'$ is the empty language. Let us consider one of the strings in the concatenation, *abbc*. This string cannot be in $\mathcal{L} \,!\, \mathcal{L}'$, as, looking at Definition 4.1, splitting it into $u = ab$ and $v = bc$, while fulfilling $u \in \mathcal{L}$ and $v \in \mathcal{L}'$, is not permitted as $b \in \mathit{prefix}(bc)$, and $abb \in \mathcal{L}$. Picking $u = abb$ leaves $v = c$, which is not in $\mathcal{L}'$. ◇

A more interesting language generated by a cut is the following.

Example 4.3 (Very Unsymmetrical Cuts) Let $\mathcal{L} = \{a, b, aa, bb, aaa, bbb, \ldots\}$ and let $\mathcal{L}' = \{ac, bc\}$. Then $\mathcal{L} \,!\, \mathcal{L}' = \{abc, bac, aabc, bbac, aaabc, bbbac, \ldots\}$. The reason is simple; if the u (in the sense of Definition 4.1) is chosen to be some number of "a"s, then v cannot be picked to be *ab*, since the *a* prefix will be consumed by $\mathcal{L}$. Similarly, if the string starts with a *b* it becomes impossible to pick v as *bc*.

This illustrates that while the cut $\mathcal{L} \,!\, \mathcal{L}'$ produces a subset of $\mathcal{L} \cdot \mathcal{L}'$ it does *not necessarily* produce a language of the form $L \cdot L'$ for some $L \subseteq \mathcal{L}$ and $L' \subseteq \mathcal{L}'$. ◇

4.2 Reasoning About the Cut

As a short aside, let us consider how the cuts relate to the other formalisms presented here. Where the shuffle operator effectively takes two strings, let us call them w and v, and interleaves them in such a way that, reading the result left to right, we (in general) have no idea which string each symbol belongs to. Perhaps v has not even started yet, or perhaps all of it has already been seen. The cut, on the other hand, is such that it *only permits* the concatenations where there is *no* ambiguity about where w ends. The only way wv is in the language generated by the cut is if the point of concatenation is decided entirely by the language w belongs to. That is, reading a string from a cut language from left to right we know that we have finished reading the w part when it is no longer possible for w to be longer. Then the remaining string must be the v part.

It is in this way the cut and the shuffle can be viewed as opposite directions from the concatenation, where the shuffle permits more ways of combining w and v, and the cut permits only a subset of all possible concatenations based on removing ambiguity when reading from the left.

4.3 Real-World Cut-Like Behavior

In the case of the cuts the real-world motivation is rather immediate and, hopefully, compelling. Regular expressions are a very popular tool for programmers, and regular languages of other forms also show up with great frequency. However, in mixing non-deterministic constructions for testing language membership and the deterministic flow control of the "main" program some very interesting effects can be achieved (or, alternatively, unexpected problems may be created, as the case may be).

Consider the Python function shown below, which successively matches multiple regular expressions to the same string.

Listing 4.4 (A Repeated Regular Expression Python Program)

```
# match argument against successive regular expressions
def matchx(s):
    # match a prefix of s against aa*|bb*
    m1 = re.match("^aa*|bb*", s)
    if m1 != None:
        # if ok, match the remainder of s against ab|bc
        m2 = re.match("ab|bc$", s[m1.end(0):])
        if m2 != None:
            # if both succeded report success
            return "Matched"
    # otherwise failure
    return "Did not match"
```

Basically, the code in Listing 4.4 is a function, which takes a string s as input, and then matches a *prefix* of s to the regular expression $aa^*|bb^*$ (the language $\{a,b,aa,bb,aaa,bbb,\ldots\}$), if that match is successful the *remaining suffix*, that is, whatever remains after removing the prefix that the first regular expression matched, of s is matched against the regular expression $ab|bc$ (the language $\{ab,bc\}$). If that match is successful success is reported.

The language "matched" by this program is exactly the language $(aa^*|bb^*)\,!\,(ab|bc)$. It might be easy to think that is *should* match $(aa^*|bb^*)\cdot(ab|bc)$ (which includes e.g. *aab* and *bbc*, which are not included in Example 4.3), but this is not the case. The thinking is exactly the one discussed for the cuts, the deterministic behavior of the outer program lets the first regular expression read as much as it wants, and the default behavior or regular expressions in most software libraries is to make the longest possible match. Once that has happened the suffix has been deterministically selected, and the second regular expression *must* match whatever is left for the overall match to work. In comparison, $(aa^*|bb^*)\cdot(ab|bc)$ features the non-deterministic behavior "intended" in regular expressions, the default behavior in most software packages will still be that the first part should match as much as possible, but if the overall match fails it will backtrack and choose a smaller match (if possible) in deference to the entire expression succeeding.

4.4 Regular Expressions With Cut Operators Remain Regular

4.4.1 Constructing Regular Grammars for Cut Expressions

Next we in short recap a result given in full in Paper III, showing that adding the cut operator to regular languages (or, of course, regular expressions) creates regular languages. We will, with some further extensions later, call these expressions which add

the cut to the normal set of regular expression operators *cut expressions*. The straightforward way to demonstrate that cuts preserve regularity is by employing a product construction, a variation of which was already employed in Section 2.4 to demonstrate the regularity of regular expressions extended with the binary shuffle operator.

Assume that we have some regular grammars (in the vein of Figure 1.3) R_1 and R_2, and that we wish to construct a *regular* grammar R for $R_1 \,!\, R_2$. Basically we will for each non-terminal A_1 in R_1 and each non-terminal A_2 in R_2 construct the non-terminals $(A_1, \perp)$, $(\perp, A_2)$ and (A_1, A_2) in the new grammar. The extra symbol $\perp$ is intended to signify "absent" here, and the new grammar will start out in $(S, \perp)$ where S is the initial non-terminal from R_1. The full construction then carefully ensures that whenever we have read a prefix of the input-string that R_1 *could* accept it starts R_2 on *its* initial non-terminal (i.e., if we are in (A, B) and R_1 can get rid of A without reading any more we go to (A, S'), where S' is the initial non-terminal for R_2). That is, as the string is read whenever R_1 can accept the string it restarts R_2 in its initial state, whenever R_2 can accept the grammar for $R_1 \,!\, R_2$ can accept.

The basic intuition behind this construction is that for every prefix that R_1 can accept Definition 4.1 says two things:

- It is *possible* that R_2 may start matching at this point, if R_1 cannot go on to match something longer.
- It is not allowed that we have *already* switched to matching in R_2.

In effect the construction speculatively keeps track of both R_1 and R_2, ensuring that R_1 gets its longest possible prefix of the string read, while keeping track of what R_2 has otherwise done.

The construction is hard to further simplify in a form that is more instructive than the full version, so refer to Lemma 2 of Paper III for a deeper explanation.

4.4.2 Potential Exponential Blow-Up in the Construction

While the cut expressions generate only regular languages, proving no more powerful than regular expressions, this does not mean that it is a pointless formalisms. There are two additional considerations to make.

1. Does the formalism allow something important to be conveniently expressed?
2. Does it express some languages in a more compact way?

In the case of cut expressions both are true. Modeling the loss of non-determinism illustrated in Listing 4.4 (and later in Listing 4.7) is interesting, and as we will demonstrate next there are also some families of languages where the smallest regular expression is exponentially larger than the equivalent cut expression, even when restricting ourselves to use only a single binary cut operator.

The core of this argument is simply that the cut can express a set difference of sorts on languages.

Lemma 4.5 Let $\Gamma = \Sigma \cup \{\#\}$, we assume that $\# \notin \Sigma$. Then $\left((L\#\Gamma^*) \,|\, \varepsilon\right) \,!\, (\Sigma^*\#) = (\Sigma^* \setminus L)\#$ for all languages L over Σ. ◇

A complete proof of this lemma is out of the scope of this introduction, but it is fairly intuitive when one considers the cases. Assume that $w \in L$, then, for the lemma to hold, $w\#$ should *not* be in $((L\#\Gamma^*)\,|\,\varepsilon)\,!\,\Sigma^*\#$. This is clearly the case, as the $L\#\Gamma^*$ will consume it entirely, leaving nothing for the trailing $\Sigma^*\#$ to match. This in fact holds for any string v with $w\#$ as a prefix, as the Γ^* keeps consuming all symbols. In the other direction, assume that $w \notin L$. Then $w\#$ is not matched by $L\#\Gamma^*$, meaning that the ε part of the branch is chosen, and then $\Sigma^*\#$ matches it and the match succeeds.

To exploit Lemma 4.5 we can now construct a *regular* expression R over Σ such that the shortest string R does *not* match is exponential in length (compared to the length of R). We can then apply Lemma 4.5, taking L as $\mathcal{L}(R)$, to produce a cut expression for which the shortest matching string is exponential in the length of the expression. From this we can then draw the conclusion that the smallest regular grammar (or finite automaton or regular expression) is at least exponentially larger than the cut expression. Recall Definition 1.6 in preparation, and note that R^k (for some regular expression R and $k \in \mathbb{N}$) is not a regular expression operator, but is here used as an abbreviation for

$$\underbrace{R \cdot R \cdots R}_{k \text{ times}}.$$

If regular expressions are extended with an actual R^k operator the explosion in size would be even greater.

We select $\Sigma = \{0, 1, \$\}$ as the alphabet, and let $\Delta = \{0, 1\}$. For each $n \in \mathbb{N}$ the regular expression R_n has five components, $R_n = A\,|\,B\,|\,C_n\,|\,D_n\,|\,E_n$, which are as follows. In the end the language considered will be $(\Sigma^* \setminus \mathcal{L}(R_n))\#$, so the aspect to consider is e.g. the language $\Sigma^* \setminus A$ and so on.

1. $A = \Delta\Sigma^*\,|\,\$\Delta^*1\Delta^*\$\Sigma^*$. Note that all strings in $\Sigma^* \setminus A$ start with \$ and contain no 1 until the next \$.

2. $B = \Sigma^*\Delta\,|\,\Sigma^*\$\Delta^*0\Delta^*\$$. Note that all strings in $\Sigma^* \setminus B$ end with \$ and contain no zero between the last two \$ symbols.

3. $C_n = \Sigma^*\$\Delta^{n+1}\Delta^*\$\Sigma^*\,|\,C_{n,0}\cdots C_{n,n-1}$ where $C_{n,i} = \Sigma^*\$\Delta^i\$\Sigma^*$ for each i. Note that all strings in $\Sigma^* \setminus C_n$ have exactly n zeroes and ones between each pair of \$ symbols.

4. $D_n = D_{n,1}\,|\cdots|\,D_{n,n-2}$, where $D_{n,i} = \Sigma^*\$\Delta^i0\Delta^*0\Delta^*\$\Delta^i1\Delta^*\$\Sigma^*$ for each i. Note that the strings in $\Sigma^* \setminus D_n$ are such that every substring $\$x\$y\$$ (with $x, y \in \Delta^n$, which will be enforced by C_n), is such that if the ith symbol in x is a zero, and the ith symbol in y is a 1, then symbols $i+1$ through n in x must be ones.

5. $E_n = E_{n,1}\,|\cdots E_{n,n-2}$ where $E_{n,i} = \Sigma^*\$\Delta^i01^{n-i-1}\$\Delta^i1\Delta^*1\Delta^*\$\Sigma^*$ for each i. Note that all strings in $\Sigma^* \setminus E_n$ are such that every substring $\$x\$y\$$ (with $x, y \in \Delta^n$, which will be enforced by C_n), is such that if the ith symbol in x is a 0, symbols $i+1$ through n are ones, and the ith symbol in y is a 1, then the remainder of y must be zeroes.

Taking all these together we learn that each $w \in (\Sigma^* \setminus \mathcal{L}(R_n))$ are strings such that $w = \$x_1\$x_2\$\cdots\$x_m\$$ where each x_i is a string of n zeroes and ones (due to C_n), such that $x_1 = 0\cdots0$ (due to A), and $x_n = 1\cdots1$ (due to B). At most one zero in x_i can be turned into a one in x_{i+1}, and only if all the subsequent positions were ones in x_i and are zeroes in x_{i+1}. From this it directly follows that the shortest string in $\Sigma^* \setminus \mathcal{L}(R_n)$ will be the sequence of all n-bit binary strings in order, for example

$$\$000\$001\$010\$011\$100\$101\$110\$111\$ \in R_3.$$

In addition a number of *longer* strings exist, which show up since a one in x_i may turn into a zero in x_{i+1}. However, the only way to get from the initial zero sequence to the final one sequence is to increment by one in the binary addition sense at least 2^n times.

Notice, however, that the actual expression R_n is on the order of n^2 symbols long, where C_n, D_n and E_n are the big part. Applying Lemma 4.5 constructs a cut expression accepting $(\Sigma^* \setminus \mathcal{L}(R_n))\#$, which is still on the order of n^2 symbols long, but as argued above the shortest string it accepts is exponential in n.

Non-extended regular expressions, regular grammars (as sketched in figures here) and finite automata are all such that the shortest string they accept is at most linear in the size of the expression/grammar/automaton (if they accept any string at all). This is easy to see, some efficient shortest path algorithm can be employed to find a path through the expression/grammar/automaton. As such, cut expressions are exponentially more succinct in some cases, and converting an arbitrary cut expression into one of those listed formalisms may create an exponentially larger representation. This will be rather key to understanding the complexity of solving the membership problem.

4.5 The Iterated Cut

In a further parallel with Chapter 2 we also consider a unary iterated version of the cut. Much like $R^{\odot} = R \odot R \odot \cdots \odot R$ we let $R^{!*} = R \mathbin{!} (R \mathbin{!} (R \mathbin{!} \cdots (R \mathbin{!} R)\cdots))$. However, notice that while the shuffle operator is associative, the cut operator is not.

Example 4.6 Let us consider two expressions differing only in associativity $((ab)^* \mathbin{!} a) \mathbin{!} b$ and $(ab)^* \mathbin{!} (a \mathbin{!} b)$.

- For $((ab)^* \mathbin{!} a) \mathbin{!} b$ clearly $(ab)^* \mathbin{!} a$ is the same as $(ab)^*a$, since the $(ab)^*$ part cannot cover the final a, so $\mathcal{L}((ab)^* \mathbin{!} a) = \{a, aba, ababa, \ldots\}$, and, hence, $\mathcal{L}(((ab)^* \mathbin{!} a) \mathbin{!} b) = \{ab, abab, ababab, \ldots\}$.
- However, if we instead consider $(ab)^* \mathbin{!} (a \mathbin{!} b)$, we notice that $a \mathbin{!} b$ is the same as ab, so we have $(ab)^* \mathbin{!} ab$ which is clearly empty, as the $(ab)^*$ will consume all repetitions of ab ensuring the second part never gets to match anything.

It follows that the cut operator is *not* associative. ◇

The iterated cut, in a sense similar to how the binary cut models the program in Listing 4.4, permits the modeling of loops of regular expression matching, like in the below listing.

Listing 4.7 (A Looping Regular Expression Python Program)

```
# match s against the regular expression R repeatedly
def matchy(R, s):
    # keep matching
    while True:
        # match s to re once
        m = re.match(R, s)
        # if the match failed report failure
        if m == None:
            return "Did not match"
        # otherwise, extract the remainder of the match
        s = s[m.end(0):]
        # if the whole string matched, report success
        if len(s) == 0:
            return "Matched"
```

This listing will give the same behavior as trying to match s to $R^{!*}$.

The iterated cut is hard to express directly in a regular expression with just the addition of the binary cut operator. Notably it is not a matter of nesting cuts inside of Kleene closures, like $(R\,!\,R)^*$ or similar, as this will give too much non-deterministic freedom in general. However, adding both the binary cut operator *and* the iterated cut to regular expression *still* produces expressions that can only generate regular languages. The construction for this part is slightly trickier than for the case of the binary cut operators, so it is best to refer to Paper III where complete and formal constructions for both cases are given.

4.6 Regular Expression Extensions, Impact and Reality

4.6.1 Lifting Operators to the Sets

Recall Definition 1.6 where the basic operations in regular expressions are defined. It is an important fact to note that each of those classical regular expression operators are expressed string-wise. That is, an operator f takes n argument subexpressions $R_1, \ldots, R_n$, and the language it generates is then

$$\mathcal{L}(f(R_1, \ldots, R_n)) = \{f(v_1, \ldots, v_n) \mid v_1 \in \mathcal{L}(R_1), \ldots, v_n \in \mathcal{L}(R_n)\}.$$

That is, the classic operators all operate "point-wise" on strings, and this is then lifted to the level of sets (i.e. we can take the categorial view and consider a functor here) to generate languages. However, the cut does not operate on this level. Instead Definition 4.1 operates on the level of the language. We can talk about $L\,!\,L'$ for languages, but informed that $w \in L$ and $v \in L'$ we cannot from this determine whether $wv \in L\,!\,L'$.

This *should* be viewed as a flaw with the cuts, their introduction into expressions does change the nature of the expression in a fundamental way. On the other hand, the impact is comparatively small when contrasted to the cut-like operators that many

regular expression software packages include. These have behavior that is even further from the clean nature of the classical operators.

4.6.2 An Aside: Regular Expression Matching In Common Software

This way of phrasing how matching happens may appear unusual for anyone more familiar with the more classic regular expression constructions, where the semantics are described in a composed way, i.e., $\mathcal{L}(R_1|R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, etc., or by constructing finite automata for the expression (constructing the Glushkov automaton [Glu61] and determinising it, or directly constructing a deterministic finite automaton using e.g. derivatives [Brz64]). Most practical software packages, however, use a depth-first backtracking search across some abstract syntax tree representation of the regular expression. The reasons for this are two-fold.

1. The efficiency of this approach is in many cases great. Constructing the syntax tree is efficient, and the representation is in general far more compact than the automata approach. The actual search may in the worst case be a lot slower (exponential in the length of the string), but the semantics are straightforward enough that the task of structuring the expression in a way that gives efficient matching in the most common cases can be left to the programmer.

2. It enables a multitude of additional regular expression features. Most immediately it makes it possible to deterministically talk about which part of the regular expression matches which part of the string. That is, $(a^*|b^*)(a|b)^*$ matches *aaababa*, but which part of the expression matches the *aaa* prefix? In the theoretical setting this is a nonsense question, all we state is that $aaababa \in \mathcal{L}$, the *how* is entirely undefined. In regular expression software packages however the initial three *a*s will be matched by the first a^*, and this information can be extracted with the API provided. Which parts of the expression will "prefer" to match what can be controlled further with a variety of operators, and the pieces of the string matched by a certain subexpression can even be recalled inside the expression (permitting the language $\{ww \mid w \in \Sigma^*\}$ to be matched by recalling a copy of the string already matched).

In short; the accepted approach has numerous implications for the functionality and performance of regular expression matching in practice.

4.6.3 Real-World Cut-Like Operators

There are a variety of operators in practical regular expression packages which behave *somewhat* similar to cuts. The first, and most common, are the possessive quantifiers. Let us look specifically at the possessive variation of the Kleene star R^* as defined in Definition 1.6, denoted R^{*+}. Defining the language generated by R^{*+} leads to disappointment however, $\mathcal{L}(R^{*+}) = \{\varepsilon\} \cup \{vw \mid v \in \mathcal{L}(R), w \in \mathcal{L}(R^{*+})\}$ inductively. Unfortunately this is precisely the same language as generated by $\mathcal{L}(R^*)$, which is because the possessive quantifier does not operate on the same level as classical regular operators, or even the set-level behavior of the cut operators. Instead the semantics of the possessive quantifiers are intertwined with the overall matching of the entire expression in a

way that is hard to formalize. Consider the examples in Table 4.8 which are produced using the Java (1.6.0u18) regular expression implementation. Notice how applying

Table 4.8: Some regular expressions using possessive quantifiers and the language they accept in Java 1.6.0u18.

Expression	Language
$(aa)^{*+}a$	$\{a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^{*+}a)^*$	$\{\varepsilon, a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^{*+}a)^*a$	$\{a\}$

the Kleene star to the expression in the first row does not (in the second row) generate for example *aa*, despite *a* being in the language of the first row.

We will not attempt to deeply explain the semantics of this operator, but it operates by manipulating the internal backtracking search. The outcome does not easily fit into the compositional classic explanation of how regular expressions generate languages. See Paper III for more examples of this type of operator.

As an addition, some regular expression engines feature an additional binary operator, `(*PRUNE)`, that compares fairly directly to the binary cut operator (in that it is not attached to a Kleene star), but still has semantics that are hard to comprehend form the compositional perspective. See Table 4.9 for some examples of expressions and the languages recognized in Perl 5.16.2.

Table 4.9: Some regular expressions using the `(*PRUNE)` operator and the language they accept in Perl 5.16.2, similar to the examples in Table 4.8.

Expression	Language
$(aa)^*$ `(*PRUNE)` a	$\{a, aaa, aaaaa, aaaaaaa, \ldots\}$
$((aa)^*$ `(*PRUNE)` $a)^*$	$\varnothing$
$((aa)^*$ `(*PRUNE)` $a)^*a$	$\varnothing$

4.6.4 Exploring Real-World Regular Expression Matchers

Paper IV explores the behavior of these practical software package matchers. They effectively operate by constructing an automaton (or grammar) where rules are prioritized, whenever there are multiple rules that *could* be applied there is a preferred rule that is tried first. If applying that rule does not lead to accepting the string the procedure backtracks and tries the other options. The full discussion requires a deep technical look at the behavior of the software, and is best explored by reading Paper IV. Suffice it to say, beyond exploring the semantics to analyze additional operators, this search procedure will additionally at times require exponential time. Consider for ex-

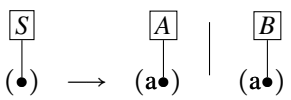
ample the expression $(a|a)^*$, trying to match the string $a\cdots ab$. It will fail to match the b, but in the process the matching procedure will pick whether the first or second a should match each a in the string, and when the failure on b happens the backtracking will attempt *every other* way of matching the as to the string. In fact, attempting this match using Java on the authors (reasonably modern) machine the runtimes shown in Table 4.10 are achieved. The main contribution in Paper IV is in statically analyzing

Table 4.10: The time in seconds it takes to match the string $a\cdots ab$ to the regular expression $(a|a)^*$ in Java on the authors desktop PC, as it depends on the number of as in the string. Notice the almost perfect power of two exponential growth.

Number of "a"s in $w = a\cdots ab$	23	24	25	26	27	$\cdots$	30
Seconds to match w **to** $(a|a)^*$	1.04	2.00	3.66	7.22	13.56	$\cdots$	118.81

regular expressions for this type of exponential worst-case behavior (i.e., a^* can never take exponential time, since there is only one choice, but $(a^*)^*$ can).

One additional point of interest in Paper IV is how the matcher picks which choice to explore first. This is done by giving finite automata priorities, where one choice is more prioritized than another. This leads to the definition of the prioritized non-deterministic finite automata (pNFA) formalism in Paper IV. These are fairly straightforward, if we imagine the rule



we now say that the first possibility, which generates the non-terminal A, is *prioritized*. That is, if A can generate the rest of the string we prefer to have it do so, and try generating the rest with B only if A fails. This distinction makes no difference for the language accepted, but it makes it unambiguous how the string is generated, which ensures that the solution to a parsing problem instance is unequivocal.

4.7 The Membership Problem for Cut Expressions

Parts of the membership problem for cut expressions should already be clear; namely, Section 4.4 and Section 4.5 together demonstrate that the cut expressions generate only regular languages. The *non-uniform* membership problem for regular languages is decidable in linear time, so we can decide the non-uniform membership problem for cut expressions in linear time, since we can just rewrite the cut expression into a regular grammar or similar (through the arguments in the aforementioned sections).

However, as Section 4.4.2 demonstrates, the regular grammar may be exponentially large, so the equivalence to regular grammars gives us no more than an exponential algorithm for deciding the uniform membership problem. Luckily a very

direct table parsing algorithm can decide membership in cubic time. Let us sketch very quickly how it is done.

Algorithm 4.11 (Parsing for Cut Expressions) Take as input a cut expression E and a string $\alpha_1 \cdots \alpha_n$. Let S_E denote the set of subexpressions of E (including E itself).

1: Construct the table $T : S_E \times \{1,\ldots,n+1\} \times \{1,\ldots,n+1\} \to \{true, false\}$.
2: Set $T(E,i,j) := false$ for all E,i,j at the start
3: **for** $S \in S_E$, working bottom-up through the sub-expressions **do**
4: **if** $S = \varepsilon$ **then** $T(S,i,i) := true$
5: **else if** $S \in \Sigma$ **then** $T(S,i,i+1) := true$ for all i with $\alpha_i = S$
6: **else if** $S = E_1 \,|\, E_2$ **then**
7: $T(S,i,j) := true$ for all $i \leq j$ s.t. $T(E_1,i,j) \vee T(E_2,i,j)$
8: **else if** $S = E_1 \cdot E_2$ **then**
9: $T(S,i,k) := true$ for all $i \leq j \leq k$ s.t. $T(E_1,i,j) \wedge T(E_2,j,k)$
10: **else if** $S = E_1^*$ **then**
11: $T(S,i_1,i_n) := true$ for all n, $i_1 \leq \cdots \leq i_n$ s.t. $T(E_1,i_1,i_2) \wedge \cdots \wedge T(E_1,i_{n-1},i_n)$
12: **else if** $S = E_1 \,!\, E_2$ **then**
13: $T(S,i,k)$ for all $i \leq j \leq k$ such that:
14: $T(E_1,i,j) \wedge T(E_2,j,k)$, and,
15: $\neg T(E_1,i,j')$ for all $j < j' \leq k$.
16: **end if**
17: **end for**

This algorithm is trivially cubic (quadratic in the length of the string), since every table position is set true at most once. The case for the shuffle closure is not included, but is a trivial addition.

After the threat of potentially exponentially large regular grammars the cubic time (and space) of the above algorithm may be calming, but given the typical efficiency of matching classical regular expressions cubic time is still not entirely pleasing. Better algorithms remain an open question however, very notably Section 4.4.2 demonstrates a case where applying a cut exponentially blows up the size of the smallest corresponding regular grammar exponentially, but for the upper bound we only know that it cannot be *worse* than non-elementary, which is not very satisfying. This in fact follows from the product-style construction discussed in Section 4.4.1, and is discussed at greater length in Paper III.

CHAPTER 5
Block Movement Reordering

This short chapter discusses matters of block reordering, which is once again a non-obvious term, this time lifted form the field of edit distance, where operations that modify multiple symbols in a contiguous substring at once are referred to as block operations. Specifically the topic of interest is attempting to study the results of reordering nodes in the parse tree for a string, which gives rise to a sort of hierarchical block movement reordering in the underlying string language.

5.1 String Edit Distance

String edit distance is a long studied field. It is concerned with defining a distance between strings using a sequence of operations (reminiscent of the rule-based derivations discussed in earlier chapters, but starting from another, possibly *longer*, string). The distance measure is defined in terms of a set of operations, each of which makes some small modification to a string, and then the distance between a string $w \in \Sigma^*$ and $v \in \Sigma^*$ is the minimum number of operations (possibly weighted in some way) we need to apply to modify w into v. The problem of finding this sequence is known as the *string correction problem*. A classic set of operations for this is to have an operation to *delete* a single symbol and one to *insert* a single symbol. Making the distance from e.g. *abc* to *cca* four, since the initial *ab* must be removed (two removal operations) and *ca* must be added at the end. A typical additions to the set of operations is to add an operator to *replace* one symbol by another, this set of three operators is called Levenshtein distance [Lev66]. The next typical addition, and most important for us here, is the *swap*, which swaps the positions of two adjacent symbols, the resulting set of operators is called Damerau-Levenshtein distance [Dam64].

5.2 A Look at Error-Dilating a Language

The direction of interest here starts out from the question of an error *dilation* of a language. Consider Figure 5.1. That is, we choose a language class $\mathcal{G}$ (perhaps the regular or context-free languages) and a string edit distance e, then for each language $L \in \mathcal{G}$ and each $k \in \mathbb{N}$ we define $L_{e=k}$ to mean that $w \in L_{e=k}$ if and only if there exists some $v \in L$ such that w is k or less distance from v. Notably, as k approaches ∞ the

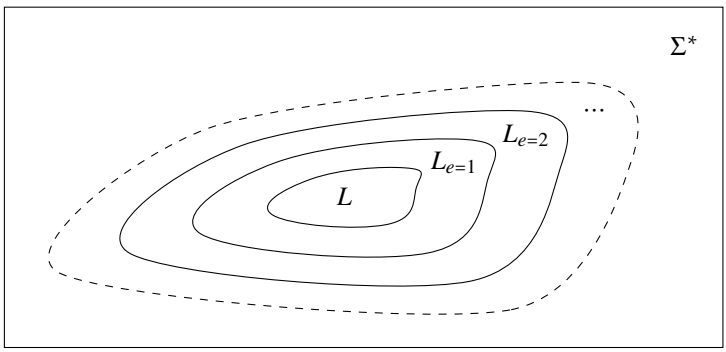


Figure 5.1: A diagram of the dilation of a language through error measures.

language $L_{e=k}$ approaches Σ^*.

Performing such a construction is fairly straightforward for most choices of formalisms. If we consider just the case of *insert* and *delete* with a regular grammar G, and the constant k chosen, then we can for each non-terminal A construct $k+1$ new non-terminals $A_k, A_{k-1}, \ldots, A_0$. The non-terminal A_i has all the rules that A would have, with i preserved, so for example the left rule turns into the right in the following way.

$$\boxed{A} \to (\bullet) \longrightarrow \boxed{B} \to (\alpha\bullet) \qquad \boxed{A_i} \to (\bullet) \longrightarrow \boxed{B_i} \to (\alpha\bullet)$$

In addition for each $\alpha \in \Sigma$ and non-terminal A_i with $i > 0$ we add the following rule.

$$\boxed{A_i} \to (\bullet) \longrightarrow \boxed{A_{i-1}} \to (\alpha\bullet)$$

This allows one "insertion" to be used, we count down the number allowed and add an arbitrary symbol.

Finally, for each existing rule that *would* add a symbol we simulate a deletion by adding a rule that counts down i but "fails" to generate the symbol, as above with the left original rule and the right new rule (though one for each $0 < i \leq k$ must be generated of course).

$$\boxed{A} \to (\bullet) \longrightarrow \boxed{B} \to (\alpha\bullet) \qquad \boxed{A_i} \to (\bullet) \longrightarrow \boxed{B_{i-1}} \to (\bullet)$$

Finally, we let the starting non-terminal S go to S_k (to signify that we start out with k operations available). Notice how, once the subscript gets to zero, only the "original" rules are usable, each use of a insertion/deletion rule "costs" one from the subscript.

5.3 Adding Reordering

5.3.1 Reordering Through Symbol Swaps

Adding the simple symbol swap to the prior construction is only minimally more complex. We can extend the tagging of the non-terminals to remember "we pretended to swap α for a β", meaning that we generated a β from a rule that should have generated α, and this tagging of the non-terminal lets it only take rules which have been modified such that they *originally* generated β, but in this modified rule they generate the missing α and the derivation continues on as normal.

5.3.2 Derivation-Level Reordering

We now get to the real aim of this section, the intent of this edit distance is to model some sort of error or imprecision, however, in the context of a lot of languages simply replacing symbols may not really reflect the nature of errors properly. Consider for example in natural languages, where large grammatical restructurings may be only "slightly" bad, since they still obey some basic rules. That is, "She chases a blue ball" is correct English, whereas "A blue ball she chases" is slightly ungrammatical, but still completely understandable, whereas "ball she chases a blue" is incomprehensible despite involving less reordering.

We have so far dodged the issue of parse trees, but here they become rather core to the question. Consider Figure 5.2. This illustrates (a possible interpretation of)

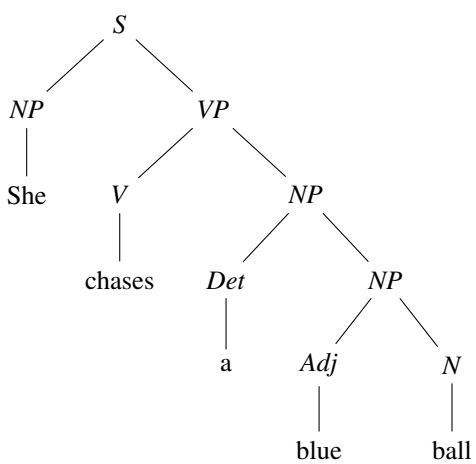


Figure 5.2: A parse tree for the sentence "She chases a blue ball", the internal nodes of the tree corresponds to the non-terminals which generated that part of the sentence, the words in the leafs are the symbols of Σ in this case.

the structure of the natural language sentence. It stands to reason that small modifications of this tree will have a closer relationship to the original sentence than small modifications to the string which forms the sentence.

5.3.3 Tree Edit Distance

Tree edit distance is a natural way to think about this, that is, we would like to create an error dilation of a language, in the style of Figure 5.1 using tree operations on the parse tree (which requires a specific instance of a grammar for the language to make sense) to modify the final strings.

The problem of tree edit distance is fairly well explored in some limited settings, see e.g. [Sel77, Tai79]. This work has for the most part however been constrained to just allowing insert and delete operators, the swap, or similar subtree movement operators, is a trickier matter [ZS89, Kle98, Bil05]. This is partially necessary, the tree edit distance on unordered trees (i.e., we allow deletions and insertions of nodes, but siblings in the tree have no order) is NP-complete [ZSS92]. We can simulate the unordered case if swaps are permitted, by simply replacing each internal node by a long chain of copies of the node. This way the swap remains cheap (it does not care how many nodes it moves in swapping two siblings) while making insertions and deletions expensive. If we add sufficiently many of these nodes the result will be that all orders can be achieved cheaper than it is to perform a single insertion or deletion, effectively making the problem behave like the unordered case.

5.4 Analyzing the Reordering Error Measure☆

Paper VI considers the very limited case of *only* permitting tree swaps in the distance measure, each swap having a cost of one. Let us consider the proper definition, recalling the definitions of trees from Section 2.9.1. First we define the swap distance between permutations.

Definition 5.3 Let $\pi_n \subset \mathbb{N}^n$ denote the set of permutations of length n, that is, $p_1 \cdots p_n \in \pi_n$ if and only if $\{p_1, \ldots, p_n\} = \{1, \ldots, n\}$. Then $p_1 \cdots p_n \in \pi_n$ has a swap distance less than or equal to k, denoted $swap(p_1 \cdots p_n, k)$ if and only if

- $k \geq 0$ and $p_1 \cdots p_n = 1 \cdots n$,
- there exists some i such that $swap(p_1 \cdots p_{i-1} p_{i+1} p_i p_{i+2} \cdots p_n, k-1)$. ◇

Then the tree variant is as follows.

Definition 5.4 For two trees t, t' we say that t and t' are within tree swap distance k, denoted $swap(t, t', k)$, if and only if $t = \alpha[t_1, \ldots, t_n]$ and $t' = \alpha[t'_1, \ldots, t'_n]$ for some $\alpha \in \Sigma$ and n, and there exists some $p_1 \cdots p_n \in \pi_n$ and $l_0, \ldots, l_n \in \mathbb{N}$ such that

- $k \geq \sum_{i=0}^{n} l_i$,
- $swap(p_1 \cdots p_n, l_0)$, and
- $swap(t_i, t_{p_i}, l_i)$ for all i.

The triple (t, t', k) is a "yes" instance of the tree swap distance problem if and only if $swap(t, t', k)$. ◇

Unfortunately this problem is proven to be NP-hard in Paper VI (the problem is obviously *in* NP, since the permutations for each level of the tree can be guessed and then verified in polynomial time). Let us briefly outline the process. The reduction starts with the *extended string-to-string correction problem*, which is an edit distance with *only* delete and swap operations[1]. This problem is known to be NP-complete (see e.g. [GJ90]). The reduction makes an intermediary stop in a problem that may be interesting in itself.

Definition 5.5 Let $M : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be an n by n matrix (i.e. a matrix with n rows and columns, letting $M(i, j)$ denote the value at the ith row and jth column), then (M, k) is a "yes" instance of the swap assignment problem if and only if there exists a permutation $p_1 \cdots p_n \in \pi_n$ and $l \in \mathbb{N}$ such that

- $swap(p_1 \cdots p_n, l)$
- $k \geq l + \sum_{i=1}^{n} M(p_i, i)$ ◇

That is, the problem is to decide whether it is possible to swap rows positions in M in such a way that the sum of the number of swaps used and the diagonal in the resulting matrix is less than or equal to k. The reduction is such that e.g. position $M(i, j)$ is zero if symbol i of the first string in the original edit distance problem is the same as symbol j in the second, combined with some trickery to enable deletions.

This matrix problem is then in turn reduced to the tree swap problem of Definition 5.4. This reduction is not overly difficult, the tree constructed will have height 3, the root has n immediate children corresponding to the rows, these have n children corresponding to the column positions in that row, and finally these have a coding of the number they should contain as children. Everything is distinctly coded so the swaps can only be used to reorder the rows, and to make the binary representations equal (which costs exactly the absolute difference between the numbers).

The fact that the tree swap distance is NP-complete is unfortunate, however, the amount of distance permitted in the error-dilating of languages should be very constrained (e.g. a sentence with three or more errors will often be incomprehensible already), so fixed parameter analysis and other more nuanced analysis would be of great interest.

[1] This statement abuses the notion of a distance heavily, since it is asymmetric. It does however fall into a similar class of problems.

CHAPTER 6
Summary and Loose Ends

None of the matters here can be considered settled or treated with some deep finality. This is a snapshot of ongoing research, here tied together with an overarching theme, but it is both likely and desirable that everything here treated will be supplanted with new greater results in the future. As such this concluding chapter attempts to look forward, while and noting the missing pieces, as well as summarizing some of the aspects of the attached papers that have not yet been brought up.

6.1 Open Questions and Future Directions

6.1.1 Shuffle Questions

There are two open questions from the preceding licentiate thesis [Ber12] that may be interesting to recall.

1. Deciding the membership problem for the shuffle of palindromes:

$$\{ww^{\mathcal{R}} \mid w \text{ is any string, } w^{\mathcal{R}} \text{ is } w \text{ reversed}\}.$$

2. Deciding the membership problem for the language of shuffle squares,

$$\{w \odot w \mid w \text{ is any string}\}.$$

Notice that as the languages we are concerned with are specified as part of the problem these should be viewed as non-uniform membership problems.

The first remains a point of interest, Paper I demonstrates that the non-uniform membership problem for two linear deterministic context-free languages is NP-hard (see Chapter 2), and the shuffle of two palindromes seems like, in a spirit rather similar to the ideas of Paper V, or possibly more illustratively the Chomsky-Schützenberger theorem [CS63], the next step. That is, the palindromes are sort of the most primitive representation of the basic power that differentiates the linear deterministic context-free languages from the regular, in that both the intersection and homomorphism in the Chomsky-Schützenberger decomposition of it do "nothing". The author has no speculation whether this problem should be expected to be in P or not.

The shuffle square, on the other hand, has seen some important developments since [Ber12], and is proven NP-complete in a rather tricky reduction in [BS13].

In addition, let us note that the problems as stated above deal with languages with arbitrarily large alphabets (i.e., when it says that w is any string it may be over an alphabet up to $|w|$ in size). The reduction in [BS13] works for a finite alphabet version of the shuffle square as well, meaning that the language is NP-complete either way. No results are known for the palindrome shuffle, so a possibility is that the problem is NP-complete for arbitrarily large alphabets, but is in P for all alphabet sizes smaller than some constant.

Beyond that, there are numerous additional problems that may be considered in shuffle, especially as many aspects are of practical interest. Beyond simply improving on many of the results in Paper I, and considering both more generalized and restricted cases (shuffle on trajectories is a lively and interesting case), the problem the author most wants to highlight is the one considered in Paper V. That is, proving that for all context-free languages $L \subseteq \Sigma^*$ and $L' \subseteq \Gamma^*$ (with $\Sigma \cap \Gamma = \varnothing$) the shuffle $L \odot L'$ is context-free if and only if one of L and L' is regular.

6.1.2 Synchronized Substrings Questions

The synchronized substrings formalisms, such as linear context-free rewriting systems, are a prime example of where the details of parsing complexity are hugely important. The uniform membership algorithm appears inefficient from the classical complexity theory perspective, but in practice the algorithms are considered reasonably efficient (recall Section 3.4.3). Paper II does find some potentially efficient cases, but they are not necessarily entirely satisfactory, as the one truly efficient case identified is where the rank, fan-out *and* derivation length are included in the parameter (i.e., if all three are small the parsing problem is efficient).

The most obvious case not yet studied is to take the opposite approach from the classical non-uniform membership problem[1]; we let the length of the string be the parameter and consider the grammar in full, or near full. To see the reasoning here the intended application may need to be clarified. These formalisms are typically used for natural language processing. In this case it is easy to see that the sizes of the components are backwards from what is usually assumed, the strings are simply natural language sentences, and, while they can be long, like this run-on sentence, there are still very real practical limits on how many words there can be in one. A *reasonably complete* grammar for English however is vast at the best of times, simply enumerating exceptions will create tens of thousands of rules. As such the complexity in the grammar is actually more important than the complexity in the string.

6.1.3 Regular Expression Questions

The two Papers III and IV both deal with very similar issues, in that they are motivated by the order-dependencies that exist in practical regular expression semantics as an effect of matching methodology employed. Their approaches are very different how-

[1] Notice however, here we talk about parameterized complexity, the intent is not to clumsily assume some parts constant like in the non-uniform case. Parameterized complexity bounds still take the parts put in the "parameter" into account, but differentiate between how large a role that part plays in the complexity. See Section 3.5.1.

ever, in that Paper III attempts to bring an approximation (attempting to make them behave nicely within the classical framework) of these effects into a formal framework, whereas Paper IV tries to analyze the actual state of being of these regular expression engines using formal techniques. The way forward here is not immediately obvious, there are clear open questions that follow directly from Paper III (e.g. an upper bound on the automaton size), as well as some mechanical improvements already considered. On the other hand Paper IV having a continuation is to a great extent a question of impact, as the paper may very well inspire changes in regular expression engines, which would make continued research chase a moving target. A possibility which has both advantages and disadvantages.

As such there is a wealth of possible work in the area of regular expression semantics, but beyond incremental open questions which are already listed in the papers themselves this direction depends on the expected and actual impact of the research.

6.1.4 Other Questions

Paper V is obviously a work in progress published primarily for inclusion in this thesis. The conjecture presented does, however, appear very promising and significant, far beyond proving the open question for context-free shuffles discussed above. In the other direction, Paper VI is the oldest paper included, and is concerned with a direction that has not gotten a high level of attention from the author since. Continuing work appears to be a matter of extending the discussion in Chapter 5 in a way that arrives at a reasonably compelling language class, while having clearly motivated fixed parameter complexity problem with a positive outcome.

6.2 Conclusion

As a final remark the author wishes to again thank all his collaborators and colleagues, as well as everyone who worked on the many pieces of research leveraged as preliminaries in this work. Finally, the author thanks the reader for the interest shown.

References

[Bar85] G. Edward Barton. On the complexity of ID/LP parsing 1. *Computational Linguistics*, 11(4):205–218, 1985.

[BBB13] Martin Berglund, Henrik Björklund, and Johanna Björklund. Shuffled languages – representation and recognition. *Theoretical Computer Science*, 489-490:1–20, 2013.

[BBD13a] Martin Berglund, Henrik Björklund, and Frank Drewes. On the parameterized complexity of Linear Context-Free Rewriting Systems. In *Proceedings of the 13th Meeting on the Mathematics of Language (MoL 13)*, pages 21–29, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[BBD+13b] Martin Berglund, Henrik Björklund, Frank Drewes, Brink van der Merwe, and Bruce Watson. Cuts in regular expressions. In Marie-Pierre Béal and Olivier Carton, editors, *Proceeding of the 17th International Conference on Developments in Language Theory (DLT 2013)*, pages 70–81, 2013.

[BDvdM14] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. Submitted to the 14th International Conference on Automata and Formal Languages (AFL 2014), 2014.

[Ber11] Martin Berglund. Analyzing edit distance on trees: Tree swap distance is intractable. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 59–73. Prague Stringology Club, Czech Technical University, 2011.

[Ber12] Martin Berglund. *Complexities of Parsing in the Presence of Reordering*. Licentiate thesis, Umeå University, 2012.

[Ber14] Martin Berglund. Characterizing non-regularity. Technical Report UMINF 14.12, Computing Science, Umeå University, `http://www8.cs.umu.se/research/uminf/`, 2014. In collaboration with Henrik Björklund and Frank Drewes.

[Bil05] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3):217–239, 2005.

[BN01] Eberhard Bertsch and Mark-Jan Nederhof. On the complexity of some extensions of rcg parsing. In *IWPT*, 2001.

[Bou98] Pierre Boullier. Proposal for a Natural Language Processing Syntactic Backbone. Research Report RR-3342, INRIA, 1998.

[Bou04] Pierre Boullier. *Range concatenation grammars*, pages 269–289. Kluwer Academic Publishers, Norwell, MA, USA, 2004.

[Brz64] Janusz Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[BS13] Sam Buss and Michael Soltys. Unshuffling a square is np-hard. *Journal of Computer and System Sciences*, 2013.

[CS63] Noam Chomsky and Marcel Paul Schützenberger. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, Amsterdam, 1963.

[Dam64] Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

[EB98] Zoltan Ésik and Michael Bertol. Nonfinite axiomatizability of the equational theory of shuffle. *Acta Informatica*, 35(6):505–539, 1998.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Springer-Verlag, 2006.

[Gaz88] Gerald Gazdar. Applicability of indexed grammars to natural languages. In Uwe Reyle and Christian Rohrer, editors, *Natural Language Parsing and Linguistic Theories*. Reidel Dordrecht, 1988.

[Gis81] Jay L. Gischer. Shuffle languages, Petri nets, and context-sensitive grammars. *Communications of the ACM*, 24(9):597–605, 1981.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[Glu61] Victor Michailowitsch Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1–53, 1961.

[Göt08] Daniel Norbert Götzmann. Multiple context-free grammars. Technical report, Universität des Saarlandes, 2008.

[GS65] Seymour Ginsburg and Edwin H. Spanier. Mappings of languages by two-tape devices. *J. ACM*, 12:423–434, July 1965.

[Hab92] Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.

[HMU03] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Ed.)*. Pearson Education International, 2003.

[JLT75] Aravind K. Joshi, Leon S. Levy, and Masako Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975.

[Jos85] Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural description? *Natural Language Processing — Theoretical, Computational and Psychological Perspective*, 1985.

[JS01] Joanna Jedrzejowicz and Andrzej Szepietowski. Shuffle languages are in P. *Theorical Computer Science*, 250(1-2):31–53, 2001.

[JSW90] Aravind K. Joshi, K. Vijay Shanker, and David J. Weir. The convergence of mildly context-sensitive grammar formalisms, 1990.

[Kle98] Philip N. Klein. Computing the edit-distance between unrooted ordered trees. In *In Proceedings of the 6th annual European Symposium on Algorithms (ESA*, pages 91–102. Springer-Verlag, 1998.

[KNSK92] Y. Kaji, R. Nakanisi, H. Seki, and T. Kasami. The universal recognition problem for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems*, E75-D(1):78–88, 1992.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[LW87] Klaus-Jörn Lange and Emo Welzl. String grammars with disconnecting or a basic root of the difficulty in graph grammar parsing. *Discrete Applied Mathematics*, 16:17–30, 1987.

[MRS98] Alexandru Mateescu, Grzegorz Rozenberg, and Arto Salomaa. Shuffle on trajectories: syntactic constraints. *Theoretical Computer Science*, 197(1-2):1–56, 1998.

[MS94] Alain J. Mayer and Larry J. Stockmeyer. Word problems – this time with interleaving. *Information and Computation*, 115:293–311, 1994.

[ORR78] William F. Ogden, William E. Riddle, and William C. Rounds. Complexity of expressions allowing concurrency. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 185–194, New York, NY, USA, 1978. ACM.

[Pol84] Carl Pollard. *Generalized phrase structure grammars, head grammars and natural language*. PhD thesis, Stanford University, 1984.

[Sel77] Stanley M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[Sha78] Alan C. Shaw. Software descriptions with flow expressions. *IEEE Trans. Softw. Eng.*, 4:242–254, May 1978.

[SMFK91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. On multiple context-free grammars. *Theor. Comput. Sci.*, 88(2):191–229, October 1991.

[Ste87] Mark Steedman. Combinatory Grammars and Parasitic Gaps. *Natural Language and Linguistic Theory*, 5:403–439, 1987.

[Tai79] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26:422–433, July 1979.

[VdlC02] Éric Villemonte de la Clergerie. Parsing mildly context-sensitive languages with thread automata. In *Proceedings of the 19th international conference on Computational linguistics - Volume 1*, COLING '02, pages 1–7, Stroudsburg, PA, USA, 2002. Association for Computational Linguistics.

[Wei88] David J. Weir. *Characterizing mildly context-sensitive grammar formalisms*. Graduate School of Arts and Sciences, University of Pennsylvania, 1988.

[Wei92] David J. Weir. Linear context-free rewriting systems and deterministic tree-walking transducers. In *Proceedings of the 30th annual meeting on Association for Computational Linguistics*, ACL '92, pages 136–143, Stroudsburg, PA, USA, 1992. Association for Computational Linguistics.

[WH84] Manfred K. Warmuth and David Haussler. On the complexity of iterated shuffle. *J. Comput. Syst. Sci.*, 28(3):345–358, 1984.

[ZS89] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.

[ZSS92] Kaizhong Zhang, Rick Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Information Processing Letters*, 42(3):133 – 139, 1992.

I

Shuffled Languages – Representation and Recognition[☆]

Martin Berglund[a,*], Henrik Björklund[a,*], Johanna Björklund[a,1,*]

[a] *Computing Science Department, Umeå University, 901 87 Umeå, Sweden*

Abstract

Language models that use interleaving, or shuffle, operators have applications in various areas of computer science, including system verification, plan recognition, and natural language processing. We study the complexity of the membership problem for such models, in other words, how difficult it is to determine if a string belongs to a language or not. In particular, we investigate how interleaving can be introduced into models that capture the context-free languages.

Keywords: Interleaving, shuffle languages, membership problems

1. Introduction

We study the membership problem for various language classes that make use of the shuffle operator $\odot$. When applied to a pair of strings u and v, the operator returns the set of all possible interleavings of the symbols in u and v. For example, the shuffle of ab and cd is $\{abcd, acbd, acdb, cabd, cadb, cdab\}$. This type of interleaving operation has been considered as far back as in a 1965 paper by S. Ginsburg and E. Spanier [22]. The operator is lifted to languages by defining $\mathcal{L}_1 \odot \mathcal{L}_2$ to be the set $\bigcup\{u \odot v \mid u \in \mathcal{L}_1, v \in \mathcal{L}_2\}$. We also consider the shuffle closure operator, whose relationship to the shuffle operator resembles that of the Kleene star to concatenation, iterating the shuffle operation. As our starting point, we take the *shuffle languages* considered by Gischer [23] and by Jedrzejowicz and Szepietowski [31]. These are the languages defined by regular expressions augmented with the shuffle operation and the shuffle closure operators, inspired by Flow Expressions [43].

Shuffling of languages is of interest in a number of different areas:

- One of the original motivations for studying shuffle is in the *modelling* and *verification* of systems, where shuffling is useful for reasoning about

[☆]The present article extends [3], which was presented at the *5th International Conference on Language and Automata Theory and Applications (LATA) 2011.*

*Corresponding author

Email addresses: mbe@cs.umu.se (Martin Berglund), henrikb@cs.umu.se (Henrik Björklund), johanna@cs.umu.se (Johanna Björklund)

[1]Née Högberg

interleaved or parallel processes [19, 43, 41]. There is a close connection between shuffle languages and Petri nets [23, 19, 8, 6].

- The shuffle operator is used in *XML database systems* for schema definitions, see for example the work on schema languages by Gelade et al. [20].
- In *plan recognition*, the challenge is to identify an agent's objectives, based on observations of the its actions [11, 42]. In a generalized version, a number of independent agents perform their actions in an interleaved fashion. To model such a scenario, one could combine shuffle operators and context-free grammars [29]. For this approach to be tractable, the membership problem for the resulting languages must remain efficiently solvable.
- In *natural language processing*, there is a growing interest in linguistic models for languages with relatively free word order. Recent work in this direction includes parse algorithms for dependency grammars [39, 32].

Many fundamental questions regarding the membership problem for shuffled languages remain unanswered. We consider and answer some of them in this paper. In particular, we are interested in language classes that capture the context-free languages. Among the above application areas, such languages are primarily of interest in plan recognition and natural language processing.

It is important to distinguish between the *uniform* and the *non-uniform* version of the membership problem. In the uniform version, both the string and a representation of the language is given as input. It is therefore relevant *how* the language is represented. In the non-uniform version, only the string to be tested is considered as input. The language is fixed, so its representation never enters into the equation.

Contributions. To facilitate the study of languages combining restricted forms of recursion and interleaving, we define *Concurrent Finite State Automata* (CFSA) which have an expressive power between those of context-free grammars and context-sensitive grammars. These automata can be viewed as ground tree rewriting systems (see, e.g., [33, 12]) used as language acceptors. We show that the emptiness problem for CFSA is solvable in polynomial time, list the closure properties of the automata, and identify the language classes that correspond to certain syntactic restrictions.

Our results for the complexity of the membership problems for various language classes are summarized in Table 1. It should be noted that all problems we consider, except the membership problem for CFSA, are trivially in NP. For the full class of languages recognized by CFSA, we show that both the uniform and the non-uniform membership problem are NP-complete.

For the *shuffle languages* (as used in [23, 31]), the *uniform membership problem* is NP-complete [44, 2, 35], while the *non-uniform membership problem* can be decided in polynomial time [31]. We shed further light on the complexity of the membership problem by establishing that the uniform version, when parameterized by the number of shuffle operations, is hard for the complexity class

Table 1: Summary of results for the membership problem. The shuffle languages are abbreviated by Sh, the regular by Reg, and the deterministic linear context-free by DLCF. The results of this paper appear in bold face. In the case of CFSA membership NP-hardness follows from [40] and inclusion is demonstrated here.

	Sh	Reg $\odot$ CF	Sh $\odot$ CF	DLCF $\odot$ DLCF	CFSA
Non-Uniform	P	**P**	**P**	**NPC**	**NPC**
Uniform	**W[1]-hard**	**P**	NPC	**NPC**	**NPC**

W[1]. This result suggests a strong dependence on the number of shufflings. For this reason, we do not expect to find a particularly efficient algorithmic solution to the non-uniform membership problem for language definitions involving many shufflings, even when it is theoretically polynomial.

For the interleaving of a regular language and a context-free language, we show that the uniform (and thus also the non-uniform) membership problem can be solved in polynomial time. The regular language is assumed to be represented by a nondeterministic finite automaton and the context-free language by a context-free grammar.

For the shuffling of a shuffle language and a context-free language, the uniform problem is NP-hard, since this holds already for the shuffle languages. In one of our main results, we show that the non-uniform problem, on the other hand, is solvable in polynomial time.

It is known that already the non-uniform version of the membership problem is NP-hard for the shuffling of two deterministic context-free languages [40]. We strengthen this result by demonstrating that it holds even for the shuffle of deterministic *linear* context-free languages. Here, the results in [28] may also be of interest, since it draws parallels between two-stack Turing machines and the shuffle of context-free languages, which is similar to the technique we use.

It should be noted that we only investigate which broad complexity classes the problems belong to. In particular, for the problems that belong to P, our aim has not been to find optimal algorithms. Future work in this direction includes finding the exact complexities of these problems, as well as heuristic algorithms and tractable restrictions of the NP-complete problems.

Related work. Various aspects of shuffling have been studied in formal language theory and its effects on regular languages have received particular interest. Câmpeanu et al. establish $2^{mn} - 1$ as a tight upper bound on the state complexity of the shuffle of two regular languages [10], represented by finite deterministic automata of size m and n, respectively. Biegler et al. provide a similar result for singleton languages and identify properties that trigger an exponential blow-up in state complexity [5]. On the descriptive side, it follows from a result by Gruber and Holzer that the addition of a shuffle operator to regular expressions may reduce representation sizes exponentially [25]. A generation algorithm with linear complexity for approximate size sampling (i.e., random generation) of regular specifications including shuffle has been provided

by Darrasse et al. [14]. Brozozowski et al. consider the complexity of *ideal* languages [9], which are regular languages invariant under shuffle with the universal language [26]. Further results for sub-families of the regular languages are found in [24, 27, 4, 13]. Warmuth and Haussler gives complexity results (NP-completeness proofs) for some specific and very simple shuffle languages, notably, given two strings v and w as input deciding whether $v \in w^{\odot}$ is NP-complete [44].

Shuffling has also been investigated in a more algebraic setting. The axiomatization of shuffle theory was addressed by Ésik together with Bloom [7] and Bertol [16]. Another important direction is shuffling on *trajectories*, which is the idea of shuffling two strings in a way informed by a third string, the trajectory. That is, for example $abc \odot_{011010} def = adebfc$ where the binary string is the trajectory which identifies a specific interleaving of abc and def. This is then generalized to languages, which makes the shuffle considered here a special case, where the trajectory is always the universal language (e.g., $x \odot y$ is equivalent to $x \odot_{\{0,1\}^*} y$). See [34] for more information on shuffle on trajectories.

Another related formalism is *permutation languages*, first considered by Nagy [37, 38], which allow rules of the form $AB \to BA$ in an otherwise normal context-free grammar. These rules can be applied to interchange positions of adjacent non-terminals in intermediary derivation steps, and thus allows for certain forms of shuffling.

2. Preliminaries

Sets and numbers. If S is a set, then $|S|$ denotes the cardinality of S, S^* is the set of all finite sequences of elements of S, and $\mathit{precl}(S)$ is the set of all finite prefix-closed subsets of S^*. In other words, for every $S' \in \mathit{precl}(S)$, if $uv \in S'$ for some $u, v \in S^*$ then $u \in S'$. We write $\mathbb{N}$ for the natural numbers. For $k \in \mathbb{N}$, we write $[k]$ for $\{1, \ldots, k\}$. Note that $[0] = \emptyset$. The domain of a mapping f is denoted $\mathit{dom}(f)$.

A *total order* on a set S is an binary relation $\leq$ on S that is antisymmetric, transitive, and such that for every $s, s' \in S$, either $s \leq s'$ or $s' \leq s$.

An *alphabet* is a finite nonempty set. Let Σ be an alphabet and let ε be the empty string, then $\Sigma \cup \{\varepsilon\}$ is denoted by Σ_ε. The length of a string $w = \alpha_1 \cdots \alpha_n$ is written $|w|$, and for every $\alpha \in \Sigma$, $|w|_\alpha = |\{i \in [n] \mid \alpha_i = \alpha\}|$. The *reversal* of $w = \alpha_1 \cdots \alpha_n$ is $w^{\mathcal{R}} = \alpha_n \cdots \alpha_1$. For strings $w, w' \in \Sigma$ the concatenation is usually denoted ww'. When necessary for clarity, however, the operation is written explicitly as $w \cdot w'$. Concatenation distributes over sets, e.g. for $S, S' \subseteq \Sigma^*$ we have $w \cdot S = \{w \cdot w' \mid w' \in S\}$, and $S \cdot S' = \{w \cdot w' \mid w \in S, w' \in S'\}$.

Trees. The set T_Σ of *(unranked) trees* over the alphabet Σ consists of all mappings $t \colon D \to \Sigma$, where $D \in \mathit{precl}(\mathbb{N})$. The *empty tree*, denoted t_ε, is the unique tree such that $\mathit{dom}(t) = \emptyset$. We henceforth refer to $\mathit{dom}(t)$ as the *nodes of* t and write $\mathit{nodes}(t)$ rather than $\mathit{dom}(t)$. The size of a tree $t \in T_\Sigma$, denoted $\mathit{size}(t)$, is $|\mathit{nodes}(t)|$. The height of t, denoted $\mathit{height}(t)$, is $1 + \max(n \mid \alpha_1 \cdots \alpha_n \in \mathit{nodes}(t))$.

For a tree $t \in T_\Sigma$ and a node $v \in nodes(t)$, the *subtree of t rooted at v* is denoted by t/v. It is defined by $nodes(t/v) = \{v' \in \mathbb{N}^* \mid vv' \in nodes(t)\}$ and, for all $v' \in nodes(t/v)$, $(t/v)(v') = t(vv')$. The *leaves* of t is the set $leaves(t) = \{v \in \mathbb{N}^* \mid \nexists i \in \mathbb{N}\ s.t.\ vi \in nodes(t)\}$. The *substitution* of t' into t at node v is denoted $t[\![v \leftarrow t']\!]$. It is defined by

$$nodes(t[\![v \leftarrow t']\!]) = (nodes(t) \setminus \{vu \mid u \in \mathbb{N}^*\}) \cup \{vu \mid u \in nodes(t')\} \ ;$$

and, for every $u \in nodes(t[\![v \leftarrow t']\!])$, if $u = vv'$ for some $v' \in nodes(t')$ then $t[\![v \leftarrow t']\!](u) = t'(v')$, otherwise $t[\![v \leftarrow t']\!](u) = t(u)$.

For a tree $t \in T_\Sigma$ let $v_1, \ldots, v_k \in nodes(t)$ be the immediate child nodes of the root ordered by numeric value. That is, $\{v_1, \ldots, v_k\} = \{v \in nodes(t) \mid |v| = 1\}$, ordered such that $v_i < v_{i+1}$ for all $i \in [k-1]$. Then we will write t as $f[t_1, \ldots, t_k]$, where $f = t(\varepsilon)$ and $t_j = t/v_j$ for all $j \in [k]$. In the special case where $k = 0$ (i.e., when $nodes(t) = \{\varepsilon\}$), the brackets may be omitted, thus denoting t as f.

Shuffle operations and shuffle expressions. We recall the definitions of the operations shuffle and shuffle closure, and of shuffle expressions, from [23, 31].

The *shuffle* operation $\odot : \Sigma^* \times \Sigma^* \to pow(\Sigma^*)$ is inductively defined as follows: for every $u \in \Sigma^*$ it is given by $u \odot \varepsilon = \varepsilon \odot u = \{u\}$, and by

$$\alpha_1 u_1 \odot \alpha_2 u_2 = \{\alpha_1 w \mid w \in (u_1 \odot \alpha_2 u_2)\} \cup \{\alpha_2 w \mid w \in (\alpha_1 u_1 \odot u_2)\} \ ,$$

for every $\alpha_1, \alpha_2 \in \Sigma$, and $u_1, u_2 \in \Sigma^*$. The operation extends to languages with

$$\mathcal{L}_1 \odot \mathcal{L}_2 = \bigcup_{u_1 \in \mathcal{L}_1, u_2 \in \mathcal{L}_2} u_1 \odot u_2 \ .$$

The *shuffle closure* of a language $\mathcal{L} \in \Sigma^*$, denoted $\mathcal{L}^{\odot}$, is

$$\mathcal{L}^{\odot} = \bigcup_{i=0}^{\infty} \mathcal{L}^{\odot_i}, \text{ where } \mathcal{L}^{\odot_0} = \{\varepsilon\} \text{ and } \mathcal{L}^{\odot_i} = \mathcal{L} \odot \mathcal{L}^{\odot_{i-1}} \ .$$

Shuffle expressions are regular expressions that can additionally use the shuffle operators. The shuffle expressions over the alphabet Σ are as follows. The empty string ε, the empty set $\emptyset$, and every $\alpha \in \Sigma$ is a shuffle expression. If s_1 and s_2 are shuffle expressions, then so are $(s_1 \cdot s_2), (s_1 + s_2), (s_1 \odot s_2), s_1^*$, and ${s_1}^{\odot}$. Shuffle expressions that do not use the shuffle closure operator are said to be *closure free*. The language $\mathcal{L}(s)$ of a shuffle expression s is defined in the usual way. *Shuffle languages* are the languages defined by shuffle expressions.

3. Concurrent Finite-State Automata

In this section, we introduce *concurrent finite-state automata* (CFSA). They are inspired by *recursive Markov models*, but differs from these in two aspects: the global state space is not partitioned into component automata and, more importantly, they differ in that recursive calls can be made in parallel. The latter feature allows for an unbounded number of invocations to be executed

simultaneously, but each symbol can only be read by one invocation. In Definition 1, the string $p^{\odot}$ is to be read as single symbol. In the later definition of CFSA semantics, transitions of the form $(q, \alpha, q'[p^{\odot}])$ will be interpreted as rule schema.

Definition 1 (CFSA). A *Concurrent FSA* is a tuple $M = (Q, \Sigma, \delta, I)$, where

- Q is a finite set of *states*;
- Σ is an alphabet of *input symbols*;
- $\delta \subseteq Q \times \Sigma_\varepsilon \times T$ is a set of *transitions*, where T is the finite set

$$\{q, q[p], q[p, p'], q[p^{\odot}] \mid q, p, p' \in Q\} \cup \{t_\varepsilon\} \ .$$

  A transition $(q, \alpha, t) \in \delta$ is

  - *terminal* if $|nodes(t)| = 0$, in this case it must also hold that $\alpha = \varepsilon$,
  - *horizontal* if $|nodes(t)| = 1$, and
  - *vertical* if $|nodes(t)| > 1$.

- $I \subseteq Q$ is a set of *initial* states. □

We now establish the semantics of CFSA. Whereas a FSA is in a single state at a time, a concurrent FSA maintains a branching call-stack of states, represented by an unranked tree over an alphabet of states. In each step, exactly one leaf node of the state tree is rewritten. Vertical transitions model the invocation of child processes; horizontal transitions the continued execution within a process; and terminal transitions the completion of a process. A CFSA accepts a string if, upon reading the entire string, it can reach a configuration in which every processes has been completed, i.e., the state tree is empty.

Definition 2 (Concurrent FSA semantics). A *configuration* of the CFSA $M = (Q, \Sigma, \delta, I)$ is a tuple $(w, t) \in \Sigma^* \times T_Q$. The set of all configurations of M is denoted $\Delta(M)$. A configuration $(w, t) \in \Delta(M)$ is *initial* (with respect to the string $w \in \Sigma^*$) if $t \in I$.

Consider the configurations $(w, t), (w', t') \in \Delta(M)$. There is a *transition step* from (w, t) to (w', t'), written $(w, t) \to (w', t')$, if there is a transition $(q, \alpha, s) \in \delta$ and a node $v \in nodes(t)$ such that $w = \alpha w'$, $t/v = q$ (so v is a leaf), and either

- $s \in T_Q$ and $t' = t[\![v \leftarrow s]\!]$, or
- $s = p'[p^{\odot}]$ and $t' = t[\![v \leftarrow p'[\underbrace{p, \ldots, p}_{n}]]\!]$ for some for $p, p' \in Q$ and $n \in \mathbb{N}$.

The reflexive and transitive closure of $\to$ is denoted $\xrightarrow{*}$. The *language recognized by* M is $\mathcal{L}(M) = \{w \in \Sigma^* \mid \exists q \in I : (w, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)\}$. □

For the sake of brevity only the state-tree part of a configuration, called a *configuration tree*, may be shown in cases where the string is irrelevant.

Remark. For simplicity we will assume that the *terminal* transitions of a CFSA form a subset of $Q \times \{\varepsilon\} \times \{t_\varepsilon\}$, that is, we assume that terminal transitions do not read symbols. This causes no loss of generality with respect to the recognised string language, since a CFSA can be rewritten to fulfil this requirement in linear time by adding a designated terminal state q (the only transition for q is $(q, \varepsilon, t_\varepsilon)$), and change all other terminal rules $(q', \alpha, t_\varepsilon)$ into the horizontal rule (q', α, q).

Example 1. Recall that a Dyck language consists of all well-balanced strings over a given set of parentheses. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the Dyck languages over the symbol pairs $\lfloor, \rfloor$ and $\lceil, \rceil$, respectively. Their shuffle $\mathcal{L} = \mathcal{L}_1 \odot \mathcal{L}_2$ is recognized by the concurrent FSA $M = (\{q_0, q_0', q_1, q_1', q_2, q_2'\}, \{\lfloor, \rfloor, \lceil, \rceil\}, \delta, \{q_0\})$, where

$$\delta = \{\ (q_0, \varepsilon, q_0'[q_1, q_2]),\quad (q_0', \varepsilon, t_\varepsilon),\quad (q_1, \lfloor, q_1'[q_1]),\quad (q_1', \rfloor, q_1),$$
$$(q_1, \varepsilon, t_\varepsilon),\quad (q_2, \lceil, q_2'[q_2]),\quad (q_2', \rceil, q_2),\quad (q_2, \varepsilon, t_\varepsilon)\qquad \}\ .$$

To illustrate the automaton's semantics, we step through an accepting run of M on the string $w = \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor$ (see Figure 1). Note that since $w \in w_1 \odot w_2$ for $w_1 = \lfloor\lfloor\rfloor\lfloor\rfloor\rfloor \in \mathcal{L}_1$ and $w_1 = \lceil\rceil \in \mathcal{L}_2$, it follows that $w \in \mathcal{L}_1 \odot \mathcal{L}_2$. □

It is known that $\mathcal{L}_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ is a context-free language, but it is not a shuffle language. Conversely, $\mathcal{L}_2 = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$ is a shuffle language but is not context-free. Both $\mathcal{L}_1$ and $\mathcal{L}_2$ is recognized by a CFSA, and so is $\mathcal{L}_1 \cup \mathcal{L}_2$, which is neither a context-free nor a shuffle language. As will be shown below, the CFSA languages properly extend both the context-free languages and the shuffle languages. They also have comparatively nice closure properties.

Theorem 1. *The languages recognized by CFSA are closed under union, concatenation, Kleene star, shuffle and shuffle closure. They are not closed under intersection with a regular language or complementation.*

Proof. Let $M = (Q, \Sigma, \delta, I)$ and $M' = (Q', \Sigma, \delta', I')$ be CFSA. We assume without loss of generality that $Q \cap Q' = \emptyset$, and that the automata have only one initial state each, i.e., $I = \{q_0\}$ and $I' = \{q_0'\}$. The latter assumption can be made without loss of recognizing power since ε-transitions are allowed.

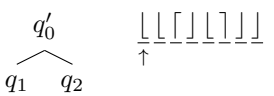
Union. A CFSA for the union of M and M' can be constructed by adding a new initial state q together with ε-transitions from q to each of q_0 and q_0'.

Concatenation. A CFSA for the the concatenation of M with M' can be constructed by adding a new initial state q, new states q' and q'', and the transitions $(q, \varepsilon, q'[q_0])$, $(q', \varepsilon, q''[q_0'])$, and $(q'', \varepsilon, t_\varepsilon)$. This allows the automaton to first simulate a run of M and then a run of M'.

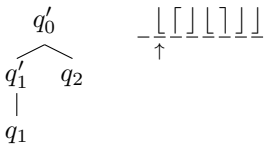
Kleene closure. A CFSA for the Kleene closure of M can be constructed by adding a new initial state q and the transitions $(q, \varepsilon, t_\varepsilon)$ and $(q, \varepsilon, q[q_0])$. This allows the automaton to simulate any number of runs of M, one after the other.

Initial configuration:

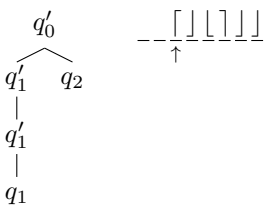
Via transition $(q_0, \varepsilon, q_0'[q_1, q_2])$:



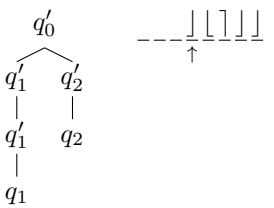
Via transition $(q_1, \lfloor, q_1'[q_1])$:



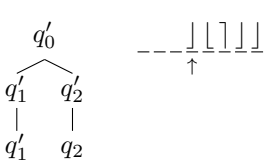
Via transition $(q_1, \lfloor, q_1'[q_1])$:



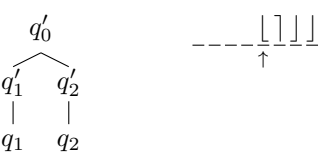
Via transition $(q_2, \lceil, q_2'[q_2])$:



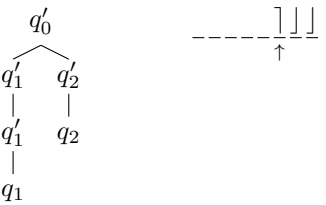
Via transition $(q_1, \varepsilon, t_{\varepsilon:})$



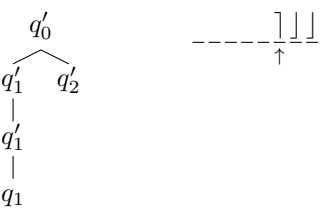
Via transition $(q_1', \rfloor, q_1)$:



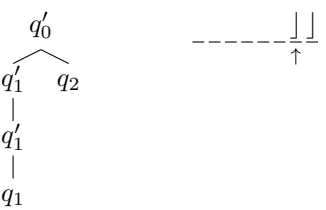
Via transition $(q_1', \lfloor, q_1'[q_1])$:



Via transition $(q_2, \varepsilon, t_{\varepsilon:})$



Via transition $(q_2', \rceil, q_2)$:



After the sequence of transitions $(q_1, \varepsilon, t_\varepsilon), (q_1', \rfloor, q_1)$, twice applied:

Via $(q_1, \varepsilon, t_\varepsilon)$, followed by $(q_2, \varepsilon, t_\varepsilon)$ twice in a row we obtain q_0', then via $(q_0', \varepsilon, t_\varepsilon)$ we arrive at the empty state tree t_ε, so the run is accepting.

Figure 1: An accepting run of the CFSA M on input $\lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor$.

Shuffle. For the shuffle of $\mathcal{L}(M)$ and $\mathcal{L}(M')$ we add states q, q', where q becomes the unique initial state of the new automaton. We also add the vertical transition $(q, \varepsilon, q'[q_0, q_0'])$ and the terminal transition $(q', \varepsilon, t_\varepsilon)$.

Shuffle closure. To construct the shuffle closure of the language of M, we again add states q, q', where q becomes the unique initial state of the new automaton. Additionally, we add the vertical transition $(q, \varepsilon, q'[q_0{}^{\odot}])$ and the terminal transition $(q', \varepsilon, t_\varepsilon)$. This allows the new automaton to spawn any number of copies of M that can then run in parallel over the input string.

Intersection. Consider the languages $\mathcal{L}_1 = (abc)^{\odot}$ and $\mathcal{L}_2 = a^*b^*c^*$. The former is a shuffle language, and the latter clearly a regular language, so both are recognizable by CFSA. As we shall see, their intersection $\mathcal{L} = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not. The proof is by contradiction, so let us assume that $\mathcal{L}$ is recognized by some CFSA $M = (Q, \Sigma, \delta, I)$.

To make the upcoming argument clearer, we introduce some convenient definitions. For every $q \in Q$, M_q denotes the CFSA $(Q, \Sigma, \delta, \{q\})$. The *substrings* of a language $\mathcal{L}$, written $substring(\mathcal{L})$, is the set $\{v \mid uvw \in \mathcal{L} \text{ for some } u, w \in \Sigma^*\}$.

Now, if a transition r of the form $(q, \alpha, q'[p, p']) \in \delta$ is applied in an accepting run of M, then $\mathcal{L}(M_p) \odot \mathcal{L}(M_{p'}) \subseteq substrings(\mathcal{L})$. For this reason, $\mathcal{L}(M_p) \cup \mathcal{L}(M_{p'}) \subseteq \alpha^*$ for some $\alpha \in \{a, b, c\}$. Otherwise, if for example $w \in \mathcal{L}(M_p)$ and $w' \in \mathcal{L}(M_{p'})$ with $|w|_a > 0$ and $|w'|_b > 0$, the string $w'w \in w \odot w'$ would be in $substrings(\mathcal{L})$, but this is impossible since a b occurs before an a in $w'w$. It follows that the order of p and p' in r is irrelevant. Hence, r can equivalently be replaced by a pair of transitions such that $(\varepsilon, q) \xrightarrow{*} (\alpha, q'[p[p']])$. The same argument justifies the replacement of transitions of the form $(q, \alpha, q'[p^{\odot}])$ with transitions that yield $(\varepsilon, q) \xrightarrow{*} (\alpha, q'[p[p[\ldots[p]]]])$.

After this language-preserving normalization, the resulting CFSA only generates monadic configuration trees, which means that no shuffling is done. However, without shuffle operations, $\mathcal{L}(M)$ is a context-free language (cnf. Theorem 2), and it is well known that $\mathcal{L}$ is not a context-free language. Consequently, $\mathcal{L}$ is not recognizable by a CFSA.

Complementation. Since the CFSA languages are closed under union, but not under intersection, they are not closed under complementation either, since $(L_1 \cap L_2)$ can be expressed as $\overline{(\overline{L_1} \cup \overline{L_2})}$. □

Restrictions and expressive power. We introduce CFSA to provide an automaton model that can be syntactically restricted to capture the combination of shuffle operations with some well-known languages classes. The restrictions considered here are as follows. A CFSA $M = (Q, \Sigma, \delta, I)$ is:

- *horizontal* if δ contains no vertical transitions;
- *non-branching* if every vertical transition is in $Q \times \Sigma \times \{q'[q] \mid q, q' \in Q\}$;
- *finitely branching* if no vertical transition is in $Q \times \Sigma \times \{q'[q^{\odot}] \mid q, q' \in Q\}$;
- *acyclic* if there is no configuration $(w, t) \in \Delta(M)$ and state $q \in Q$ such that q appears twice on a path from the root of t to a leaf.

Theorem 2. *A language is:*

- *regular if and only if it is recognized by a horizontal CFSA;*
- *context-free if and only if it is recognized by a non-branching CFSA;*
- *a shuffle language if and only if it is recognized by an acyclic CFSA;*
- *a closure-free shuffle language if and only if it is recognized by an acyclic and finitely branching CFSA.*

Proof sketch. Horizontal CFSA are equivalent to nondeterministic finite automata in that they recognize the regular languages.

It is easy to turn a context-free grammar $G = (N, \Sigma, \gamma, S)$ in Chomsky normal form into a non-branching CFSA $M = (Q, \Sigma, \delta, I)$. Let $Q = N \cup \{\overline{q} \mid q \in N\}$, $I = \{S\}$, and define δ as follows.

- For every rule $q \to \alpha$ in γ, where $\alpha \in \Sigma_\varepsilon$, there is a horizontal transition $(q, \alpha, \overline{q})$ and a terminal transition $(\overline{q}, \varepsilon, t_\varepsilon)$ in δ.
- For every rule $q \to pp'$ in γ, there is a transition $(q, \varepsilon, p'[p])$ in δ_2.

For the opposite direction, it is equally easy to turn a non-branching CFSA into a language-equivalent push-down automaton.

Next, we show that acyclic CFSA correspond to the shuffle languages. The only-if direction follows directly from the proof of Theorem 1 since the constructions there preserve automata acyclicity.

Given an acyclic CFSA $M = (Q, \Sigma, \delta, I)$ we show how to construct a shuffle expression s recognizing $\mathcal{L}(M)$. Two states $q, q' \in Q$ are said to be *connected* if there is a transition $(q, \alpha, t) \in \delta$, where the label of the root of t is q', for some $\alpha \in \Sigma_\varepsilon$. With this notion of connectivity, let $C_1, \ldots, C_k$ be the connected components of M. Consider the directed graph $G_M = (C_1, \ldots, C_k, E)$, where $(C_i, C_j) \in E$ if there is a state $q \in C_i$, a *vertical* transition $(q, \alpha, t) \in \delta$, and a state $p \in C_j$ such that p (or $p^\odot$) labels a leaf of t. Since M is acyclic, also G_M is acyclic.

Let $\delta_v \subseteq \delta$ be the set of all vertical transitions. We create an alphabet Σ_v with one unique new symbol for each vertical transition. Let $h : \delta_v \to \Sigma_v$ be the bijection mapping each $d \in \delta_v$ to the corresponding alphabet symbol. Also, for each $d \in \delta_v$, let q_d be a new state. Define H to be the CFSA obtained from M by replacing each vertical transition $d = (q, \alpha, q'[...])$ with the horizontal transitions (q, α, q_d) and $(q_d, h(d), q')$. Notice that the connected components of H are the same as the connected components of M and that H is a finite automaton recognizing a regular language.

For each $q \in Q$, let the *regular* expression $r(q)$ be such that $\mathcal{L}(r(q)) = \mathcal{L}(H_q)$, that is, $r(q)$ describes the language that H recognizes when starting from state q. Such a regular expression can be computed using standard constructions.

We are now ready to describe how to construct the shuffle expression corresponding to M. To be precise, for each state $q \in Q$, we will define a shuffle

expression $s(q)$ such that the language of $s(q)$ is the language of M_q, in other words, the CFSA obtained from M by replacing I by $\{q\}$. We do this by induction on the structure of G_M.

If C is a leaf of G_M, then there are no vertical transitions in the connected component C. Hence, for every $q \in C$, we have $s(q) = r(q)$.

Suppose that q belongs to a connected component C_i such that for all states in all components reachable from C_i in G_M, we have already computed the corresponding shuffle expressions. In this case we get the shuffle expression for q by taking $r(q)$ and replacing symbols in Σ_v by appropriate shuffle expressions. In particular, consider symbol $h(d) \in \Sigma_v$ that corresponds to $d = (q', \alpha, t) \in \delta_v$. The shuffle expression for $h(d)$ is obtained from t as follows.

- If $t = p[p']$, for some $p, p' \in Q$, then the shuffle expression is $s(p')$.
- If $t = p[p'_1, p'_2]$ then the shuffle expression is $s(p'_1) \odot s(p'_2)$.
- If $t = p[p'^{\odot}]$, then the shuffle expression is $(s(p'))^{\odot}$.

The shuffle expression for M is the union of those for the states in I, i.e.,

$$s = \bigcup_{q \in I} s(q) \ .$$

The equivalence $\mathcal{L}(M) = \mathcal{L}(s)$ can be shown by a standard induction.

Finally, that acyclic and finitely branching CFSA correspond to the closure free shuffle languages follows from the constructions in the proof of Theorem 1 as only the shuffle closure operator induces unbounded branching. □

Since the closure free shuffle languages are regular [21], we can conclude that acyclic and finitely branching CFSA also recognize the regular languages.

To see that CFSA do not provide us with the full power of linear bounded Turing machines, we first note that they can be augmented in polynomial time with "shortcuts", that is, contractions of ε-consuming transition sequences into single transitions.

Definition 3 (ε-efficient). A CFSA $M = (Q, \Sigma, \delta, I)$ is *ε-efficient* if it fulfills the following conditions:

1. For every $q \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ then $(\varepsilon, q) \to (\varepsilon, t_\varepsilon)$.
2. For every choice of $q, q' \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, q')$ then $(\varepsilon, q) \to (\varepsilon, q')$.
3. For every choice of $q, q', p, p' \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, q'[p, p']) \xrightarrow{*} (\varepsilon, q'[p])$ then $(\varepsilon, q) \to (\varepsilon, q'[p])$.

Lemma 1. *Every CFSA $M = (Q, \Sigma, \delta, I)$ can be rewritten into a language-equivalent ε-efficient CFSA in polynomial time.*

PROOF (SKETCH). A simple procedure based on the emptiness test (see Theorem 4) suffices to add any missing transition to M in polynomial time. For example, construct the automaton $M' = (Q, \Sigma, \delta', \{q\})$ where $\delta' \subseteq \delta$ contains only the transitions that do not consume any symbol. Then $(\varepsilon, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ if and only if M' is nonempty. Once Condition 1 is satisfied, the transitions needed to satisfy the remaining two conditions can be added through similar constructions. □

Lemma 2. *Let $M = (Q, \Sigma, \delta, I)$ be an ε-efficient CFSA. In a sequence of transition steps that accepts the string w and is of minimum length, no intermediary configuration tree needs to have more than $|w|$ leaves or be of height greater than $|Q|(|w| + 1)$.*

PROOF. Let $c = (w, t)$ and $c' = (w, t')$ be a pair of configurations in $\Delta(M)$. If there is a sequence of ε-transitions from c to c', then there is also a sequence of length at most $n \leq size(t) + 2size(t')$. Such a short sequence can be found by organizing the transitions as follows: $(w, t) \xrightarrow{*} (w, \hat{t}) \xrightarrow{*} (w, t')$ where the $t \to \hat{t}$ part of the derivation *only deletes nodes*, and the $\hat{t} \to t'$ part *never deletes nodes*. This reorganization is possible since M is ε-efficient, so all possible node deletions/relabelings can be performed without generating extraneous nodes. In turn, this means that no node needs to be generated only to subsequently be deleted. It follows that at most $|nodes(t)|$ may need to be deleted, and at most $|nodes(t')|$ nodes may need to be created and/or relabeled with a new state.

Consider an accepting sequence of transitions of minimal length. Only $|w|$ symbols are consumed by the transitions, so if there are $|w| + 1$ leaves in any intermediate configuration tree, then one of them must consume ε. The existence of such a leaf violates the assumption that the sequence is of minimal length (notice that conditions 1–3 in Definition 3 ensure that useless nodes never have to be added). The height bound holds since a higher tree would have $|w| + 2$ or more copies of some state q along some path. With $|w| + 2$ instances of q-labeled nodes, there are $|w| + 1$ such q-delimited sections on the path. Only $|w|$ symbols are consumed, so one of those sections will be matched up against the empty string. The redundant section could be omitted without affecting the accepted string, which violates the assumption that the original sequence was of minimum length. □

Theorem 3. *The languages recognized by CFSA are properly contained in the context-sensitive languages.*

PROOF. Let $M = (Q, \Sigma, \delta, I)$ be a CFSA and w string. If there is an accepting run of M on w from an initial state q_0, then a nondeterministic Turing machine can guess and verify this run in linear space by a depth-first left-to-right search.

The TM simulates a run of M on w starting from q_0, but when a vertical transition $(q, \alpha, q'[s])$ is used, where s is a sequence of labels, the TM guesses what prefix w' of the subsequent string is to be consumed by the state trees derived from s, and calls itself recursively with w' and s as arguments. If the recursive call succeeds, it goes on to verify the remainder of w from q'.

Let w' and s be such a prefix and sequence of labels. If s is a single state p, the TM recursively verifies that w' is accepted by M when starting from state p, i.e., $w' \in \mathcal{L}(M_p)$. If s is a pair $p, p' \in Q$, the TM guesses a way to partition w' into subsequences u, u' so that $w' \in u \odot u'$. It then recursively verifies that $u \in \mathcal{L}(M_p)$, and if that is the case, that $u' \in \mathcal{L}(M_{p'})$. Finally, if $s = p^{\odot}$, the TM guesses a non-empty subsequence of w', verifies recursively that this subsequence belongs to $\mathcal{L}(M_p)$, and if so, verifies recursively that the remainder of w' (if it is non-empty) can be accepted from the sequence of labels $p^{\odot}$.

We note that as the TM explores a candidate run top-down, it need only remember a sequence of labels s and a partitioning of the argument string at each level in the call stack. By Lemma 1, the CFSA M can be assumed to be ε-efficient, so by Lemma 2, the height of the call stack can be restricted to $|w| + 1$. The information about where in the string partitions end can easily be maintained in linear space, for example as a bit string where the ith zero signifies the ith symbol in w, and the jth one signifies the end of the partition for the jth object on the current call stack. If follows that the information maintained during any step of the simulated run is linear in $|w|$, so the non-uniform membership problem for CFSA languages can be decided by a linearly bounded nondeterministic TM. As shown in the proof of Theorem 1, no CFSA recognizes the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$, so the CFSA languages form a proper subset of the context-sensitive languages. □

Since not all CFSA-languages are context-free (e.g., there are non-context-free shuffle languages), we conclude that their expressive powers lies strictly between that of context-free grammars and that of context-sensitive grammars.

Also unlike linear bounded Turing machines, CFSA can be efficiently checked for emptiness.

Theorem 4. *The emptiness problem for CFSA is decidable in polynomial time.*

Proof. Let $M = (Q, \Sigma, \delta, I)$ be a CFSA. A state q of M is *live* if $\mathcal{L}(M_q)$ is nonempty. Let $\mathcal{F} \subseteq Q$ be the smallest set satisfying the following conditions.

1. $F_0 = \{q \mid (q, \varepsilon, t_\varepsilon) \in \delta\}$
2. $F_i \subseteq F_{i+1}$
3. if $(q, \alpha, q') \in \delta$ and $q' \in F_i$ then $q \in F_{i+1}$
4. if $(q, \alpha, q'[p^{\odot}])$ and $q' \in F_i$, for any $p \in Q$, then $q \in F_{i+1}$ (recall that the shuffle closure may generate zero instances of p)
5. if $(q, \alpha, q'[s]) \in \delta$ for some $q' \in F_i$ and some s such that every state that appears in s belongs to F_i, then $q \in F_{i+1}$
6. $\mathcal{F} = \cup_{i=0}^{\infty} F_i$

Claim. A state q of M is live if and only if $q \in \mathcal{F}$.

For the if-direction, we prove by induction on the smallest i such that $q \in F_i$ that q is live. For $i = 0$ this is trivially true, since $(q, \varepsilon, t_\varepsilon) \in \delta$, and thus M_q accepts the string ε.

Assume that every state in F_i is live, and consider the state $q \in F_{i+1} \setminus F_i$. If $(q, \alpha, q') \in \delta$, with $q' \in F_i$, then there is a string w such that $M_{q'}$ accepts w. This means that M_q accepts αw and we conclude that q is live. If there is no such rule, there must be a rule $(q, \alpha, q'[s])$ in δ such that q' and either $s = p^\odot$ or every state that appears in s belongs to F_i. If this is the case, then there is a word $w_{q'}$ accepted by $M_{q'}$. If $s = p$, there is a word $w_p \in \mathcal{L}(M_p)$ and conclude that M_q accepts $\alpha \cdot w_p \cdot w_{q'}$. Similarly, if $s = p, p'$ there are strings $w_p \in \mathcal{L}(M_p)$, $w_{p'} \in \mathcal{L}(M_{p'})$, and $w_{p \odot p'} \in w_p \odot w_{p'}$ such that M_q accepts $\alpha \cdot w_{p_1 \odot p_2} \cdot w_{q'}$. Finally, if $s = p^\odot$, we know that M_q accepts $\alpha \cdot w_{q'}$. Thus q is live.

For the other direction, assume that q is live as witnessed by some word $w = \alpha_1 \cdots \alpha_m$ in $\mathcal{L}(M_q)$ with $\alpha_i \in \Sigma \cup \{\varepsilon\}$. Let

$$(w, q) = (w_1, t_1) \to \cdots \to (w_m, t_m) = (\varepsilon, t_\varepsilon)$$

be an accepting sequence of transition steps of M_q on w. We show by induction that every state that appears in $t_1, \ldots, t_m$ is in $\mathcal{F}$. In particular, this means that q belongs to $\mathcal{F}$, because $t_1 = q$. Since $t_m = t_\varepsilon$, all states in t_m belong to $\mathcal{F}$. Assume that all states appearing in t_i belong to $\mathcal{F}$ and consider t_{i-1}. One of the following cases apply (for some leaf node v).

1. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow q']\!]$, and there is a transition $(q, \alpha_i, q') \in \delta$. If this is the case, $q \in \mathcal{F}$ and thus all states of t_{i-1} belong to $\mathcal{F}$.

2. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow q'[u_1, \ldots, u_n]]\!]$, and there is a transition $(q, \alpha_i, q'[s]) \in \delta$ such that

   - $s = p$, $n = 1$, and $u_1 = p$,
   - $s = p_1, p_2$, $n = 2$, $u_1 = p_1$ and $u_2 = p_2$, or
   - $s = p^\odot$ and $u_1 = \cdots = u_n = p$.

   In either case, $q \in \mathcal{F}$ and thus all states of t_{i-1} belong to $\mathcal{F}$.

3. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow t_\varepsilon]\!]$. In this case, q belongs to F_0 and we can conclude that all states appearing in t_{i-1} belong to $\mathcal{F}$.

The set $\mathcal{F}$ can be computed in polynomial time and $\mathcal{L}(M)$ is empty if and only if $\mathcal{F} \cap I = \emptyset$. Thus emptiness for CFSA can be decided in polynomial time. □

4. Membership Problems

4.1. The membership problem for unrestricted CFSA

The membership problem for unrestricted CFSA is intractable, both in the uniform and the non-uniform case.

Theorem 5. *Both the uniform and the non-uniform membership problem for CFSA is NP-complete.*

PROOF. NP-hardness for the uniform membership problem for shuffle expressions is already known; see, e.g., [44, 2, 35]. The non-uniform membership problem is also NP-complete, this follows from both [40] and Corollary 4 in this paper. Corollary 4 states that the non-uniform membership problem for the shuffle of two deterministic linear context-free languages is NP-hard. Theorem 2 says that CFSA can represent all context-free languages and Theorem 1 that they are closed under the shuffle operation, so a CFSA can be constructed to represent the shuffle of context-free languages, establishing NP-hardness.

Demonstrating that the membership problem for CFSA is *in* NP is deferred to Lemma 3 below. □

Lemma 3. *Given a CFSA $M = (Q, \Sigma, \delta, I)$ and a string $w \in \Sigma^*$ it is possible to determine if $w \in \mathcal{L}(M)$ in nondeterministic polynomial time.*

Proof sketch. Due to Lemma 1, we may assume that M is ε-efficient. We show that there is a polynomial P such that for every $w \in \mathcal{L}(M)$, there is a state $q_0 \in Q$ and a sequence of transition steps

$$(w, q_0) = (w_1, t_1) \to \cdots \to (w_n, t_n) = (\varepsilon, t_\varepsilon)$$

such that $n \leq P(|Q|+|w|)$. This result allows an accepting sequence of transition steps to be "guessed" as part of a nondeterministic polynomial-time decision algorithm for the membership problem.

It follows from Lemma 2 that the size of the configuration trees necessary to accept an input string w is bounded by $|w|^2|Q|$, and any sequence of transitions on polynomially sized trees can be limited to a polynomial number of steps. There is thus, for every $w \in \mathcal{L}(M)$, a sequence of polynomial length, which means that a nondeterministic algorithm can check membership by guessing the sequence. □

4.2. The membership problem for acyclic CFSA

We now turn to the membership problem for acyclic CFSA, i.e., the restriction of CFSA that recognizes the shuffle languages.

Corollary 1. *For acyclic CFSA*

1. *the non-uniform membership problem is solvable in polynomial time, and*
2. *the uniform membership problem is NP-complete.*

PROOF. The result for non-uniform membership follows directly from Theorem 2 and the fact, proved in [31], that non-uniform membership for shuffle expressions is polynomial. For the uniform membership problem, membership in NP follows from Theorem 5. NP-hardness follows by an easy adaptation of a result by Barton on the complexity of ID/LP parsing [2]. □

The uniform membership problem is NP-complete already for acyclic and finitely branching CFSA, which only recognize regular languages. This is not too surprising since the similar NFA(&) employed by Gelade et al. [20], which also recognize the regular languages, has PSPACE-complete uniform membership. For some languages, CFSA offer a more succinct form of representation than NFA and the shuffle automata from [31]. One example is the language family $\{\{a^n\} \mid n \in \mathbb{N}\}$, for which the smallest NFAs and shuffle automata have sizes linear in n, while the smallest CFSAs are logarithmic in n.

Corollary 1 states that the membership problem is polynomial for a fixed automaton but NP-hard if the automaton is considered as part of the input. The question then remains whether the size of the automaton merely influences the coefficients of the polynomial or if it affects the degree itself. We give a partial answer by showing that when parameterized by the maximal size of a configuration tree for the automaton, the uniform membership problem for acyclic and finitely branching CFSAs is *not fixed-parameter tractable*, unless FPT = W[1]. This class equivalence is considered very unlikely and would have far-reaching complexity-theoretic implications. For more on parameterized complexity theory, see, e.g., [15, 17].

We state the result for acyclic and finitely branching CFSA, but it could be equivalently stated for closure-free shuffle expressions. We first define the parameterized version of the problem.

Definition 4. An instance of the parameterized uniform membership problem for acyclic and finitely branching CFSA is a pair (M, w) where M is an acyclic and finitely branching CFSA over a finite alphabet Σ and w is a string in Σ^*. The parameter is the maximal size of any configuration tree for M. The question is whether $w \in \mathcal{L}(M)$. □

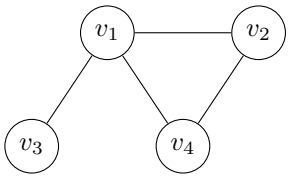
For acyclic and finitely branching CFSA, the maximal size of the configuration trees depends only on the automaton. If the membership problem for these automata was fixed-parameter tractable, it would have an algorithm with running time $f(k) \cdot n^c$, where f is a computable function, k is the parameter (the maximal tree size), n is the instance size, and c is a constant. Theorem 6 gives strong evidence to the contrary.

Theorem 6. *The parameterized uniform membership problem for acyclic and finitely branching CFSA is W[1]-hard.*

The proof is by a fixed-parameter reduction from parameterized clique, which is known to be $W[1]$-complete [15].

Definition 5. An instance of k-Clique is a pair (G, k), where $G = (V, E)$ is an undirected graph and k is an integer. The question is whether there is a set $C \subseteq V$ of size k such that the subgraph of G induced by C is complete. The parameter is k. □

Proof. The proof consists in a reduction from k-Clique to the membership problem at hand. Let $(G = (V, E), k)$ be an instance of k-Clique, and



$$w = v_1v_1v_1v_2v_2v_2v_3v_3v_3v_4v_4v_4e_{1,2}e_{1,3}e_{1,4}e_{2,4}$$
$$s = (v_1v_1v_1 + v_2v_2v_2 + v_3v_3v_3 + v_4v_4v_4)$$
$$t = \Sigma^*$$
$$u = v_1v_2e_{1,2} + v_1v_3e_{1,3} + v_1v_4e_{1,4} + v_2v_4e_{2,4}$$

Figure 2: A graph together with the corresponding input word w and the regular expressions s, t, and u, given $k = 3$.

let $n = |V|$ and $m = |E|$. We construct an alphabet Σ, a shuffle expression r, and a string $w \in \Sigma^*$ such that $|\Sigma| = O(n + m)$, $|r| = O(k \cdot n^2 + k^2 \cdot m)$, $|w| = O(k \cdot n + m)$, the shuffle operator appears $O(k^2)$ times in r, and $w \in \mathcal{L}(r)$ if and only if G has a clique of size k. To construct Σ, we assume that the vertices in V are named $v_1, v_2, \ldots, v_n$ and that the edges are named $e_{i,j}$ where $i < j$ are the numbers of the two incident vertices and let $\Sigma = V \cup E$. The word w is $v_1^k \cdot v_2^k \cdots v_n^k \cdot \mathit{edges}$, where *edges* is any enumeration of the edges in E.

We define the regular languages s, t, u by

- $s = (v_1^k + v_2^k + \cdots + v_n^k)^{n-k}$,
- $t = \Sigma^*$, and
- $u = \Sigma_{e_{i,j} \in E}(v_i \cdot v_j \cdot e_{i,j})$.

Finally, we define

$$r = s \odot t \odot \Big(\bigodot_{i=1}^{k(k-1)/2} u \Big) \ .$$

A graph, together with the expressions and the input string resulting from the reduction with $k = 3$ is shown in Figure 2. The intuition behind the reduction is as follows:

- The expression s matches $n - k$ sequences of k copies of a vertex name. This leaves only k such sequences in w for the rest of r to match against, so the remainder of the expression can only use k distinct vertex names. In the example shown in Figure 2, we have $n = 4$ and $k = 3$. Thus s matches exactly one group of three identical vertex names.
- Each instance of expression u matches one sequence $v_i \cdot v_j \cdot e_{i,j}$. Thus, the $k(k-1)/2$ instances of u match against $k(k-1)$ vertex names and $k(k-1)/2$ edge names. Due to the matching of s, the $k(k-1)$ vertex names can only

be chosen from among k vertices. Thus the $k(k-1)/2$ edge names, which are distinct since *edges* is an enumeration of E, represent edges that have both their endpoints in a set of vertices of size k. In the example from Figure 2, we have $k = 3$ and thus $3(3-1)/2 = 3$ copies of u are used. Since s matches one group of vertex names, these three copies of u can only be matched against a total of three distinct vertex names.

- The expression t matches all remaining vertex and edge names.
- Any graph that has $k(k-1)/2$ distinct edges whose endpoints are all in a set of vertices of size k has a clique of size k.

Thus w belongs to $\mathcal{L}(r)$ if and only if G has a clique of size k. Notice that $|r|$ is polynomial in $|G|$ and that the number of shuffle operators depends only on k.

Using Theorem 2 it is easy to find an acyclic and finitely branching CFSA M_r such that $\mathcal{L}(M_r) = \mathcal{L}(r)$, the size of M_r is polynomial in the size of G, and the maximum size of a configuration tree for M_r is $O(k^2)$. Thus there is a fixed-parameter reduction from k-Clique to parameterized membership for acyclic and finitely branching CFSA, so the latter problem is W[1]-hard. □

The following corollary is immediate.

Corollary 2. *The uniform membership problem for closure-free shuffle expressions, parameterized by the number of shuffle operators, is W[1]-hard.*

4.3. The membership problem for Reg $\odot$ CF *and* Sh $\odot$ CF

We next show that the shuffle of a context-free language and a regular language is efficiently recognizable, even if the language descriptions are considered to be part of the input.

Theorem 7. *The uniform membership problem for the shuffle of two languages, one represented by context-free grammar and one represented by a nondeterministic finite automaton, is solvable in polynomial time.*

Proof (Sketch). It is well known that the shuffle of a regular and a context-free language is context-free. It remains to argue that a grammar for the language can be constructed in polynomial time. To achieve this, it is enough to construct a nonterminal (A, q_1, q_2) for every nonterminal A of the input grammar and every pair (q_1, q_2) of states of the input automaton. Working bottom up, it is straightforward to construct the rules of the grammar in such a way that a string w can be produced from (A, q_1, q_2) if and only if there are w_1 and w_2 such that $w = w_1 \odot w_2$, where w_1 can be produced from A in the input grammar and w_2 can take the input automaton from q_1 to q_2. □

Since acyclic and finitely branching CFSA only provide a more compact representation of the regular languages, Theorem 7 extends to the non-uniform membership problem for the shuffle of a context-free language and a closure-free shuffle language:

Corollary 3. *The non-uniform membership problem for the shuffle of two languages, one represented by a context-free grammar and one represented by an acyclic and finitely branching CFSA, is solvable in polynomial time.*

Extending Theorem 7 with techniques inspired by [31], we get the following:

Theorem 8. *The non-uniform membership problem for the shuffle of a shuffle language and a context-free language is solvable in polynomial time.*

Since the languages are not part of the input, we may assume that they are represented by an acyclic CFSA M, and a context-free grammar G, respectively. We prove the above theorem in several steps. First, we show that we can assume that the CFSA for a shuffle language has certain structural properties. Second, we define *simple* configuration trees, and show that any computation of a CFSA for a shuffle language that has the above-mentioned structural properties can be assumed to use only simple configuration trees. Third, we show an upper bound on the number of different simple configuration trees that need to be taken into account during a computation, and provide a compact representation for these. Finally, we prove the theorem, using an extension of the CYK algorithm.

The first structural property of CFSAs that we consider is *stratification*.

Definition 6. An acyclic CFSA $M = (Q, \Sigma, \delta, I)$ is *stratified* if, for every $q \in Q$, there is at most one $p \in Q$ such that, in a configuration tree, a node with label p can be the parent of a node with label q. □

To proceed, we need a canonical translation from shuffle expressions to CFSAs:

Definition 7. Let s be a shuffle expression. Then M_s is the CFSA constructed from s as in the proof of Theorem 1. We call M_s the *canonical* CFSA for s. □

Observation 1. Let s be a shuffle expression, and let $M_s = (Q, \Sigma, \delta, q_0)$ be the canonical CFSA for s. Then M_s has the following properties.

- It is *stratified*.
- It is *acyclic*.
- For each $q \in Q$, there is at most one vertical transition (p, α, t) in δ with q labelling the root of t. We say that an automaton A with this property is *vertically separated*. We write $scp(A)$ (for shuffle-closure-parent) for the set of states that can have an unbounded number of children in configuration trees, i.e., $scp(M_s) = \{q \mid \exists p, p', \alpha : (p', \alpha, q[p^{\odot}]) \in \delta\}$. □

Having covered the first step of our proof outline, we continue to introduce and reason about so-called simple configuration trees. For this purpose, we introduce the notions of pruned configuration trees and symmetrically equivalent nodes. Prunings delete subtrees produced through shuffle-closure; a pair of nodes in a configuration tree t are symmetrically equivalent if they are identical modulo an automorphism in a pruned version of t, i.e., when we disregard their exact number of descendant subtrees created through shuffle-closure.

Definition 8 (Pruning). Let M_s be the canonical CFSA for a shuffle expression s, let t be a configuration tree of M_s and let v, v' be nodes in t.

We denote by $P(v, v')$ the set of the closest shuffle-closure-parent descendants of v and v'. More formally, let $P(v, v')$ be the set of nodes u of t such that:

1. $t(u) \in scp(M_s)$,
2. u is a descendant of v or v', and
3. there is no node with a label in $scp(M_s)$ on the path from v (or v') to u.

The *pruning* of t with respect to v, v', written $prune(t, v, v')$, is obtained from t by removing all subtrees rooted at children of nodes in $P(v, v')$. □

Definition 9 (Symmetrical equivalence). Let M_s be the canonical CFSA for a shuffle expression s, let t be a configuration tree of M_s and let v, v' be nodes in t. The nodes v and v' are *symmetrically equivalent* if there is an automorphism f on the nodes of $t' = prune(t, v, v')$ such that

- $f(v) = v'$ and $f(v') = v$,
- for every $u \in nodes(t')$, $t'(f(u)) = t'(u)$, and
- for every $u, u' \in nodes(t')$, it holds that $f(u)$ is a child of $f(u')$ if and only if u is a child of u'.

We write $se(v, v')$ if v and v' are symmetrically equivalent.

It is easy to check that symmetrical equivalence is an equivalence relation in the algebraic sense, and thus reflexive, symmetric and transitive. When considering a configuration tree from a computational point of view, we notice that the ordering of its nodes is not important, only its hierarchical structure. It is therefore meaningless to distinguish between symmetrically equivalent nodes in the rewriting process. For our purposes this is an advantage, because we only have to remember to what class of symmetrically equivalent nodes a subtree attaches, not the exact location.

Observation 2. Let $k \in \mathbb{N}$, let t and $s_1, \ldots, s_k$ be configuration trees, let $v_1, \ldots, v_k$ be symmetrically equivalent nodes in $nodes(t)$, and let

$$T = \{t[\![v_1 \leftarrow s_{\phi(1)}, \ldots, v_k \leftarrow s_{\phi(k)}]\!] \mid \phi \text{ is a permutation on } [k]\} \ .$$

For every $t_1, t_2 \in T$ and $w \in \Sigma^*$, if $(t_1, w) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ then $(t_2, w) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ □

Due to Observation 2, it is never useful to apply a transition $r = (p', \alpha, q[p^{\odot}])$ below a node v, when there is a symmetrically equivalent node v' below on which r has already been applied. This claim, which will be proved later on, means that the search space can be reduced to so-called *simple* configuration trees.

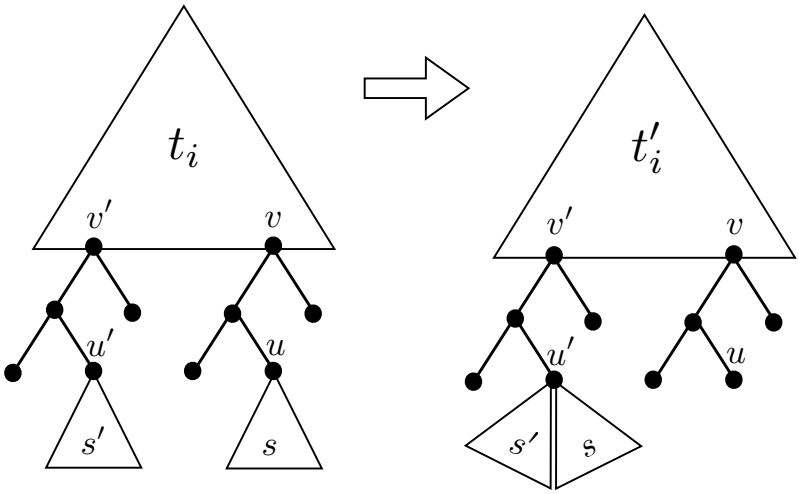


Figure 3: Children obtained through shuffle-closure can be moved between descendants of symmetrically equivalent nodes.

Definition 10. A configuration tree t is *simple* if is it does not contain symmetrically equivalent nodes v and v', such that both v and v' have descendants which are labeled by states in $scp(M_s)$ and have children.

A run of a CFSA is *simple* if all configuration trees of the run are simple. □

Lemma 4. *Let M_s the be the canonical CFSA for a shuffle expression s and let w be a word. Then M_s has a simple accepting run on w, if and only if M_s has an accepting run on w.*

Proof sketch. For the "only if" direction we note that every simple accepting run is an accepting run.

For the opposite direction, we provide a rewrite procedure that rearranges the configuration trees in an accepting run into an alternative run that is also accepting. After applying this procedure a finite number of times we are guaranteed to reach a run that is both accepting and simple.

Assume that M_s has an accepting run $\rho = t_0, t_1, \ldots, t_n$ on w, and that ρ is not simple. Let t_i be the first non-simple tree. Then the transition from t_{i-1} to t_i must have been a vertical transition of the form $(p', \varepsilon, q[p^{\odot}])$ that changed the label of some leaf node u from p' to q and gave it a number of children with label p, say m children. Also, there must be an ancestor v of u (possibly, $v = u$) and a node v' such that v and v' are symmetrically equivalent in t_i. Let ϕ be the corresponding automorphism on $prune(t_i, v, v')$. Let $u' = \phi(u)$. If all the children of u were instead children of u', the tree t_i would be a simple configuration tree. And, indeed, because of the vertical separation of M_s, the

transition that labeled u' by q must have been $(p', \varepsilon, q[p^{\odot}])$. Thus, it could as well have created m extra children of u' with label p, in addition to the children it originally created. This would not have affected any transitions up to configuration tree t_{i-1}. Symmetrically, the transition from t_{i-1} to t_i might not have created any children at all under u. Thus, with the same sequence of transitions, we could equally well have ended up with the configuration tree t'_i which is identical to t_i except that u has no children in t'_i and u' has m more p-labeled children than in t_i. Figure 3 depicts the situation.

It remains to argue that any sequence of transitions used in ρ from t_i forward is also possible from t'_i. Let $j > i$ be the smallest number such that in t_j, either v has no children or v' has no children. We show that the partial run $\rho_{i,j} = t_i, \ldots, t_j$ can be mirrored in a partial run $\rho'_{i,j} = t'_i, \ldots, t'_j$, using the same transitions. If a transition of $\rho_{i,j}$ affects a node in t_k that does not belong to the subtree of v or v', we mirror it directly on t'_k. Now consider a transition from t_k to t_{k+1} that affects a node in a subtree of v or v'. If the same operation is possible on t'_k, we perform it. If not, this can only have two causes.

1. The affected node in t_k is a descendant of u that does not exist in t'_k. In this case, we perform the operation on the corresponding descendant of u'.

2. The affected node in t_k is u' which in t'_k still has children. In this case, we perform the operation on u.

In each of t_i and t_j, we have that exactly one of v and v' is childless. If this is the same node in both trees, they are identical and we are done. If not, we still have to argue that the transitions from t_j forward can be mirrored from t'_j. If not, we use the fact that in t_i and t'_i, v and v' were symmetrically equivalent. Thus we are free to use the automorphism ϕ to reinterpret the sequence $t'_i, \ldots, t'_j$. Under this reinterpretation, t_j and t'_j are identical.

After performing the above operation, all configuration trees up to and *including* t_i are simple. This means that after going through the procedure at most a linear number of times, all configuration trees will be simple. □

Lemma 4 concludes the second step in our proof outline. What remains is to provide a compact representation for simple configuration trees. This makes it necessary to compress the potentially large number of subtrees produced through shuffle closures. Under nodes labelled by states in $scp(M_s)$ we therefore only record which types of subtrees appear, and annotate each of them with a "repetition counter", which encodes the number of times they appear.

Definition 11. Let $M_s = (Q, \Sigma, \delta, q_0)$ be the canonical CFSA for a shuffle expression and let t be a simple configuration tree of M_s. The *compact configuration tree* $\mathrm{cct}(t)$ for t is a tree with nodes labelled by $Q \times \mathbb{N}^*$ where the second component is used as a sequence of counters, one for each direct subtree of the node in question. We define $\mathrm{cct}(t)$ by induction on the structure of t as follows.

- If $t = q$, then $\mathrm{cct}(t) = (q, \langle\rangle)$.

- If $t = q[t_1, \ldots, t_k]$ and $q \in Q \setminus scp(M_s)$, then

$$\mathrm{cct}(t) = (q, \langle \underbrace{1, \ldots, 1}_{k} \rangle)[\mathrm{cct}(t_1), \ldots, \mathrm{cct}(t_k)] \ .$$

  Notice that in this case, k always equals 1 or 2.

- If $t = q[t_1, \ldots, t_k]$ and $q \in scp(M_s)$, then

$$\mathrm{cct}(t) = (q, \langle n_1, \ldots, n_m \rangle)[\mathrm{cct}(t'_1), \ldots, \mathrm{cct}(t'_m)] \ ,$$

  where

  1. $t'_1, \ldots, t'_m$ is an enumeration of the elements in $\{t_1, \ldots, t_k\}$, so t'_i is not isomorphic to t'_j for any $i, j \in [m]$, making m the number of unique trees, up to isomorphism, in $t_1, \ldots, t_k$,
  2. $n_i = |\{j \mid j \in [k], t_j \text{ isomorphic to } t'_i\}|$ for all i.

We write $\mathrm{CCT}(M_s)$ for the set of all compact configuration trees of M_s. □

It should be clear that there is a many-to-one correspondence between simple configuration trees t and their respective compact configuration trees $\mathrm{cct}(t)$.

Next, we show that the size of compact representation trees for simple configuration trees depends only on the automaton, not on the input word.

Lemma 5. *Let M_s be the canonical CFSA for a shuffle expression s. Then there is a constant $c \in \mathbb{N}$ that depends only on M_s, such that for every simple configuration tree t of M_s, the size of* $\mathrm{cct}(t)$ *is at most c.*

Proof. Let t be a simple configuration tree of M_s. Since M_s is acyclic we know that *height*(t), and thus also *height*$(\mathrm{cct}(t))$, is at most $|Q|$. We argue that the index (i.e., the number of equivalence classes) of the relation *se* on t is completely decided by M_s.

Let *SCFree* be the set of subtrees t' of simple configuration trees of M_s such that in t' no $scp(M_s)$-labeled node has children. We note that since the height of trees in *SCFree* is bounded by $|Q|$ and since they branch only binarily, we know that $|SCFree|$ is finite and depends only on M_s.

Let *Layer*(i, t) be the tree obtained from t by removing all nodes v such that there are i or more $scp(M_s)$-labeled nodes on the path from the root to v (not including v itself). We argue by induction on i, that the index of *se* on *Layer*(i, t) depends only on i and on M_s. Since i is itself bounded by $|Q|$ this will in the end give us what we need.

In the base case, where $i = 1$, the claim holds, since *Layer*$(1, t) \in$ *SCFree* and thus *Layer*$(1, t)$ has a maximum number of nodes that depends only on M_s. The index of *se* can of course not exceed the number of nodes.

For the inductive case, we assume that there is a number e_i that depends only on i and on M_s, such that for all simple configuration trees t of M_s, the index of *se* on *Layer*(i, t) is at most e_i. We obtain *Layer*$(i+1, t)$ from *Layer*(i, t)

by adding trees from *SCFree* as children to $scp(M_s)$-labeled leaves of $Layer(i,t)$. For two nodes v_1 and v_2 in $nodes(Layer(i+1,t)) \setminus nodes(Layer(i,t))$ not to be symmetrically equivalent, they must either belong to two such subtrees from *SCFree* that are not isomorphic or their closest ancestors in $Layer(i,t)$ belong to different equivalence classes of se. This means that in $Layer(i+1,t)$ there can be no more than $e_i \cdot |SCFree| \cdot m$ equivalence classes of se, where m is the maximum size of any tree in *SCFree*. Using the induction hypothesis, this quantity depends only on i and M_s.

Since t is a simple configuration tree, in any set of symmetrically equivalent nodes, there is at most one whose corresponding subtree contains an $scp(M_s)$-labeled node that has children. Take a set $\{v_1, \ldots, v_n\}$ of symmetrically equivalent nodes (n can be arbitrarily large). Then, $\{t/v_1, \ldots, t/v_n\}$ contains at most two unique trees, the single one with $scp(M_s)$-labeled nodes with children being one, while all other subtrees are necessarily isomorphic. This immediately implies that the number of unique, up to isomorphism, subtrees of t depends only on M_s.

All that remains is to note that in $\mathrm{cct}(t)$, every node either has at most two children (non-*scp* nodes) or it has only unique, up to isomorphisms, children. Since the number of unique subtrees depends only on M_s, and $height(t) \leq |Q|$ this means that the number of nodes of $\mathrm{cct}(t)$ depends only on M_s. □

Lemma 6. *Let $M_s = (Q, \Sigma, \delta, I)$ be the canonical CFSA for a shuffle expression. Then there exists a constant $k \in \mathbb{N}$ that depends only on M_s, such that the number of distinct compact configuration trees needed by M_s for accepting all words in $\mathcal{L}(M_s)$ of length at most n is bounded by $O(n^k)$.*

Proof. We may assume, thanks to Lemma 1, that M_s is ε-efficient. This means that no intermediate configuration tree in a run over a word of length n needs to contain more than $n+1$ leaf nodes. Indeed, whenever a configuration tree contains $n+1$ leaf nodes, by the pigeon hole principle, at least one of the states must ultimately derive ε, since there are only n symbols in the string. As such, whenever a configuration contains $n+1$ leafs we can safely nondeterministically choose a leaf state which can derive ε and replace it by t_ε in the next step, creating a new run. Iterating this process produces a run in which no configuration tree has more than $n+1$ leaf nodes.

Since an acyclic CFSA will have configuration trees of height at most $|Q|$, no configuration tree needs to be of size greater than $(n+1)|Q|$.

Lemma 5 establishes that there is a constant c such that no compact configuration tree corresponding to a simple configuration tree of M_s has more than c nodes. This also means that they contain at most c repetition counters (the counters that are placed as part of the children in *scp*-nodes in the $\mathrm{CCT}(M_s)$ construction). We have also shown that no intermediary configuration tree of M_s running on a word of length n needs to have more than $(n+1)|Q|$ nodes.

To conclude, we note that during any step of a simple run of M_s on a word of length n, there are less than $(|Q|+1)^c$ possible compact configuration trees when ignoring the values of the repetition counters. Furthermore, there are less

than $(n+1)|Q|$ "units" to be divided among the c counters, which can be done in less than $((n+1)|Q|)^c$ ways. Therefore, there are less than $(|Q|+1)^c((n+1)|Q|)^c$ possible compact configuration trees for any step of M_s. Since c depends only on M_s, we have the desired bound of $O(n^k)$ with k depending only on M_s. □

Finally, we are ready to prove Theorem 8.

Proof (of Theorem 8). To compute membership for the shuffle of a shuffle language and a context-free language, we outline an extension of the CYK algorithm. The extension maintains triples consisting of a nonterminal from the context-free grammar G and two configuration trees with respect to the CFSA M_s. A triple (A, t, t') is assigned to a substring w' of the input string w if

1. $w' \in w'_1 \odot w'_2$,
2. the string w'_1 can take M from t to t', and
3. the string w'_2 can be derived from A in the grammar G.

A pair of triples (A, t, t') and (B, t', t'') for the substrings w' and w'' can be combined into a triple (C, t, t'') for the substring $w'w''$ if there is a derivation rule $C \to AB$ in G. To decide whether there is a parse for w, one starts by deriving all possible triples for every substring of w of length 1, and then uses the above combination rule to dynamically complete the parse chart.

A string of length n has $O(n^2)$ substrings, which means that $O(n^2)$ sets of triples have to be computed. From Lemma 6 we know that there is a $k \in \mathbb{N}$, that depends only on the shuffle language involved, such that no more than $O(n^k)$ distinct configuration trees have to be considered. If G has m nonterminals, there are thus no more than $O(m \cdot n^k)$ possible triples. Given that we have the sets of triples for all substrings of w, deciding whether a particular triple belongs to the set of triples for w can be done in polynomial time. Since m and k are constants, the problem is polynomial in the length n of the string. □

5. An NP-complete Shuffle of Two Deterministic Linear Context-Free Languages

In this section we will construct two deterministic linear context-free languages such that deciding the membership problem for their shuffle is NP-complete. Phrased differently we will demonstrate the non-uniform membership problem for DLCF ⊙ DLCF is NP-complete.

5.1. Proof Preliminaries

Deterministic linear context-free languages, denoted DLCF, will be used extensively in the following. It is assumed that the reader is familiar with the relevant formalisms for these languages, for instance, deterministic pushdown

automata restricted to a single pushdown reversal. For an introduction to the subject, see, e.g., the textbook by Hopcroft and Ullman [30]. We typically give inductive definitions for the DLCF languages, from which push-down automata can be easily deduced.

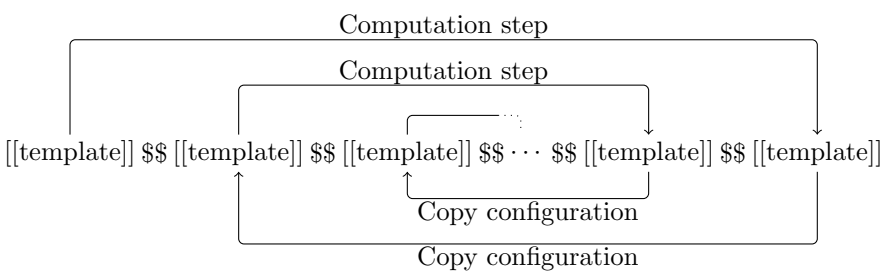
Non-deterministic polynomial time-bounded Turing machines are used heavily in the proofs to demonstrate NP-completeness. Full definitions of the machines are given, but for more complete background information on these topics see, for example, [18, 36].

5.2. Proof Overview

To make two DLCF languages perform a computation, they have to be made interdependent. This is done by constructing a template input string containing sequences of double-bracketed bits:

$$w = [[01]][[01]][[01]]\$\$[[01]][[01]][[01]]\$\$[[01]][[01]][[01]].$$

Assume that the *first* language contributes the string [0][1][0]$[1][1][1]$[1][0][1] to w, then the *second* language *has to* contribute the string [1][0][1]$[0][0][0]$[0][1][0] if the whole input string w is to be assembled. Notice that the bit sequence in this string is the complement of the bit sequence in the first. In this way the two shuffled languages can communicate arbitrary choices by only accepting properly bracketed input. The proof will use this to let one language make computation steps for a Turing machine, while the other language copies the configuration to link the computation steps. The following figure acts as a visual aid to see how the languages will cooperate to simulate the computation. Many details are left out; the figure only serves as a structural overview.



5.3. Parsing the Shuffle of Deterministic Linear Context-Free Languages

The reduction hinges on representing the computations of Turing machines as strings. To simplify the presentation, we give custom definitions of non-deterministic Turing machine configurations and runs, and work with totally ordered state spaces.

Definition 12. Let S be an *ordered set*, i.e., a set S together with a total order on S. For $i \in [|S|]$, let $S(i)$ denote the ith element in S according to this total order. When a set is given in the form of an enumeration $S = \{s_1, \ldots, s_n\}$, it is implied that $s_i = S(i)$.

Definition 13. A *non-deterministic Turing machine (NTM)* is a tuple (Q, Δ) where

- Q is a finite ordered set of states,
- $\Delta \subset Q \times \{0,1\} \times \{\leftarrow, \rightarrow\} \times Q \times \{0,1\}$ is a finite set of rules,
- $Q(1)$ is the initial state, and $Q(|Q|)$ is the accepting state.

The DLCF languages that we shall consider are made up of symbols from the alphabet

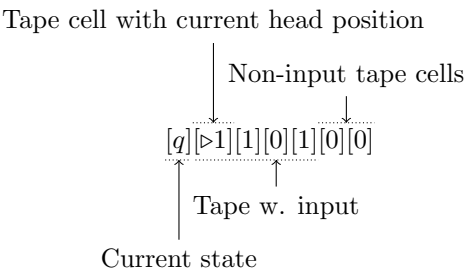
$$\Sigma_M = Q \cup \{0, 1, \triangleright, [,], \$, \#\} \ .$$

A configuration becomes a simple string containing both the state, tape contents, and tape position, allowing rule applications to be expressed as string rewrites.

Definition 14. The set of *configurations* of an NTM $M = (Q, \Delta)$, denoted C_M, is the set

$$C_M = \{[\} \cdot Q \cdot \{]\} \cdot \{[0],[1]\}^* \cdot \{[\triangleright 0],[\triangleright 1]\} \cdot \{[0],[1]\}^* \subset \Sigma_M^*.$$

Example 2. As will be seen in Definition 16, an NTM will be provided with tape cells it can work on by padding the input with additional cells filled with zeros. For example, an NTM with initial state q, the input string 1101, and 6 tape cells at its disposal would start in the following configuration.



Definition 15. For an NTM $M = (Q, \Delta)$, the rule $r \in \Delta$ is *applicable* to a configuration $c \in C_M$ and yields the configuration $c' \in C_M$ under the following conditions. Let $r = (q, \alpha, d, q', \alpha')$. For all strings t_1 and t_2, and $\beta \in \{0,1\}$,

- if $d = \rightarrow$ and $c = [\cdot q \cdot] \cdot t_1 \cdot [\triangleright \cdot \alpha \cdot][\cdot \beta \cdot] \cdot t_2$ then $c' = [\cdot q' \cdot] \cdot t_1 \cdot [\cdot \alpha' \cdot][\triangleright \cdot \beta \cdot] \cdot t_2$,
- if $d = \leftarrow$ and $c = [\cdot q \cdot] \cdot t_1 \cdot [\cdot \beta \cdot][\triangleright \cdot \alpha \cdot] \cdot t_2$ then $c' = [\cdot q' \cdot] \cdot t_1 \cdot [\triangleright \cdot \beta \cdot][\cdot \alpha' \cdot] \cdot t_2$.

We denote this rule application by $c \xrightarrow{r} c'$, or $c \rightarrow c'$ leaving r implicit.

Example 3. For example, in the configuration [q][0][1][0][▷1][1][0] it is possible to apply the rule $(q, 1, \rightarrow, q', 0)$ to produce the configuration [q′][0][1][0][0][▷1][0]. In the configuration [q][0][1][▷0] a rule $(q, 0, \rightarrow, q', 0)$ cannot be applied since there is no room to move to the right, nor can the rule $(q, 1, \leftarrow, q', 0)$, since ▷ is pointing to a 0 and the rule requires a 1.

Let us now define what it means for an NTM to accept a language in time bounded by some function.

Definition 16. Take an NTM $M = (Q, \Delta)$, a function $\psi : \mathbb{N} \to \mathbb{N}$ and a string $\alpha_1 \cdots \alpha_n \in \{0,1\}^*$. The *initial configuration* is defined as

$$I(M, \psi, \alpha_1 \cdots \alpha_n) = [\,\cdot\, Q(1) \,\cdot\,] \underbrace{[\triangleright \cdot \alpha_1 \cdot] \cdots [\cdot \alpha_n \cdot][0][0] \cdots [0]}_{\psi(n)+1 \text{ bracketed bits}},$$

the set of *final configurations* is $F(M) = ([\,\cdot\, Q(|Q|) \,\cdot\,] \cdot \Sigma_M^*) \cap C_M$.

M accepts $\alpha_1 \cdots \alpha_n$ in ψ-bounded time if and only if the initial configuration can be transformed into some final configuration by *exactly* $\psi(n)$ rule applications. That is, there exists $\psi(n)+1$ configurations, $c_1, \ldots, c_{\psi(n)+1}$ such that $c_1 = I(M, \psi, \alpha_1 \cdots \alpha_n)$, $c_{\psi(n)+1} \in F(M)$ and $c_i \to c_{i+1}$ for all $i \in [\psi(n)]$. The *language M accepts in ψ-bounded time* is exactly the set of strings M accepts in ψ-bounded time.

The above definition is slightly irregular in that M is required to take *exactly* $\psi(n)$ steps to accept a string of length n, but any Turing machine that would accept the string in *at most* $\psi(n)$ steps can remain in the accepting state indefinitely to fulfil this condition. This makes the above definition equivalent to, e.g., that of Minsky [36]. It follows that every problem $L \in \mathrm{NP}$ is, when suitably encoded, accepted by some NTM M in polynomially bounded time [18].

The template string defined next will be used as the input for the membership query, encoding the Turing machine input and a long specially formatted suffix to make the shuffled computation possible.

Definition 17. The *run template string* for running the machine $M = (Q, \Delta)$ in ψ-bounded time on the input string $\alpha_1 \cdots \alpha_n \in \{0,1\}^*$ ($n \in \mathbb{N}$) is denoted $S(M, \psi, \alpha_1 \cdots \alpha_n)$ and is defined as follows. First the *configuration template* is

$$T = [[\,\cdot\, Q(1) \cdots Q(|Q|) \,\cdot\,]] \underbrace{[[\triangleright 01]] \cdots [[\triangleright 01]]}_{\psi(n)+1 \text{ times}}.$$

Then $S(M, \psi, \alpha_1 \cdots \alpha_n)$ equals

$$I(M, \psi, \alpha_1 \cdots \alpha_n) \cdot \underbrace{\$\$ \cdot T \cdot \$\$ \cdot T \cdots \$\$ \cdot T}_{\psi(n) \text{ occurrences of } T} \cdot \$\$\#\# \cdot \underbrace{\$\$ \cdot T^{\mathcal{R}} \cdot \$\$ \cdot T^{\mathcal{R}} \cdot \$\$ \cdots T^{\mathcal{R}}}_{\psi(n)+1 \text{ occurrences of } T^{\mathcal{R}}}.$$

Example 4. Let $M = (\{q_1, q_2\}, \Delta)$, and let $\psi(2) = 1$, then $S(M, \psi, 1)$ is

$$[q_1][\triangleright 1][0]\$\$[[q_1 q_2]][[\triangleright 01]][[\triangleright 01]]\$\$\#\#\$\$]]10\triangleright[[\,]]10\triangleright[[\,]]q_2 q_1[[\$\$]]10\triangleright[[\,]]10\triangleright[[\,]]q_2 q_1[[.$$

The logical "bracketed" units are divided by a dotted line as a visual aid, since the $T^{\mathcal{R}}$ strings are made hard to read by their reversed brackets.

Next we define the shuffle complement with respect to a template.

Definition 18. For all strings $w, t \in \Sigma_M^*$, let $comp(w,t)$ denote the *shuffle complement of w with respect to t*, defined as

$$comp(w,t) = \{x \in \Sigma_M^* \mid t \in w \odot x\}.$$

Example 5. $comp([q_1][0][\triangleright 1], [[q_1 q_2 q_3]][[\triangleright 01]][[\triangleright 01]]) = \{[q_2 q_3][\triangleright 1][0]\}$.

A very small but important lemma follows.

Lemma 7. *For any configuration $c \in C_M$ and configuration template T (as in Definition 17) if it holds that $|c|_[= \frac{1}{2}|T|_[$ then*

1. *$comp(c,T) = \{c'\}$ for some string c', and*
2. *$comp(c',T) = \{c\}$.*

Proof (sketch). If we have a configuration template string T as in Definition 17 and a configuration c, such that $|c|_[= \frac{1}{2}T_[$, then this means that T and c have the *same number of bracketed sections* (T has each section double-bracketed, $[[\triangleright 01]]$, c has each single-bracketed as in $[\triangleright 1]$). As a consequence $comp(c,T) = \{c'\}$ is a singleton. This is easy to see, by observing that the interleaving of c can only ever pick *one* of the [symbols in each [[pair in T, since it needs to read a] symbol before reading another left bracket. This forces it to skip the other bracket in the pair, meaning that the bracketed sections will match up one-to-one in the shuffle.

This in turn enforces that c' will *also* have $|c'|_[= \frac{1}{2}|T|_[$, and will have similarly single-bracketed sections, containing the complement of those in c with respect to the string $\triangleright 01$. The same argument therefore establishes that $comp(c',T) = \{c\}$. □

Next we define a deterministic linear context-free language which will encode the steps a given NTM can make.

Definition 19. For an NTM $M = (Q, \Delta)$ the *step language for M*, denoted $L_{step(M)}$, is the smallest language that contains the string #, and all strings $c_1 \cdot \$ \cdot l \cdot \$ \cdot c_2^{\mathcal{R}}$, where $l \in L_{step(M)}$, $c_1, c_2 \in C_M$, and $c_1 \xrightarrow{r} c_2$ for some $r \in \Delta$.

Example 6. Let $M = (\{q_1, q_2\}, \{(q_1, 0, \rightarrow, q_2, 1)\})$, then for example

$$[q_1][\triangleright 0][1][0]\$\#\$]0[\,]1\triangleright[\,]1[\,]q_2[\in L_{step(M)},$$

$$[q_1][0][\triangleright 0][0]\$\#\$]0\triangleright[\,]1[\,]0[\,]q_2[\in L_{step(M)},$$

$$[q_1][\triangleright 0][1][0]\$[q_1][\triangleright 0][1][0]\$\#\$]0[\,]1\triangleright[\,]1[\,]q_2[\$]0[\,]1\triangleright[\,]1[\,]q_2[\in L_{step(M)}.$$

It might not be immediately obvious that this language is both linear and deterministic, so let us look at how a deterministic linear push-down automaton can accept it. An automaton for $L_{step(M)}$ can start by pushing the first half of the string onto its stack, validating that it is in the regular language $(C_M \cdot \$)^*$

in the process. When it encounters # it switches to popping off the stack, while popping $c_1 \in C_M$ reading the reverse of $c_2 \in C_M$ on the string, and immediately rejecting unless c_2 differs from c_1 by exactly one rule application from Δ. The automaton can achieve this by checking that c_1 and c_2 are equal in all positions except the states and the immediate neighbourhoods of the $\triangleright$ symbol, both of which are constant-sized and can be remembered in the state of the automaton. It then validates that these differences correspond to a rule in Δ.

Now we turn to the other DLCF language, which is responsible for linking the computation steps by making copies of the complement of configurations. It consists of strings of the form $\bar{c}_1 \cdot \$ \cdot \bar{c}_2 \cdots \bar{c}_2^{\mathcal{R}} \cdot \$ \cdot \bar{c}_1^{\mathcal{R}}$ where each $\bar{c}_i$ is such that $\{\bar{c}_i\} = comp(c, T)$ for some configuration c and configuration template T. Compare the constructed strings to those in Example 5.

Definition 20. For an NTM $M = (Q, \Delta)$ the *inverted copy language for* M, denoted $L_{copy(M)}$, is defined as $L_{copy(M)} = \$ \cdot L$ where L is in turn defined as follows. First let

- $\bar{Q}_i = [\,\cdot Q(1) \cdot Q(2) \cdots Q(i-1) \cdot Q(i+1) \cdots Q(|Q|) \cdot\,]$ for $i \in [|Q|]$,
- $U = \{[\triangleright 0], [\triangleright 1], [0], [1]\} \cdot \{[\triangleright 0], [\triangleright 1], [0], [1]\}^*$.

Then the strings in L are exactly the following. First, for all $t \in U$

$$\#\$ \cdot (\bar{Q}_{|Q|} \cdot t)^{\mathcal{R}} \in L.$$

Second, for all $\bar{c} \in \{\bar{Q}_i \mid i \in [|Q|]\} \cdot U$, and $l \in L_{copy(M)}$

$$\bar{c} \cdot \$ \cdot l \cdot \$ \cdot \bar{c}^{\mathcal{R}} \in L_{copy(M)}.$$

Example 7. Let $M = (\{q_1, q_2, q_3\}, \Delta)$, where q_3 is the final (last) state. Then among the strings in $L_{copy(M)}$ are

$$\#\$]0[\,]1\triangleright[\,]0\triangleright[\,]q_2q_1[,$$

$$[q_1q_3][\triangleright 0][\triangleright 1][1]\$\#\$]0[\,]1\triangleright[\,]0\triangleright[\,]q_2q_1[\$]1[\,]1\triangleright[\,]0\triangleright[\,]q_3q_1[,$$

$$[q_2q_3][1][\triangleright 0]\$[q_1q_3][\triangleright 0][\triangleright 1][1]\$\#\$]0[\,]1\triangleright[\,]0\triangleright[\,]q_2q_1[\$]1[\,]1\triangleright[\,]0\triangleright[\,]q_3q_1[\$]0\triangleright[\,]1\triangleright[\,]q_3q_2[.$$

It should be clear that this language is both deterministic and linear, the symbol # marking the centre playing a key role. The argument is similar to the one in the proof of Lemma 7, but slightly simpler, because no rules need to be taken into account.

This only leaves us to assemble the pieces to prove the main result.

Theorem 9. *Take any $w \in \{0,1\}^*$, NTM M and function $\psi : \mathbb{N} \to \mathbb{N}$. Then M accepts w in ψ-bounded time if and only if $S(M, \psi, w) \in L_{step(M)} \odot L_{copy(M)}$.*

The proof of Theorem 9 is divided into Lemma 8 and Lemma 9; the first showing the "only if" direction, the second the "if" direction.

Lemma 8. *Take any string $\alpha_1 \cdots \alpha_n \in \{0,1\}^*$, NTM $M = (Q, \Delta)$ and function $\psi : \mathbb{N} \to \mathbb{N}$. If M accepts the string $\alpha_1 \cdots \alpha_n$ in ψ-bounded time then $S(M, \psi, \alpha_1 \cdots \alpha_n) \in L_{step(M)} \odot L_{copy(M)}$.*

Proof. Let $c_1, \ldots, c_{\psi(n)+1} \in C_M$ be the sequence of configurations which makes M accept $\alpha_1 \cdots \alpha_n$ (so $c_1 = I(M, \psi, \alpha_1 \cdots \alpha_n)$ and $c_{\psi(n)+1} \in F(M)$). Then construct the string

$$w_{step} = c_1 \cdot \$ \cdot c_2 \cdot \$ \cdots \$ \cdot c_{\psi(n)} \cdot \$\#\$ \cdot c^{\mathcal{R}}_{\psi(n)+1} \cdot \$ \cdot c^{\mathcal{R}}_{\psi(n)} \cdot \$ \cdots \$ \cdot c^{\mathcal{R}}_2.$$

Notice that $w_{step} \in L_{step(M)}$ by construction. Now, for each $i \in [\psi(n) + 1]$ let $\{\bar{c}_i\} = comp(c_i, T)$ where T is a configuration template as in Definition 17. Recall that this complement is always a singleton. Now let

$$w_{copy} = \$ \cdot \bar{c}_2 \cdot \$ \cdot \bar{c}_3 \cdot \$ \cdots \bar{c}_{\psi(n)} \cdot \$\#\$ \cdot \bar{c}^{\mathcal{R}}_{\psi(n)+1} \cdot \$ \cdot \bar{c}^{\mathcal{R}}_{\psi(n)} \cdots \$ \cdot \bar{c}^{\mathcal{R}}_2.$$

It is then straightforward to check that $w_{copy} \in L_{copy(M)}$ by construction.

As an abbreviation denote the template string $S(M, \psi, \alpha_1 \cdots \alpha_n)$ by w. All that remains is to show that $w \in w_{step} \odot w_{copy}$. To illustrate:

$$\begin{array}{rl} w = & c_1\$\$T\$\$\cdots\$\ \ T\ \ \$\$\#\#\$\$\ \ T^{\mathcal{R}}\ \ \$\cdots\$T^{\mathcal{R}}, \\ w_{step} = & c_1\ \$\ c_2\ \$\ \cdots\$c_{\psi(n)}\ \ \$\#\$\ \ c^{\mathcal{R}}_{\psi(n)+1}\$\cdots\$\,c^{\mathcal{R}}_2, \\ w_{copy} = & \ \ \$\,\bar{c}_2\,\$\,\cdots\$\bar{c}_{\psi(n)}\ \ \$\#\$\ \ \bar{c}^{\mathcal{R}}_{\psi(n)+1}\$\cdots\$\,\bar{c}^{\mathcal{R}}_2. \end{array}$$

w and w_{step} both start with c_1, so cancel that bit. Next w contains two dollar signs, one corresponds to the initial in w_{copy} and one the next symbol in w_{step}. After that a T configuration template is next in w, c_2 is next in w_{step}, and $\bar{c}_2$ is next in w_{copy}. By construction $T \in c_2 \odot \bar{c}_2$, leaving us again with \$\$ next in w and a single \$ next in the other strings, and so on through all of w. □

Lemma 9. *Take any string $\alpha_1 \cdots \alpha_n \in \{0,1\}^*$, NTM $M = (Q, \Delta)$ and function $\psi : \mathbb{N} \to \mathbb{N}$. If $S(M, \psi, \alpha_1 \cdots \alpha_n) \in L_{step(M)} \odot L_{copy(M)}$ then M accepts $\alpha_1 \cdots \alpha_n$ in ψ-bounded time.*

Proof. Let $w = S(M, \psi, \alpha_1 \cdots \alpha_n)$, and let the strings $w_{step} \in L_{step(M)}$ and $w_{copy} \in L_{copy(M)}$ such that $w \in w_{step} \odot w_{copy}$ (the lemma assumes these exist).

No string in $L_{step(M)} \cup L_{copy(M)}$ has two \$ symbols in a row, while every \$ occurrence in w consists of two \$ symbols. This enforces that every such \$\$ substring in w is divided up so that one belongs to w_{step} and one to w_{copy} (so $|w_{step}|_\$ = |w_{copy}|_\$ = \frac{1}{2}|w|_\$$). Combining this with the way $L_{step(M)}$ and $L_{copy(M)}$ are constructed it follows that the shuffling must have this structure

$$\begin{array}{rl} w = & c_1\$\$T\$\$\cdots\$\ \ T\ \ \$\$\#\#\$\$\ \ T^{\mathcal{R}}\ \ \$\cdots\$T^{\mathcal{R}}, \\ w_{step} = & c_1\ \$\ c_2\ \$\ \cdots\$c_{\psi(n)}\ \ \$\#\$\ \ d^{\mathcal{R}}_{\psi(n)+1}\$\cdots\$\,d^{\mathcal{R}}_2, \\ w_{copy} = & \ \ \$\,e_2\,\$\,\cdots\$e_{\psi(n)}\ \ \$\#\$\ \ e^{\mathcal{R}}_{\psi(n)+1}\$\cdots\$\,e^{\mathcal{R}}_2, \end{array}$$

for some configurations $c_1, \ldots, c_{\psi(n)}, d_2, \ldots, d_{\psi(n)+1} \in C_M$, and some strings $e_2, \ldots, e_{\psi(n)+1}$. That is, the assumption that $w \in w_{step} \odot w_{copy}$ does together

with the placement of \$ symbols imply that

$$T \in c_i \odot e_i \quad \text{for all } i \in \{2, \ldots, \psi(n)\}, \tag{1}$$
$$T \in d_i \odot e_i \quad \text{for all } i \in \{2, \ldots, \psi(n)+1\}. \tag{2}$$

The second is not reversed since $T^{\mathcal{R}} \in d_i^{\mathcal{R}} \odot e_i^{\mathcal{R}} \iff T \in d_i \odot e_i$. Next, recall from Lemma 7 that $comp(c_i, T)$ and $comp(d_i, T)$ are singletons for all $i \in \{2, \ldots, \psi(n)\}$. Equations 1 and 2 dictate that the string $e_i \in comp(c_i, T)$ and the string $e_i \in comp(d_i, T)$, so $comp(c_i, T) = comp(d_i, T) = \{e_i\}$. Reversing this (again by Lemma 7) yields $comp(e_i, T) = \{c_i\} = \{d_i\}$, so $c_i = d_i$. Let (the previously undefined) $c_{\psi(n)+1}$ be equal to $d_{\psi(n)+1}$ as well. The construction of $L_{step(M)}$ and $L_{copy(M)}$ dictates that

- $c_i \to d_{i+1}$, and therefore $c_i \to c_{i+1}$, for all $i \in [\psi(n)]$,
- $c_1 = I(M, \psi, \alpha_1 \cdots \alpha_n)$,
- $comp(e_{\psi(n)+1}, T) = \{c_{\psi(n)+1}\} \subset F(M)$ (since $e_{\psi(n)+1}$ does *not* contain the final state by construction).

From this it follows that $c_1, \ldots, c_{\psi(n)+1}$ is a correct configuration sequence which makes M accept $\alpha_1 \cdots \alpha_n$. □

It follows from Theorem 9 that the non-uniform membership problem for the shuffle of DLCF languages is NP-complete.

Corollary 4. *For an input string w it is an NP-complete problem to decide whether or not $w \in L \odot L'$ when L and L' are deterministic linear context-free languages, even when L and L' are fixed.*

Proof. The problem is trivially *in* NP. Membership in context-free languages can be decided in polynomial time, and we can, in polynomial time, guess any w_1 and w_2 such that $w = w_1 \odot w_2$ and check if $w_1 \in L$ and $w_2 \in L'$.

Hardness follows easily from Theorem 9. Pick any NTM M and *polynomial* function ψ such that M runs in ψ-bounded time. This characterises NP by definition. Fix the languages $L = L_{step(M)}$ and $L' = L_{copy(M)}$. It is then possible to check if M would accept an input string w in ψ-bounded time by checking if $S(M, \psi, w) \in L \odot L'$. The reduction is polynomial since $S(M, \psi, w)$ produces a string that is of length $\mathcal{O}(\psi(|w|)^2)$ and can, because of its exceedingly simple structure, be constructed in time $\mathcal{O}(\psi(|w|)^2)$. Thus, choosing M such that it accepts an NP-complete language in polynomial time (e.g. a universal NTM) concludes the proof. □

Corollary 5. *Corollary 4 holds even for languages over an alphabet of size 3.*

Proof sketch. We use the alphabet $\{0, [,]\}$ and use the 0 symbol to encode all other symbols in unary. To start with, let 1 be written as 00 and ▷ as 0000 (changing the template bit [[▷01]] into [[0000000]]). For example, if one automaton reads [00000], corresponding to [▷0] (or [0▷], but the order is irrelevant), then the remaining subsequence is [00], correctly corresponding to [1].

Now $$ can be represented as $[[0^{16}]]$ (i.e. 16 zeroes with brackets around), with each language reading $ as $[0^8]$. Similarly ## can be represented as $[[0^{32}]]$, each language reading # as $[0^{16}]$. Notice that these are sufficiently long that they cannot be divided into subsequences that allow them to be confused with any other case.

Finally, the state vector part of the template, $[[q_1 q_2 \cdots]]$, can be replaced by $|Q|$ "bits", $[[000]][[000]] \cdots [[000]]$. The step language reads [00] (representing "1") in the ith position if q_i is the current state, and [0] in all other positions, and the copy language copies the remainder as usual. □

6. Conclusions and Future Work

Concurrent finite-state automata combine the expressive power of context-free and shuffle languages. The CFSA languages are properly included in the context-sensitive languages, and minor restrictions of the device suffice to obtain the regular, context-free, and shuffle languages. CFSA have comparatively nice closure properties, and can be checked for emptiness in polynomial time.

It remains the case, however, that the computational complexity of the membership problem for a formalism is of critical importance in practical applications. This paper demonstrates the non-uniform membership problem NP-hard for a quite restricted CFSA (of the form DLCF $\odot$ DLCF). This may appear daunting, but many possibilities remain for deriving new language classes from CFSA that allow for more efficient parsing, both through syntactic and semantic restrictions. It is also not unexpected that finding efficiently decidable membership problems is difficult. This paper demonstrates that the efficiency of deciding membership for the shuffle languages relies heavily on using only a limited number of shuffle operations. On the other hand, the non-uniform membership problem for the shuffle of a shuffle language and a context-free language can be decided in polynomial time. Considering the difficulty of the membership problem for very restricted CFSA it is also positive to find that the general uniform membership problem for CFSA is just NP-complete. For the shuffle of a regular language and a context-free language even the uniform membership problem is polynomial.

Future work will strive to determine the complexity of the non-uniform membership problem for further restrictions of CFSA. For example unordered shuffle could be considered, or perhaps the class of languages that a CFSA could generate if at most one path in the configuration tree may exceed a constant depth at a time, or even cases where the CFSA is forced to encode some information about the shuffling choices in the string. In the other direction, to better understand the boundary where the membership problem turns NP-hard, the construction used to demonstrate that the membership problem for the shuffle of two deterministic linear context-free languages should be possible to extend (to, e.g., visibly pushdown languages [1]). Success in this direction will, hopefully, give a better understanding of what properties of a CFSA need to be restricted to get an efficient membership problem while still giving rise to a powerful class of languages.

Acknowledgement

We thank an anonymous reviewer of a previous version of this paper for informing us that the non-uniform version of the membership problem is NP-hard for the shuffling of two deterministic context-free languages [40] and for many helpful suggestions. Henrik Björklund was supported by the Swedish Research Council (Vetenskapsrådet) grant 621-2011-6080.

References

[1] Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing. STOC '04, New York, NY, USA, ACM (2004) 202–211

[2] Barton, G.E.: On the complexity of ID/LP parsing 1. Computational Linguistics **11**(4) (1985) 205–218

[3] Berglund, M., Björklund, H., Högberg, J.: Recognizing shuffled languages. In: Proc. Language and Automata Theory and Applications. (2011) 142–154

[4] Berstel, J., Boasson, L., Carton, O., Pin, J.E., Restivo, A.: The expressive power of the shuffle product. Information and Computation **208**(11) (2010) 1258–1272

[5] Biegler, F., Daley, M., McQuillan, I.: On the shuffle automaton size for words. In: Proc. Descriptional Complexity of Formal Systems. (2009) 79–89

[6] Björklund, H., Bojańczyk, M.: Shuffle expressions and words with nested data. In: Proc. Mathematical Foundations of Computer Science. (2007) 750–761

[7] Bloom, S.B., Ésik, Z.: Axiomatizing shuffle and concatenation in languages. Information and Comptuation **139**(1) (1997) 62–91

[8] Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: Proc. Logic in Computer Science. (2006) 7–16

[9] Brzozowski, J., Jirskov, G., Li, B.: Quotient complexity of ideal languages. In López-Ortiz, A., ed.: Proc. Latin American Theoretical Informatics Symposium. Volume 6034 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2010) 208–221

[10] Câmpeanu, C., Salomaa, K., Yu, S.: Tight lower bound for the state complexity of shuffle of regular languages. Journal of Automata, Languages and Combinatorics **7** (2002) 303–310

[11] Carberry, S.: Techniques for plan recognition. User Modeling and User-Adapted Interaction **11**(1-2) (2001) 31–48

[12] Colcombet, T.: On families of graphs having a decidable first order theory with reachability. In: Proc. International Colloquium on Automata, Languages and Programming. (2002) 98–109

[13] Daley, M., Domaratzki, M., Salomaa, K.: Orthogonal concatenation: Language equations and state complexity. Journal of Universal Computer Science **16**(5) (2010) 653–675

[14] Darrasse, A., Panagiotou, K., Roussel, O., Soria, M.: Boltzmann generation for regular languages with shuffle. In: Proc. GASCOM 2010, Montréal, Canada (2010)

[15] Downey, R., Fellows, M.: Parameterized Complexity. Springer-Verlag (1999)

[16] Ésik, Z., Bertol, M.: Nonfinite axiomatizability of the equational theory of shuffle. Acta Informatica **35**(6) (1998) 505–539

[17] Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer-Verlag (2006)

[18] Garey, M.R., Johnson, D.S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1990)

[19] Garg, V., Ragunath, M.: Concurrent regular expressions and their relationship to Petri nets. Theoretical Computer Science **96**(2) (1992) 285–304

[20] Gelade, W., Martens, W., Neven, F.: Optimizing schema languages for XML: Numerical constraints and interleaving. SIAM Journal on Computing **39**(4) (2009) 1486–1530

[21] Ginsburg, S.: The Mathematical Theory of Context Free Languages. McGraw-Hill (1966)

[22] Ginsburg, S., Spanier, E.H.: Mappings of languages by two-tape devices. J. ACM **12** (July 1965) 423–434

[23] Gischer, J.: Shuffle languages, Petri nets, and context-sensitive grammars. Communications of the ACM **24**(9) (1981) 597–605

[24] Gómez, A.C., Pin, J.E.: Shuffle on positive varieties of languages. Theoretical Computer Science **312** (2004) 433–461

[25] Gruber, H., Holzer, M.: Finite automata, digraph connectivity, and regular expression size. In: Proc. International Colloquium on Automata, Languages and Programming, Springer (2008)

[26] Haines, L.H.: On free monoids partially ordered by embedding. Journal of combinatorial theory (6) (1968) 94–98

[27] Han, Y.S., Salomaa, K., Wood, D.: Operational state complexity of prefix-free regular languages. In: Proc. Automata, Formal Languages, and Related Topics. (2009) 99–115

[28] Haussler, D., Zeiger, H.P.: Very special languages and representations of recursively enumerable languages via computation histories. Information and Control **47**(3) (1980) 201 – 212

[29] Högberg, J., Kaati, L.: Weighted unranked tree automata as a framework for plan recognition. In: Proc. Fusion. (2010)

[30] Hopcroft, J.E., Ullman, J.D.: Introduction To Automata Theory, Languages, And Computation. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)

[31] Jedrzejowicz, J., Szepietowski, A.: Shuffle languages are in P. Theorical Computer Science **250**(1-2) (2001) 31–53

[32] Kuhlmann, M., Satta, G.: Treebank grammar techniques for non-projective dependency parsing. In: Proc. Conference of the European Chapter of the Association for Computational Linguistics. (2009) 478–486

[33] Löding, C.: Ground tree rewriting graphs of bounded tree width. In: Proc. Symposium on Theoretical Aspects of Computer Science. (2002) 559–570

[34] Mateescu, A., Rozenberg, G., Salomaa, A.: Shuffle on trajectories: syntactic constraints. Theoretical Computer Science **197**(1-2) (1998) 1–56

[35] Mayer, A., Stockmeyer, L.: Word problems – this time with interleaving. Information and Computation **115** (1994) 293–311

[36] Minsky, M.L.: Computation: finite and infinite machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)

[37] Nagy, B.: Languages generated by context-free grammars extended by type $AB \to BA$ rules. Journal of Automata, Languages and Combinatorics **14** (2009) 175–186

[38] Nagy, B.: On a hierarchy of permutation languages. In Ito, M., Kobayashi, Y., Shoji, K., eds.: Proc. Automata, Formal Languages and Algebraic Systems, World Scientific, Singapore (2010) 163–178

[39] Nivre, J.: Non-projective dependency parsing in expected linear time. In: Proc. Annual Meeting of the Association for Computational Linguistics and the Joint Conference on NLP of the Asian Federation of NLP. (2009) 351–359

[40] Ogden, W.F., Riddle, W.E., Rounds, W.C.: Complexity of expressions allowing concurrency. In: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. POPL '78, New York, NY, USA, ACM (1978) 185–194

[41] Riddle, W.E.: An approach to software system behavior description. Comput. Lang. **4**(1) (January 1979) 29–47

[42] Schmidt, C., Sridharan, N., Goodson, J.: The plan recognition problem: An intersection of psychology and artificial intelligence. Artificial Intelligence **11**(1,2) (1978)

[43] Shaw, A.: Software descriptions with flow expressions. IEEE Transactions on Software Engineering **4** (1978) 242–254

[44] Warmuth, M.K., Haussler, D.: On the complexity of iterated shuffle. Journal of Computer and System Sciences **28**(3) (1984) 345 – 358

II

On the Parameterized Complexity of Linear Context-Free Rewriting Systems

Martin Berglund
Umeå University, Sweden
mbe@cs.umu.se

Henrik Björklund
Umeå University, Sweden
henrikb@cs.umu.se

Frank Drewes
Umeå University, Sweden
drewes@cs.umu.se

Abstract

We study the complexity of uniform membership for Linear Context-Free Rewriting Systems, i.e., the problem where we are given a string w and a grammar G and are asked whether $w \in \mathcal{L}(G)$. In particular, we use parameterized complexity theory to investigate how the complexity depends on various parameters. While we focus primarily on rank and fan-out, derivation length is also considered.

1 Introduction

Linear Context-Free Rewriting Systems (LCFRS) were introduced by Vijay-Shanker et al. (1987) with the purpose of capturing the syntax of natural language.[1] It is one of several suggested ways of capturing Joshi's concept of *mildly context-sensitive languages* (Joshi, 1985). As such, it strengthens the expressive power of context-free grammars, while avoiding the full computational complexity of context-sensitive grammars.

One of the defining features of mildly context-sensitive languages is that they should be decidable in polynomial time. This is indeed true for every language that can be generated by an LCFRS. Unlike the case for context-free grammars, however, the *universal* or *uniform* membership problem for LCFRSs, where both the grammar and the string in question are considered as input, is known to be PSPACE-complete (Kaji et al., 1992), making a polynomial time solution very improbable.

The best known algorithms for the problem have a running time of $\mathcal{O}(|G| \cdot |w|^{f\cdot(r+1)})$, where G is the grammar, w is the string, f is the *fan-out* and r is the *rank* of the grammar (Seki et al., 1991; Burden and Ljunglöf, 2005; Boullier, 2004). (For a definition of fan-out and rank, see Section 2.) Unlike the rank of a context-free grammar, the fan-out and rank of an LCFRS cannot in general be reduced to some fixed constant. Increasing the fan-out always gives more generative power, as does increasing the rank while keeping the fan-out fixed (Satta, 1998). The rank can be reduced to two, but at the price of an exponential increase in the fan-out.

Research into algorithms for LCFRS parsing that are efficient enough for practical use is quite active. For example, algorithms for restricted cases are being studied, e.g., by Gómez-Rodríguez et al. (2010), as well as rank reduction, primarily in special cases, where the fan-out is not affected; see, e.g., Sagot and Satta (2010).

This article is a first step towards a finer computational complexity analysis of the membership problem for LCFRSs. Specifically it asks the question "could there exist an algorithm for the uniform LCFRS membership problem whose running time is a fixed polynomial in $|w|$ times an arbitrary function in f and r?" By employing parameterized complexity theory, we show that such an algorithm is very unlikely to be found. Fixing the rank of the grammar to one, the membership problem, parameterized by the fan-out, is W[SAT]-hard. Fixing the fan-out to two and taking the rank as the parameter, the problem is W[1]-hard. Finally, if the fan-out, rank, *and derivation length* are included in the parameter, the problem is W[1]-complete. These results help guide future work, suggesting other types of parameters and grammar restrictions that may yield more favorable complexity results.

2 Preliminaries

For $n \in \mathbb{N}$, we write $[n]$ for $\{1, \ldots, n\}$ and $[n]_0$ for $\{0\} \cup [n]$. Given an alphabet Σ we write Σ^* for all strings over Σ and Σ^+ for all non-empty strings. The empty string is denoted by ε.

[1] Seki et al. (1991) independently suggested the nearly identical Multiple Context-Free Grammars.

2.1 Linear context-free rewriting systems

Let Σ be an alphabet, $x_1, \ldots, x_n$ variables, and $w_1, \ldots, w_k$ strings over Σ such that

$$w_1 \cdots w_k = \alpha_0 \cdot x_{\pi(1)} \cdot \alpha_1 \cdots x_{\pi(n)} \cdot \alpha_n$$

for some permutation π and some strings $\alpha_0, \ldots, \alpha_n \in \Sigma^*$. Then define f as a function over tuples of strings such that

$$f((x_1, \ldots), \ldots, (\ldots, x_n)) = (w_1, \ldots, w_k).$$

A function is *linear regular* if and only if it can be described in this way. For example $f((x_1), (x_2)) = (a, bx_2x_1c)$ is linear regular, and $f((aaa), (bc)) = (a, bbcaaac)$.

Definition 2.1. A *Linear Context-Free Rewriting System* is a tuple $G = (N, \Sigma, F, R, S)$, where N is an alphabet of *nonterminals*, where each $A \in N$ has an associated *fan-out* $\#(A)$; $S \in N$ is the initial nonterminal with $\#(S) = 1$; Σ is an alphabet of *terminals*; F is a set of linear regular functions; and R is a set of rules of the form $A \to g(B_1, \ldots, B_n)$, where $A, B_1, \ldots, B_n \in N$ and g is a function in F of type

$$(\Sigma^*)^{\#(B_1)} \times \cdots \times (\Sigma^*)^{\#(B_n)} \to (\Sigma^*)^{\#(A)}.$$

For rules $A \to g()$, where g has arity 0 and $g() = (\alpha_1, \ldots, \alpha_{\#(A)})$, we often simply write $A \to (\alpha_1, \ldots, \alpha_{\#(A)})$.

The *rank* of a rule is the number of nonterminals on the right-hand side. The rank of G is the maximal rank of any rule in R. The *fan-out* of G is the maximal fan-out of any nonterminal in N.

The language generated by a nonterminal A is a set of n-tuples, where $n = \#(A)$.

Definition 2.2. Let $G = (N, \Sigma, F, R, S)$ be a linear context-free rewriting system. Let $\mathcal{L}_A \subseteq (\Sigma^*)^{\#(A)}$ denote the tuples that a nonterminal $A \in N$ can generate. This is the smallest set such that if $A \to f(B_1, \ldots, B_n)$ is in R then, for all $b_i \in \mathcal{L}_{B_i}$ where $i \in [n]$, $f(b_1, \ldots, b_n) \in \mathcal{L}_A$. The language of G is $\mathcal{L}(G) = \mathcal{L}_S$.

For $i \in \mathbb{N}$, we write i-LCFRS for the class of all LCFRSs of rank at most i and LCFRS(i) for the class of all LCFRSs of fan-out at most i. We also write i-LCFRS(j) for i-LCFRS$\cap$LCFRS(j).

2.2 Parameterized complexity theory

We only reproduce the most central definitions of parameterized complexity theory. For a more thorough treatment, we refer the reader to (Downey and Fellows, 1999; Flum and Grohe, 2006).

A *parameterized problem* is a language $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a finite alphabet. The second component is called the *parameter*. An algorithm for $\mathcal{L}$ is *fixed-parameter tractable* if there is a computable function f and a polynomial p such that for every $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm decides in time $f(k) \cdot p(|x|)$ whether $(x, k) \in \mathcal{L}$. The problem of deciding $\mathcal{L}$ is fixed-parameter tractable if there is such an algorithm. If so, $\mathcal{L}$ belongs to the class FPT.

A parameterized problem $\mathcal{L} \subseteq \Sigma^* \times \mathbb{N}$ is *fpt-reducible* to another parameterized problem $\mathcal{K} \subseteq \Gamma^* \times \mathbb{N}$ if there is a mapping $R : \Sigma^* \times \mathbb{N} \to \Gamma^* \times \mathbb{N}$ such that

1. for all $(x, k) \in \Sigma^* \times \mathbb{N}$, we have $(x, k) \in \mathcal{L}$ if and only if $R(x, k) \in \mathcal{K}$,
2. there is a computable function f and a polynomial p such that $R(x, k)$ can be computed in time $f(k) \cdot p(|x|)$, and
3. there is a computable function g such that for every $(x, k) \in \Sigma^* \times \mathbb{N}$, if $R(x, k) = (y, k')$, then $k' \leq g(k)$.

Note that several parameters may be combined into one by taking their maximum (or sum).

The most commonly used hierarchy of parameterized complexity classes is the following.

$$\text{FPT} \subseteq \text{W[1]} \subseteq \text{W[2]} \subseteq \cdots \subseteq \\ \subseteq \text{W[SAT]} \subseteq \text{W[P]} \subseteq \text{XP}$$

The classes W[1],...,W[P] are defined using circuits or, alternatively, logic. None of the inclusions is known to be strict, except that FPT is a strict subclass of XP. It is widely believed, however, that each of them is strict. The class XP is the class of all parameterized problems for which there is a computable function f such that every instance (x, k) can be decided in time $|x|^{f(k)}$.

2.3 Problems of interest

We know from Kaji et al. (1992) that the universal membership problem for 1-LCFRSs is PSPACE-complete. Satta (1992) has further shown that LCFRS(2)-MEMBERSHIP is NP-hard.

We study the following decision problems, where the symbol $\mathfrak{P}$ is used to indicate what the parameter is:

- $\mathfrak{P}$-LCFRS(j)-MEMBERSHIP, where $j \in \mathbb{N}$ is the membership problem for LCFRS(j), parameterized by the rank.

- i-LCFRS($\mathfrak{P}$)-MEMBERSHIP, where $i \in \mathbb{N}$ is the membership problem for i-LCFRS, parameterized by the fan-out.
- $\mathfrak{P}$-LCFRS($\mathfrak{P}$)-MEMBERSHIP is the membership problem for LCFRS parameterized by the rank and the fan-out.
- SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$)-DERIVATION is the membership problem for LCFRS parameterized by the rank, the fan-out, and the derivation length.

Since there are algorithms that solve the membership problem for LCFRSs with rank r and fan-out t and string w in time $|w|^{(r+1)t}$ (see, e.g., (Seki et al., 1991; Burden and Ljunglöf, 2005; Boullier, 2004)), we can immediately conclude that $\mathfrak{P}$-LCFRS($\mathfrak{P}$)-MEMBERSHIP, as well as every other parameterized membership problem mentioned above, belongs to XP.

3 Fixed rank grammars

The following theorem establishes a lower bound for 1-LCFRSs parameterized by the fan-out.

Theorem 3.1. 1-LCFRS($\mathfrak{P}$)-MEMBERSHIP *is* W[SAT]*-hard.*

The proof of Theorem 3.1 is by reduction from WEIGHTED MONOTONE SATISFIABILITY. Before we get into the actual proof, we discuss some properties of this problem.

Definition 3.1. A *monotone Boolean formula* is a Boolean formula that contains only conjunctions, disjunctions, and variables. In particular, there are no negations. An instance of WEIGHTED MONOTONE SATISFIABILITY is a pair (ϕ, k), where ϕ is a monotone Boolean formula and $k \in \mathbb{N}$. The question is whether ϕ has a satisfying assignment of weight k, i.e., an assignment that sets exactly k of the variables that occur in ϕ to true. The parameter is k. WEIGHTED MONOTONE SATISFIABILITY is W[SAT]-complete (Abrahamson et al., 1993; Abrahamson et al., 1995; Downey and Fellows, 1999).

We can view a monotone Boolean formula ϕ as an unranked tree, where the root node corresponds to the top level clause and the leaves correspond to bottom level clauses, i.e., variable occurrences. The set $pos(\phi)$ of *positions* of ϕ is defined as usual, consisting of strings of natural numbers that indicate how to navigate to the clauses in a tree representation of ϕ. We denote each subclause of ϕ by C_s, where $s \in pos(\phi)$ is its position. Thus

$$\phi = (((x_1 \wedge (x_2 \vee x_3)) \vee x_3 \vee (x_3 \wedge x_4)) \wedge \\ \wedge x_2 \wedge ((x_1 \wedge (x_2 \vee x_4)) \vee (x_1 \wedge x_3)))$$

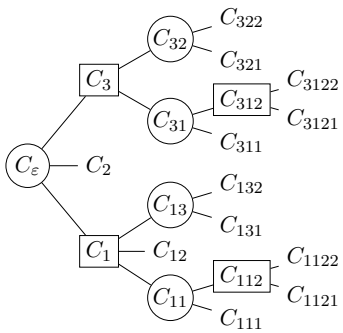


Figure 1: A formula ϕ and its tree representation. Conjunctive clauses are round and disjunctive rectangular. For example, C_{111} is the leftmost occurrence of x_1 and C_{13} the clause $(x_3 \wedge x_4)$.

C_ε denotes the whole of ϕ, while, e.g., C_{ijl} is the lth clause of the jth clause of the ith clause of ϕ. See Figure 1 for an example. We use $\mathcal{C}$ for the set of all clauses of ϕ and $\mathcal{C}_\wedge, \mathcal{C}_\vee$, and $\mathcal{C}_{Var}$ for the sets of conjunctive, disjunctive, and bottom level clauses, respectively. For all $c \in \mathcal{C}_{Var}$ let $Var(c)$ denote the variable in c, and let $Var(\phi)$ denote the set of all variables in ϕ.

Given a monotone Boolean formula ϕ and a variable assignment $\rho\colon Var(\phi) \to \mathbb{B}$, we define a *verification tour* for ϕ and ρ. Such a tour moves through the tree representation of ϕ, starting at the root node, and verifies that ρ satisfies ϕ. To this end, we first define the function $Next : pos(\phi) \to pos(\phi) \cup \{True\}$ as follows. For the root clause let $Next(\varepsilon) = True$. For all $si \in pos(\phi)$, where $s \in \mathbb{N}^*$ and $i \in \mathbb{N}$, if $C_s \in \mathcal{C}_\wedge$ and $s(i+1) \in pos(\phi)$ let $Next(si) = s(i+1)$, otherwise let $Next(si) = Next(s)$.

A verification tour over ϕ, given a variable assignment ρ is constructed by the following procedure. Set the initial position $p = \varepsilon$, then

- If $C_p \in \mathcal{C}_\wedge$ set $p \leftarrow p1$ (i.e., go to the first subclause).
- If $C_p \in \mathcal{C}_\vee$ set $p \leftarrow pi$ for any i (i.e. nondeterministically pick a subclause).
- If $C_p \in \mathcal{C}_{Var}$ verify that $\rho(Var(C_p)) = true$. If so, set $p \leftarrow Next(p)$ and repeat. Otherwise, the verification tour fails.

A verification tour succeeds if it reaches $True$.

The following lemma can be proved by straightforward induction on the structure of ϕ.

Lemma 3.2. *If a verification tour for ϕ and variable assignment ρ succeeds, then ρ satisfies ϕ.*

We are now ready to prove Theorem 3.1.

Proof of Theorem 3.1. Let (ϕ, k) be an instance of WEIGHTED MONOTONE SATISFIABILITY. Let $\{x_1, \ldots, x_n\}$ be the variables that appear in ϕ. In particular, n is the number of distinct variables. Let m be the number of bottom level clauses.

Intuitively, the LCFRS we will construct will guess a weight k variable assignment ρ and then simulate a verification tour for ϕ and ρ.

Basically, we will use one nonterminal per clause and use the structure of the grammar to simulate a verification tour. In order to verify that the necessary bottom level clauses can all be satisfied through the same k true variables, we will use the input string to be parsed. The string w will consist of bracketed sequences of m copies of each of the n variables, i.e., $w = [x_1^m] \cdots [x_n^m]$. To understand the construction of the grammar, please keep in mind that the only derivations that matter are those generating this particular input string.

The grammar will guess which k variables should be set to true and disregard the other variables. Technically, this is done by first letting a nonterminal F generate a tuple of $k+1$ strings $s_0, \ldots, s_k$ such that each s_i consists of zero or more of the bracketed sequences of variables to be disregarded. The rest of the grammar generates exactly k bracketed sequences that will be interleaved with $s_0, \ldots, s_k$. During the generation of these k bracketed sequences it is nondeterministically verified that the corresponding truth assignment satisfies ϕ.

We use the following set of nonterminals:

$$\{S, F\} \cup \{C_s \mid s \in pos(\phi) \cup \{True\}\}$$

For S, there is only one rule: $S \to f_S(F)$. The function f_S places brackets around the k variables that are guessed to be true, represented by the strings $t_1, \ldots, t_k$, and interleaves them with the remaining variables, represented by the strings $s_0, \ldots, s_k$:

$$f_S(s_0, \ldots, s_k, t_1, \ldots, t_k) = (s_0[t_1]s_1 \cdots [t_k]s_k)$$

The nonterminal F has rules $F \to f_{F,i,j}(F)$ for all $i \in [n]$ and $j \in [k]_0$. These rules produce the bracketed sequences of copies of the variables x_i to be disregarded, as can be seen from the corresponding function:

$$f_{F,i,j}(s_0, \ldots, s_k, t_1, \ldots, t_k) = (s_0, \ldots, s_j[x_i^m], \ldots, s_k, t_1, \ldots, t_k)$$

Moreover, there is a single rule

$$F \to f_F(C_\varepsilon)$$

with

$$f_F(t_1, \ldots, t_k) = (\varepsilon, \ldots, \varepsilon, t_1, \ldots, t_k)$$

The rules for the nonterminals that represent clauses differ according to the type of the clause, i.e., if the nonterminal represents a conjunctive clause, a disjunctive clause, or a variable. For each conjunctive clause C_s there is exactly one rule, representing a move to its first subclause. Here, f_{id} is the identity function.

$$C_s \to f_{id}(C_{s1})$$

For every disjunctive clause C_s and every i such that C_{si} is a subclause of C_s there is one rule.

$$C_s \to f_{id}(C_{si})$$

For every bottom level clause, i.e., $C_s \in \mathcal{C}_{Var}$, every $i \in [k]$ and every $j \in [m]$ there is one rule.

$$C_s \to f_{s,i,j}(C_{Next(s)})$$

Intuitively, such a rule corresponds to producing j copies of the variable of clause C_s in component i of the tuple and moving on to the next clause that should be visited in a verification tour. This can be seen from the corresponding function.

$$f_{s,i,j}(t_1, \ldots, t_k) = (t_1, \ldots, Var(C_s)^j t_i, \ldots, t_k)$$

The reason that the function produces j copies of the variable, rather than just one, is that it is unknown beforehand how many times a bottom level clause that represents that particular variable will be visited. Thus the number of copies to be produced has to be guessed nondeterministically in order to make sure that a total of m copies of each variable set to true are eventually produced.

If there is a weight k satisfying assignment, there will also be a verification tour that eventually reaches $True$ when $Next$ is called (by Lemma 3.2). The single rule for C_{True} simply produces a k-tuple of empty strings.

The reduction is polynomial and the fan-out of the resulting grammar is $2k+1$. Thus it is an FPT-reduction. It remains to argue that the grammar can produce w if and only if ϕ has a satisfying assignment of weight k.

We first note that whatever tuple is derived from F, the first $k+1$ entries in the tuple consist of bracketed sequences of the form $[x_l^m]$. If the grammar can produce w, it follows that the tuple $(t_1, \ldots, t_k)$ produced from C_ε must be such that each t_i equals m copies of the same variable name.

Any successful derivation of a string by the grammar corresponds to a verification tour of ϕ and the variable assignment that sets the variables that appear in $(t_1, \ldots, t_k)$ to true and all other variables to false. Thus ϕ has a satisfying assignment of weight k.

For the other direction, assume that ϕ has a satisfying assignment of weight k. Then the grammar can guess this assignment and a corresponding successful verification tour, thus producing w. □

Note that Theorem 3.1 can easily be strengthened to grammars with a binary terminal alphabet. It is enough to represent each variable name by a bitstring of length $\lceil \log_2(m) \rceil$ in the above reduction. We also note that Theorem 3.1 immediately implies that $\mathfrak{P}$-LCFRS($\mathfrak{P}$)-MEMBERSHIP is W[SAT]-hard.

4 Fixed fan-out grammars

We next turn to the case where the fan-out is fixed to two, while the rank is treated as a parameter.

Theorem 4.1. $\mathfrak{P}$-LCFRS(2)-MEMBERSHIP *is* W[1]*-hard.*

Proof. We reduce from k-CLIQUE, the problem of deciding whether a given graph has a clique of size k, with k as the parameter. This problem is known to be W[1]-complete (Flum and Grohe, 2006). Let $G = (\mathsf{V}, \mathsf{E})$ be an undirected graph. We assume, without loss of generality, that $\mathsf{V} = \{1, \ldots, n\}$ and that an edge connecting nodes $i, j \in \mathsf{V}$ is represented as the ordered pair (i, j) such that $i < j$, i.e., $\mathsf{E} \subseteq \{(i, j) \in \mathsf{V} \times \mathsf{V} \mid i < j\}$. To find out whether G has a clique of size k we construct an instance of the membership problem for LCFRSs.

The input alphabet is $\Sigma = \{0, 1\}$. Construct the input string as

$$w = \underbrace{0^n 1 0^n 1 0^n 1 \cdots 1 0^n}_{(3k+2)(k-1)/2 \text{ ones}}.$$

The nonterminals are $N = \{A, E, C, S\}$, with S being the initial nonterminal. The rules are the following.

$\{A \to 0^i \mid i \in \{1, \ldots, n\}\}$.
$\{E \to 0^{n-i} 1 0^{n-j} \mid (i, j) \in \mathsf{E}\}$.
$\{C \to (0^i, 0^{n-i} 1 0^i) \mid i \in \{1, \ldots, n\}\}$.

Handling S is a bit more complex. Let $\phi = k(k-1)/2$, the number of edges in a k-clique. Then the unique rule for S is:

$$S \to f(\underbrace{E, \ldots, E}_{\phi}, \underbrace{C, \ldots, C}_{2\phi}, \underbrace{A, \ldots, A}_{2k}).$$

Now we need to define f. Consider the following application of f.

$$f(e_1, \ldots, e_\phi, (c_1, \hat{c}_1), \ldots, (c_\phi, \hat{c}_\phi), \\ (d_1, \hat{d}_1), \ldots, (d_\phi, \hat{d}_\phi), a_1, \ldots, a_{2k}).$$

The application above evaluates to the string

$$c_1 e_1 d_1 1 c_2 e_2 d_2 1 \cdots \\ \cdots 1 c_\phi e_\phi d_\phi 1 a_1 \theta_1 a_2 1 a_3 \theta_2 a_4 1 s 1 a_{2k-1} \theta_k a_{2k}.$$

The substrings θ_1 through θ_k are left to be defined, and will contain all the $\hat{c}$ and $\hat{d}$ arguments in a careful configuration derived from the structure of a clique. Let $(\pi_1, \pi'_1), \ldots, (\pi_\phi, \pi'_\phi)$ be the lexicographically sorted sequence of edges in a k-clique with nodes numbered 1 through k. For example, $(\pi_1, \pi'_1) = (1, 2)$, $(\pi_2, \pi'_2) = (1, 3)$, $(\pi_k, \pi'_k) = (2, 3)$, and $(\pi_\phi, \pi'_\phi) = (k-1, k)$. Then, for each l, find the longest subsequences $i_1, \ldots, i_p$ and $j_1, \ldots, j_q$ of $1, \ldots, \phi$ for which $\pi_{i_1} = \cdots = \pi_{i_p} = l$ and $\pi'_{j_1} = \cdots = \pi_{j_q} = l$, and let $\theta_l = \hat{c}_{i_1} \cdots \hat{c}_{i_p} \hat{d}_{j_1} \cdots \hat{d}_{j_q}$. □

This construction is simpler than it may at first appear. Basically, the clique is found by generating $k(k-1)/2$ copies of E, each of which will be placed so that it has no choice but to generate an edge in a k-clique. Looking at the first part of the string, each $1 c_l e_l d_l 1$ must generate a string of the form $10^n 10^n 1$: e_l will generate some $0^{n-i} 1 0^{n-j}$, were (i, j) is an edge in G, which forces c_l to generate 0^i and d_l to generate 0^j. The trick is that c_l and d_l yield the first string in a *pair* generated by an instance of C. The *other* string in the pair describes the same number as the first, but in such a way that it can be carefully placed in the latter part of the derivation string, thus forcing other instances of the C nonterminal to pick *the same*

node (number of zeros) to generate. These are then placed in such a way that the edges picked by the instances of E all belong to the same clique. For example, for $k = 3$ the result of f will be $c_1e_1d_11c_2e_2d_21c_3e_3d_31a_1c_1c_2a_21c_3d_11d_2d_3$, where the latter part ensures that c_1 and c_2 have to pick the same node (lowest-numbered node in the clique), as do c_3 and d_1, and d_2 and d_3.

5 Short derivations

In this section, we consider the length of derivations as an additional parameter. As usual, the length of a derivation is the number of derivation steps it consists of. (In a derivation of an LCFRS (N, Σ, F, R, S), this is the same as the number of applications of functions in F.)

Let $G = (N, \Sigma, F, R, S)$ be an LCFRS in the following. Consider the following problem:

Definition 5.1. An instance of the SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION problem consists of a LCFRS G, some $w \in \Sigma^*$ and a constant $d \in \mathbb{N}$. The question asked is: can w be derived by G in at most d steps? The parameter is $k = d + r + f$ where r is the maximum rank and f the maximum fanout.

Lemma 5.1. SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION *is* W[1]*-hard.*

Proof. The W[1]-hardness of the problem follows immediately from the reduction in the proof of Theorem 4.1, since k-Clique is reduced to an instance of LCFRS membership with $\mathcal{O}(k^2)$ derivation steps, rank $\mathcal{O}(k^2)$, and fixed fan-out. □

We next demonstrate that SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION is *in* W[1] (and is therefore W[1]-complete) by reducing to SHORT CONTEXT-SENSITIVE DERIVATION, shown to be W[1]-complete by Downey et al. (1994). Let $H = (N_H, \Sigma_H, R_H, S_H)$ be an arbitrary context-sensitive grammar in the following. A context-sensitive grammar has nonterminals, terminals and a starting nonterminal just like a LCFRS, but the rules are of the form $\alpha \to \beta$ for strings $\alpha, \beta \in (\Sigma_H \cup N_H)^*$ where $0 < |\alpha| \leq |\beta|$. A derivation starts with the string S_H. A string $w \cdot \alpha \cdot w'$ can be turned into $w \cdot \beta \cdot w'$ in one derivation step if $(\alpha, \beta) \in R_H$.

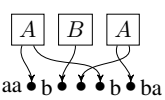
Definition 5.2. An instance of the SHORT CONTEXT-SENSITIVE DERIVATION problem consists of a context-sensitive grammar H, a string $w \in \Sigma_H^*$, and a constant $d_H \in \mathbb{N}$. The question is: can w be derived by H in at most d_H steps? The parameter is d_H.

We are now ready to prove membership in W[1] by a FPT-reduction from (G, w, d) to (H, w, d_H).

Lemma 5.2. *The* SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION *problem is* in W[1].

Proof. We can restrict ourselves to the case where no nonterminal appears twice in a right-hand side of any rule in G. This is because, e.g., a rule of the form $A \to f(B, B)$ can be turned into $A \to f(B, B')$, using a fresh copy B' of B that has the same rules as B (except for having the left-hand side B' rather than B). Note that this modification does not affect the parameter, and increases the size of the grammar only polynomially.

The complete reduction is somewhat lengthy, but the core intuition is very simple. The string is kept the same, and a context-sensitive grammar H is constructed such that $\mathcal{L}(H) = \mathcal{L}(G)$. H simulates G by maintaining a string serialization of the current "configuration" of G, walking through the whole string rewriting the appropriate non-terminal for every rule application in G. A configuration of G can be viewed in this way,



where the derivation has, so far, generated some terminal symbols (the lower-case letters), two instances of the non-terminal A and one instance of B. The configuration keeps track of where the symbols generated by the non-terminals should go in the string, so $\#(A) = 2$, $\#(B) = 1$, and if $(c, d) \in \mathcal{L}_A$ and $e \in \mathcal{L}_B$ this derivation can generate the final string $aacbeddbcba$. These intermediary configurations are in H serialized into strings of nonterminals, with a "nonterminal marker" symbol in each position where a nonterminal is referred to (i.e., H generates a symbol stating "the ith string generated by instance j of the nonterminal A goes here"). H then operates like a Turing machine. A special nonterminal, the rewriting head, picks a rule from G to apply, and walks through the string replacing the nonterminal markers that are affected by that rule. This procedure is then repeated d times.

We start by illustrating the principles of the reduction by an example. Consider the grammar

$\triangleright P_{r_1,1\to 2}X_{S,1,1}\triangleleft \Longrightarrow \triangleright X_{A,1,2}X_{A,2,2}P_{r_1,1\to 2}\triangleleft \Longrightarrow \triangleright X_{A,1,2}X_{A,2,2}R\triangleleft \Longrightarrow \triangleright X_{A,1,2}RX_{A,2,2}\triangleleft \Longrightarrow$
$\triangleright RX_{A,1,2}X_{A,2,2}\triangleleft \Longrightarrow \triangleright P_{r_2,2\to 3}X_{A,1,2}X_{A,2,2}\triangleleft \overset{*}{\Longrightarrow} \triangleright X_{A,1,3}X_{B,1,3}X_{A,2,3}B_{2,3}R\triangleleft \overset{*}{\Longrightarrow}$
$\triangleright P_{r_2,3\to 4}X_{A,1,3}X_{B,1,3}X_{A,2,3}X_{B,2,3}\triangleleft \overset{*}{\Longrightarrow} \triangleright X_{A,1,4}X_{B,1,4}X_{B,1,3}X_{A,2,3}X_{B,2,4}X_{B,2,3}R\triangleleft \overset{*}{\Longrightarrow}$
$\triangleright P_{r_6,3\to 1}X_{A,1,4}X_{B,1,4}X_{B,1,3}X_{A,2,3}X_{B,2,4}X_{B,2,3}\triangleleft \overset{*}{\Longrightarrow} \triangleright P_{r_3,4\to 1}X_{A,1,4}X_{B,1,4}bX_{A,2,3}X_{B,2,4}b\triangleleft \overset{*}{\Longrightarrow}$
$\triangleright P_{r_5,4\to 1}aX_{B,1,4}baX_{B,2,4}b\triangleleft \overset{*}{\Longrightarrow} \triangleright aabaabR\triangleleft \overset{*}{\Longrightarrow} aabaab$

Figure 2: A derivation in the context-sensitive grammar constructed to simulate an LCFRS. All steps in the application of the first rule, $r_1 = S \to f(A)$, are given, the rest is abbreviated.

$G = (\{S, A, B\}, \{a, b\}, F, R, S)$ where F is

$$\{f(x, y) = xy, \quad h_a() = (a, a), \quad h_b() = (b, b), \\ g((x, y), (x', y')) = (xx', yy')\},$$

and R contains the following

$$\begin{array}{ll} r_1 = S \to f(A) & r_2 = A \to g(A, B) \\ r_3 = A \to h_a() & r_4 = A \to h_b() \\ r_5 = B \to h_a() & r_6 = B \to h_b() \end{array}$$

Notice that $\mathcal{L}(G) = \{ww \mid w \in \{a, b\}^+\}$. We now describe how H is constructed by the reduction, after which the more general description follows. A derivation in G starts with the nonterminal S and must then apply r_1. H is constructed to start with the string $\triangleright P_{r_1,1\to 2}X_{S,1,1}\triangleleft$ (all these symbols are nonterminals, H has the same terminal alphabet as G). The symbols $\triangleright$ and $\triangleleft$ mark the beginning and end of the string. The nonterminal $X_{S,1,1}$ is a "nonterminal marker" and denotes the location where the *first* string generated by instance 1 of the nonterminal S is to be placed. Since $\#(S) = 1$ the first string is the only string generated from S. The last subscript, the instance number, is there to differentiate markers belonging to different instances of the same nonterminal. The rewriting head non-deterministically picks an instance number for a round of rewriting (single rule application) from a pool sufficiently large to differentiate between the maximal number of nonterminals (since the rank of G is at most k, no more than k^2 nonterminals can be generated in k rule applications). $P_{r_1,1\to 2}$ is the "rewriting head", the anchor for rule applications. The subscripts on P determines that it will apply the rule r_1, rewriting nonterminal markers corresponding to the left hand side nonterminal of r_1 which have instance number 1. Applying the rule may create new nonterminal markers, all of which get the instance number 2, also determined by the subscript.

That is, the rules for $P_{r_1,i\to j}$ in H will be $P_{r_1,i\to j}X_{S,1,i} \to X_{A,1,j}X_{A,2,j}P_{r_1,i\to j}$, for $i, j \in [2k^2]$, and $P_{r_1,i\to j}x \to xP_{r_1,i\to j}$ for all other $x \neq \triangleleft$. $P_{r_5,i\to j}X_{B,1,i} \to aP_{r_5,i\to j}$ is another example of a rule corresponding to rule r_5 of G. When a rewriting head hits $\triangleleft$ it is replaced by a nonterminal R which reverses through the string (with rules of the form $xR \to Rx$ for all $x \neq \triangleright$), after which a new rewriting head is non-deterministically picked using one of the rules in $\{\triangleright R \to \triangleright P_{r,i\to j} \mid r \in R, i, j \in [2k^2]\}$, after which the string is rewritten once more. Finally, there are rules $\triangleright \to \varepsilon$, $\triangleleft \to \varepsilon$ and $R \to \varepsilon$, to remove all nonterminals once rewriting has terminated. A derivation is demonstrated in Figure 2.

By induction on the length of derivations, one can show that $\mathcal{L}(H) = \mathcal{L}(G)$. Now we need to modify the construction slightly to ensure that H can simulate d steps of G in d_H steps.

Limiting steps in G. Construct a SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION instance (G', w, d) from (G, w, d) where G' is such that it *cannot* perform more than d derivation steps. Let

$$N' = \{A_i \mid A \in N, i \in [d]\},$$

and let

$$A_i \to f(B_{j_1}, C_{j_2}, \ldots) \in R'$$

for all $A \to f(B, C, \ldots) \in R$, $i \in [d]$ and $j_1 + j_2 + \cdots = i - 1$. Then $G' = (N', \Sigma, R', S_1)$. This reduction is somewhat heavy-handed, but is in FPT since it leaves k unchanged and each rule is replaced by less than k^k rules (since d and the rank of the grammar are part of the parameter k).

Deferring terminals. A problem in completing the reduction from (G, w, d) to (H, w, d_H) is that the number of terminal symbols G generates is not in its parameter k. For example, G may contain a rule like $A \to a \cdots a$, for an arbitrary number of as. Applying this rule may make the intermediary string H is operating on too long for it to complete rewriting in d_H steps. This can

easily be fixed by a polynomial-time rewriting of H. For any rule $w \to w'$ in H such that w' contains at least one terminal, replace every maximal substring $\alpha \in \Sigma^*$ by a new nonterminal T_α, a "terminal place-holder".The rewriting head P and reversal nonterminal R just walk over the place-holders without changing them. Now add the rule $T_\alpha \to \alpha$ for each T_α. For example, where a rewriting head in H might have replaced $X_{A,1,1}$ by $abcX_{B,1,1}baX_{B,2,1}cc$ it will now instead replace it by $T_{abc}X_{B,1,1}T_{ba}X_{B,2,1}T_{cc}$, and can defer replacing the place-holder nonterminals until the end.

Completing the reduction. Now we are ready to put all the pieces together. Given the SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION instance (G, w, d), apply the limiting steps reduction to construct (G', w, d'). Apply the rewriting construction to G to get the context-sensitive grammar H. Now $\mathcal{L}(H)$ equals the language G can generate in d steps. Apply the deferring terminals construction to H to get H'. All that remains is to calculate d_H, the number of steps that H' may take. For an FPT-reduction this number may only depend on the parameter k of (G', w, d'). Picking $d_H = k^5 + 10^3$ is sufficient. Each rule in G' generates less than k nonterminals (since the maximum rank is at most k), each of which will generate at most k markers in the derivation in H' (since the fanout is at most k). The rule may in addition generate $(k+1)k$ terminal place-holders (the k^2 nonterminal markers and string ends separating maximal terminal substrings). After k rule applications, without replacing terminal placeholders, the intermediary string in a derivation in H is less than $k(k^2+(k+1)k)+3$ symbols long. Simulating a rule application in H' entails walking the string twice (forward and then reversing), and k rules are applied, giving $2k(k(k^2+(k+1)k)+3)$ steps. Another $k(k+1)+3$ steps at the end replace the terminal place-holders and remove markers and the rewriting head. Adding things up we arrive at a polynomial of degree 4 that can be rounded up to $k^5 + 10^3$. □

Theorem 5.3. SHORT $\mathfrak{P}$-LCFRS($\mathfrak{P}$) DERIVATION *is* W[1]*-hard.*

Proof. This combines Lemmas 5.1 and 5.2. □

The result of Theorem 5.3 also trivially applies to another natural choice of parameters, the depth of acyclic LCFRS, since they can naturally only take a limited number of derivation steps.

Definition 5.3. A LCFRS is *acyclic* of depth d if d is the smallest integer such that there is a function $\phi : N \to [d]$ such that for all $A \to f(B_1, \ldots, B_n)$ in R and $i \in [n]$ it holds that $\phi(A) < \phi(B_i)$.

Corollary. *The membership problem for acyclic LCFRS where the rank, fan-out, and depth are taken as the parameter is* W[1]*-complete.*

6 Discussion

We have shown that the 1-LCFRS($\mathfrak{P}$)-MEMBERSHIP problem is W[SAT]-hard, but we have no upper bound, except for the trivial XP membership. A conjecture of Pietrzak (2003) may help explain the difficulty of finding such an upper bound. It states that any parameterized problem that has a property that Pietrzak calls *additive* is either in FPT or not in W[P]. Basically, additivity says that any number of instances, sharing a parameter value, can in polynomial time be combined into one big instance, with the same parameter. While 1-LCFRS($\mathfrak{P}$)-MEMBERSHIP is not additive, it has subproblems that are. This means that if Pietrzak's conjecture is true (and FPT $\neq$ W[P]), then 1-LCFRS($\mathfrak{P}$)-MEMBERSHIP cannot belong to W[P].

While our results are mostly intractability results, we see them as a first step towards a more finely grained understanding of the complexity of LCFRS parsing. Ruling out simple parameterization by fan-out or rank as a road towards efficient algorithms lets us focus on other possibilities. Many possible parameterizations remain unexplored. In particular, we conjecture that parameterizing by string length yields FPT membership. In the search for features that can be used in algorithm development, it may also be useful to investigate other formalisms, such as e.g., hypergraph replacement and tree-walking transducers.

Acknowledgments

We acknowledge the support of the Swedish Research Council grant 621-2011-6080.

References

K. A. Abrahamson, R. G. Downey, and M. R. Fellows. 1993. Fixed-parameter intractability II (Extended abstract). In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science (STACS'93)*, pages 374–385.

K. A. Abrahamson, R. G. Downey, and M. R. Fellows. 1995. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE-analogues. *Annals of Pure and Applied Logic*, 73:235–276.

P. Boullier. 2004. Range concatenation grammars. In *New Developments in Parsing Technology*, pages 269–289. Kluwer Academic Publishers.

H. Burden and P. Ljunglöf. 2005. Parsing linear context-free rewriting systems. In *Proceedings of 9th International Workshop on Parsing Technologies*.

R. G. Downey and M. R. Fellows. 1999. *Parameterized Complexity*. Springer-Verlag.

R. G. Downey, M. R. Fellows, B. M. Kapron, M. T. Hallett, and H. T. Wareham. 1994. The parameterized complexity of some problems in logic and linguistics. In *Logical Foundations of Computer Science*, pages 89–100. Springer.

J. Flum and M. Grohe. 2006. *Parameterized Complexity Theory*. Springer-Verlag.

C. Gómez-Rodríguez, M. Kuhlmann, and G. Satta. 2010. Efficient parsing of well-nested linear context-free rewriting systems. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the ACL*, pages 276–284.

A. Joshi. 1985. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions. In *Natural Language Parsing*, pages 206–250. Cambridge University Press.

Y. Kaji, R. Nakanisi, H. Seki, and T. Kasami. 1992. The universal recognition problem for multiple context-free grammars and for linear context-free rewriting systems. *IEICE Transactions on Information and Systems*, E75-D(1):78–88.

K. Pietrzak. 2003. A conjecture on the parameterized hierarchy. Notes on a talk given at Dagstuhl Seminar 03311. Unpublished manuscript.

B. Sagot and G. Satta. 2010. Optimal rank reduction for linear context-free rewriting systems with fan-out two. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL'10)*, pages 525–533.

G. Satta. 1992. Recognition of linear context-free rewriting systems. In *Proceedings of the 30th Annual Meeting of the Association for Computational Linguistics (ACL'92)*, pages 89–95.

G. Satta. 1998. Trading independent for synchronized parallelism in finite copying parallel rewriting systems. *J. Computer and System Sciences*, 56(1):27–45.

H. Seki, T. Matsumura, M. Fujii, and T. Kasami. 1991. On multiple contex-free grammars. *Theoretical Computer Science*, 88(2):191–229.

K. Vijay-Shanker, D. J. Weir, and A. K. Joshi. 1987. Characterizing structural descriptions produced by variaous grammatical formalisms. In *Proceedings of the 25th Meeting of the Association for Computational Linguists (ACL'87)*, pages 104–111.

III

Cuts in Regular Expressions

Martin Berglund[1], Henrik Björklund[1], Frank Drewes[1],
Brink van der Merwe[2], and Bruce Watson[2]

[1] Umeå University, Sweden
{mbe,henrikb,drewes}@cs.umu.se
[2] Stellenbosch University, South Africa
abvdm@cs.sun.ac.za, bruce@fastar.org

Abstract. Most software packages with regular expression matching engines offer operators that extend the classical regular expressions, such as counting, intersection, complementation, and interleaving. Some of the most popular engines, for example those of Java and Perl, also provide operators that are intended to control the nondeterminism inherent in regular expressions. We formalize this notion in the form of the *cut* and *iterated cut* operators. They do not extend the class of languages that can be defined beyond the regular, but they allow for exponentially more succinct representation of some languages. Membership testing remains polynomial, but emptiness testing becomes PSPACE-hard.

1 Introduction

Regular languages are not only a theoretically well-understood class of formal languages. They also appear very frequently in real world programming. In particular, regular expressions are a popular tool for solving text processing problems. For this, the ordinary semantics of regular expressions, according to which an expression simply denotes a language, is extended by an informally defined operational understanding of how a regular expression is "applied" to a string. The usual default in regular expression matching libraries is to search for the leftmost matching substring, and pick the longest such substring [2]. This behavior is often used to repeatedly match different regular expressions against a string (or file contents) using program control flow to decide the next expression to match. Consider the repeatedly matching pseudo-code below, and assume that `match_regex` matches the longest prefix possible:

```
match = match_regex("(a*b)*", s);
if(match != null) then
    if(match_regex("ab*c", match.string_remainder) != null) then
        return match.string_remainder == "";
return false;
```

For the string $s = abac$, this program first matches $R_1 = (a^* \cdot b)^*$ to the substring ab, leaving ac as a remainder, which is matched by $R_2 = a \cdot (b^*) \cdot c$, returning true. The set of strings s for which the program returns "true" is in fact a regular language, but it is *not* the regular language defined by $R_1 \cdot R_2$. Consider for

example the string $s = aababcc$, which is matched by $R_1 \cdot R_2$. However, in an execution of the program above, R_1 will match $aabab$, leaving the remainder cc, which is not matched by R_2. The expression $R_1 \cdot R_2$ exhibits non-deterministic behavior which is lost in the case of the earliest-longest-match strategy combined with the explicit *if*-statement. This raises the question, are programs of this type (with arbitrarily many *if*-statements freely nested) always regular, and how can we describe the languages they recognize?

Related work. Several extensions of regular expressions that are frequently available in software packages, such as counting (or numerical occurrence indicators, not to be confused with counter automata), interleaving, intersection, and complementation, have been investigated from a theoretical point of view. The succinctness of regular expressions that use one or more of these extra operators compared to standard regular expressions and finite automata were investigated, e.g., in [4, 6, 8]. For regular expressions with intersection, the membership problem was studied in, e.g., [10, 14], while the equivalence and emptiness problems were analyzed in [3, 15]. Interleaving was treated in [5, 11] and counting in [9, 12]. To our knowledge, there is no previous theoretical treatment of the cut operator introduced in this paper, or of other versions of possessive quantification.

Paper outline. In the next section we formalize the control of nondeterminism outlined above by defining the cut and iterated cut operators, which can be included directly into regular expressions, yielding so-called cut expressions. In Section 3, we show that adding the new operators does not change the expressive power of regular expressions, but that it does offer improved succinctness. Section 4 provides a polynomial time algorithm for the uniform membership problem of cut expressions, while Section 5 shows that emptiness is PSPACE-hard. In Section 6, we compare the cut operator to the similar operators found more or less commonly in software packages in the wild (Perl, Java, PCRE, etc.). Finally, Section 7 summarizes some open problems.

2 Cut Expressions

We denote the natural numbers (including zero) by $\mathbb{N}$. The set of all strings over an alphabet Σ is denoted by Σ^*. In particular, Σ^* contains the empty string ε. The set $\Sigma^* \setminus \{\varepsilon\}$ is denoted by Σ^+. We write $\mathit{pref}(u)$ to denote the set of nonempty prefixes of a string u and $\mathit{pref}_\varepsilon(u)$ to denote $\mathit{pref}(u) \cup \{\varepsilon\}$. The canonical extensions of a function $f\colon A \to B$ to a function from A^* to B^* and to a function from 2^A to 2^B are denoted by f as well.

As usual, a regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \mid E')$, $(E \cdot E')$, or (E^*). Parentheses can be dropped using the rule that * (Kleene closure[3]) takes precedence over $\cdot$ (concatenation), which takes precedence over $\mid$ (union).

[3] Recall that the Kleene closure of a language L is the smallest language L^* such that $\{\varepsilon\} \cup LL^* \subseteq L^*$.

Moreover, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language $\mathcal{L}(E)$ denoted by a regular expression is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$. We denote by $E \equiv E'$ the fact that two regular expressions (or, later on, cut expressions) E and E' are equivalent, i.e., that $\mathcal{L}(E) = \mathcal{L}(E')$. Where the meaning is clear from context we may omit the $\mathcal{L}$ and write E to mean $\mathcal{L}(E)$.

Let us briefly recall finite automata. A nondeterministic finite automaton (NFA) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ consisting of a finite set Q of states, a initial state $q_0 \in Q$, a set $F \subseteq Q$ of final states, an alphabet Σ, and a transition function $\delta\colon Q \times \Sigma \to 2^Q$. In the usual way, δ extends to a function $\delta\colon \Sigma^* \to 2^Q$, i.e., $\delta(\varepsilon) = \{q_0\}$ and $\delta(wa) = \bigcup_{q \in \delta(w)} \delta(q, a)$. A accepts $w \in \Sigma^*$ if and only if $\delta(w) \cap F \neq \emptyset$, and it recognizes the language $\mathcal{L}(A) = \{w \in \Sigma^* \mid \delta(w) \cap F \neq \emptyset\}$. A deterministic finite automaton (DFA) is the special case where $|\delta(q, a)| \leq 1$ for all $(q, a) \in Q \times \Sigma$. In this case we consider δ to be a function $\delta\colon Q \times \Sigma \to Q$, so that its canonical extension to strings becomes a function $\delta\colon Q \times \Sigma^* \to Q$.

We now introduce cuts, iterated cuts, and cut expressions. Intuitively, $E \,!\, E'$ is the variant of EE' in which E greedily matches as much of a string as it can accommodate, leaving the rest to be matched by E'. The so-called iterated cut $E^{!*}$ first lets E match as much of a string as possible, and seeks to iterate this until the whole string is matched (if possible).

Definition 1 (cut and cut expression). *The* cut *is the binary operation* ! *on languages such that, for languages* L, L'*,*

$$L \,!\, L' = \{uv \mid u \in L, v \in L', uv' \notin L \text{ for all } v' \in \mathit{pref}(v)\}.$$

The iterated cut of L*, denoted by* $L^{!*}$*, is the smallest language that satisfies*

$$\{\varepsilon\} \cup (L \,!\, (L^{!*})) \subseteq L^{!*}$$

(i.e., $L \,!\, (L \,!\, \cdots (L \,!\, (L \,!\, \{\varepsilon\})) \cdots) \subseteq L^{!*}$ *for any number of repetitions of the cut).*

Cut expressions *are expressions built using the operators allowed in regular expressions, the cut, and the iterated cut. A cut expression denotes the language obtained by evaluating that expression in the usual manner.*

The precedence rules give $^{!*}$ precedence over $\cdot$, which in turn gets precedence over ! which in turn gets precedence over $|$.

The motivation for the inclusion of the iterated cut is two-fold; (i) it is a natural extension for completeness in that it relates to the cut like the Kleene closure relates to concatenation; and, (ii) in the context of a program like that shown on page 1, the iterated cut permits the modelling of matching regular expressions in loops.

Let us discuss a few examples.

1. The cut expression $ab^* \,!\, b$ yields the empty language. This is because every string in $\mathcal{L}(ab^*b)$ is in $\mathcal{L}(ab^*)$ as well, meaning that the greedy matching of the first subexpression will never leave a b over for the second. Looking at

the definition of the cut, a string in $\mathcal{L}(ab^* \,!\, b)$ would have to be of the form ub, such that $u \in \mathcal{L}(ab^*)$ but $ub \notin \mathcal{L}(ab^*)$. Clearly, such a string does not exist. More generally, if $\varepsilon \notin \mathcal{L}(E')$ then $\mathcal{L}(E \,!\, E') \subseteq \mathcal{L}(EE') \setminus \mathcal{L}(E)$. However, as the next example shows, the converse inclusion does not hold.

2. We have $(a^* \,|\, b^*) \,!\, (ac \,|\, bc) \equiv a^+bc \,|\, b^+ac$.[4] This illustrates that the semantics of the cut cannot be expressed by concatenating subsets of the involved languages. In the example, there are no subsets L_1 and L_2 of $\mathcal{L}(a^* \,|\, b^*)$ and $\mathcal{L}(ac \,|\, bc)$, respectively, such that $L_1 \cdot L_2 = \mathcal{L}(a^* \,|\, b^*) \,!\, \mathcal{L}(ac \,|\, bc)$.
3. Clearly, $((ab)^* \,!\, a) \,!\, b \equiv (ab)^*ab$ whereas $(ab)^* \,!\, (a \,!\, b) \equiv (ab)^* \,!\, ab \equiv \emptyset$ (as in the first example). Thus, the cut is not associative.
4. As an example of an iterated cut, consider $((aa)^* \,!\, a)^*$. We have $(aa)^* \,!\, a \equiv (aa)^*a$ and therefore $((aa)^* \,!\, a)^* \equiv a^*$. This illustrates that matching a string against $(E \,!\, E')^*$ *cannot* be done by greedily matching E, then matching E', and iterating this procedure. Instead, one has to "chop" the string to be matched into substrings and match each of those against $E \,!\, E'$. In particular, $(E \,!\, \varepsilon)^* \equiv E^*$ (since $E \,!\, \varepsilon \equiv E$). This shows that $E^{!*}$ cannot easily be expressed by means of cut and Kleene closure.
5. Let us finally consider the interaction between the Kleene closure and the iterated cut. We have $L^{!*} \subseteq L^*$ and thus $(L^{!*})^* \subseteq (L^*)^* = L^*$. Conversely, $L \subseteq L^{!*}$ yields $L^* \subseteq (L^{!*})^*$. Thus $(L^{!*})^* = L^*$ for all languages L.
   Similarly, we also have $(L^*)^{!*} = L^*$. Indeed, if $w \in L^*$, then it belongs to $(L^*)^{!*}$, since the first iteration of the iterated cut can consume all of w. Conversely, $(L^*)^{!*} \subseteq (L^*)^* = L^*$. Thus, altogether $(L^*)^{!*} = L^* = (L^{!*})^*$

3 Cut Expressions Versus Finite Automata

In this section, we compare cut expressions and finite automata. First, we show that the languages described by cut expressions are indeed regular. We do this by showing how to convert cut expressions into equivalent finite automata. Second, we show that cut expressions are succinct: There are cut expressions containing only a single cut (and no iterated cut), such that a minimal equivalent NFA or regular expression is of exponential size.

3.1 Cut Expressions Denote Regular Languages

Let A, A' be DFAs. To prove that the languages denoted by cut expressions are regular, it suffices to show how to construct DFAs recognizing $L(A) \,!\, L(A')$ and $L(A)^{!*}$. We note here that an alternative proof would be obtained by showing how to construct alternating automata (AFAs) recognizing $L(A) \,!\, L(A')$ and $L(A)^{!*}$. Such a construction would be slightly simpler, especially for the iterated cut, but since the conversion of AFAs to DFAs causes a doubly exponential size increase [1], we prefer the construction given below, which (almost) saves one level of exponentiality. Moreover, we hope that this construction, though more complex, is more instructive.

[4] As usual, we abbreviate EE^* by E^+.

We first handle the comparatively simple case $L(A)\,!\,L(A')$. The idea of the construction is to combine A with a kind of product automaton of A and A'. The automaton starts working like A. At the point where A reaches one of its final states, A' starts running in parallel with A. However, in contrast to the ordinary product automaton, the computation of A' is reset to its initial state whenever A reaches one of its final states again. Finally, the string is accepted if and only if A' is in one of its final states.

To make the construction precise, let $A = (Q, \Sigma, \delta, q_0, F)$ and $A' = (Q', \Sigma, \delta', q_0', F')$. In order to disregard a special case, let us assume that $q_0 \notin F$. (The case where $q_0 \in F$ is easier, because it allows us to use only product states in the automaton constructed.) We define a DFA $\overline{A} = (\overline{Q}, \Sigma, \overline{\delta}, q_0, \overline{F})$ as follows:

- $\overline{Q} = Q \cup (Q \times Q')$ and $\overline{F} = Q \times F'$,
- for all $q, r \in Q$, $\overline{q} = (q, q') \in \overline{Q}$, and $a \in \Sigma$ with $\delta(q, a) = r$

$$\overline{\delta}(q,a) = \begin{cases} r & \text{if } r \notin F \\ (r, q_0') & \text{otherwise,} \end{cases} \quad \text{and} \quad \overline{\delta}(\overline{q},a) = \begin{cases} (r, \delta'(q', a)) & \text{if } r \notin F \\ (r, q_0') & \text{otherwise.} \end{cases}$$

Let $w \in \Sigma^*$. By construction, $\overline{\delta}$ has the following properties:

1. If $u \notin L(A)$ for all $u \in \mathit{pref}_\varepsilon(w)$, then $\overline{\delta}(w) = \delta(w)$.
2. Otherwise, let $w = uv$, where u is the longest prefix of w such that $u \in L(A)$. Then $\overline{\delta}(w) = (\delta(w), \delta'(v))$.

We omit the easy inductive proof of these statements. By the definition of $L(A)\,!\,L(A')$ and the choice of $\overline{F}$, they imply that $L(\overline{A}) = L(A)\,!\,L(A')$. In other words, we have the following lemma.

Lemma 2. *For all regular languages L and L', the language $L\,!\,L'$ is regular.*

Let us now consider the iterated cut. Intuitively, the construction of a DFA recognizing $L(A)^{!*}$ is based on the same idea as above, except that the product construction is iterated. The difficulty is that the straightforward execution of this construction yields an infinite automaton. For the purpose of exposing the idea, let us disregard this difficulty for the moment. Without loss of generality, we assume that $q_0 \notin F$ (which we can do because $L(A)^{!*} = (L(A) \setminus \{\varepsilon\})^{!*}$) and that $\delta(q, a) \neq q_0$ for all $q \in Q$ and $a \in \Sigma$.

We construct an automaton whose states are strings $q_1 \cdots q_k \in Q^+$. The automaton starts in state q_0, initially behaving like A. If it reaches one of the final states of A, say q_1, it continues in state $q_1 q_0$, working essentially like the automaton for $L(A)\,!\,L(A)$. In particular, it "resets" the second copy each time the first copy encounters a final state of A. However, should the second copy reach a final state q_2 of A (while $q_1 \notin F$), a third copy is spawned, thus resulting in a state of the form $q_1 q_2 q_0$, and so on.

Formally, let $\delta_a \colon Q \to Q$ be given by $\delta_a(q) = \delta(q, a)$ for all $a \in \Sigma$ and $q \in Q$. Recall that functions to extend to sequences, so $\delta_a : Q^* \to Q^*$ operates elementwise. We construct the (infinite) automaton $\widehat{A} = (\widehat{Q}, \Sigma, \widehat{\delta}, q_0, \widehat{F})$ as follows:

– $\widehat{Q} = (Q \setminus \{q_0\})^* Q$.
– For all $s = q_1 \cdots q_k \in \widehat{Q}$ and $a \in \Sigma$ with $\delta_a(s) = q'_1 \cdots q'_k$

$$\widehat{\delta}(s,a) = \begin{cases} q'_1 \cdots q'_k & \text{if } q'_1, \ldots, q'_k \notin F \\ q'_1 \cdots q'_l q_0 & \text{if } l = \min\{i \in \{1,\ldots,k\} \mid q'_i \in F\}. \end{cases} \tag{1}$$

– $\widehat{F} = \{q_1 \cdots q_k \in \widehat{Q} \mid q_k = q_0\}$.

Note that $\widehat{\delta}(s,a) \in \widehat{Q}$ since we assume that $\delta(q,a) \neq q_0$ for all $q \in Q$ and $a \in \Sigma$.

Similar to the properties of $\overline{A}$ above, we have the following:
Claim 1. Let $w = v_1 \cdots v_k \in \Sigma^*$, where $v_1 \cdots v_k$ is the unique decomposition of w such that (a) for all $i \in \{1,\ldots,k-1\}$, v_i is the longest prefix of $v_i \cdots v_k$ which is in $L(A)$ and (b) $pref_\varepsilon(v_k) \cap L(A) = \emptyset$.[5] Then $\widehat{\delta}(w) = \delta(v_1 \cdots v_k)\delta(v_2 \cdots v_k) \cdots \delta(v_k)$. In particular, $\widehat{A}$ accepts w if and only if $w \in L(A)^{!*}$.

Again, we omit the straightforward inductive proof.

It remains to be shown how to turn the set of states of $\widehat{A}$ into a finite set. We do this by verifying that repetitions of states of A can be deleted. To be precise, let $\pi(s)$ be defined as follows for all $s = q_1 \cdots q_k \in \widehat{Q}$. If $k = 1$ then $\pi(s) = s$. If $k > 1$ then

$$\pi(s) = \begin{cases} \pi(q_1 \cdots q_{k-1}) & \text{if } q_k \in \{q_1, \ldots, q_{k-1}\} \\ \pi(q_1 \cdots q_{k-1}) q_k & \text{otherwise.} \end{cases}$$

Let $\pi(\widehat{A})$ be the NFA obtained from $\widehat{A}$ by taking the quotient with respect to π, i.e., by identifying all states $s, s' \in \widehat{Q}$ such that $\pi(s) = \pi(s')$. The set of final states of $\pi(\widehat{A})$ is the set $\pi(\widehat{F})$.

This completes the construction. The following lemmas prove its correctness.

Lemma 3. *For all $s \in \widehat{Q}$ and $a \in \Sigma$ it holds that $\pi(\widehat{\delta}(s,a)) = \pi(\widehat{\delta}(\pi(s),a))$.*

Proof. By the very definition of π, for every function $f\colon Q \to Q$ and all $s \in \widehat{Q}$ we have $\pi(f(s)) = \pi(f(\pi(s)))$. In particular, this holds for $f = \delta_a$. Now, let $s = q_1 \cdots q_k$ be as in the definition of $\widehat{\delta}$. Since the same set of symbols occurs in $\delta_a(s)$ and $\delta_a(\pi(s))$, the same case of Equation 1 applies for the construction of $\widehat{\delta}(s,a)$ and $\widehat{\delta}(\pi(s),a)$. In the first case $\pi(\widehat{\delta}(s,a)) = \pi(\delta_a(s)) = \pi(\delta_a(\pi(s))) = \pi(\widehat{\delta}(\pi(s),a))$. In the second case

$$\begin{aligned} \pi(\widehat{\delta}(s,a)) &= \pi(\delta_a(q_1 \cdots q_l) q_0) \\ &= \pi(\delta_a(q_1 \cdots q_l)) q_0 \\ &= \pi(\delta_a(\pi(q_1 \cdots q_l))) q_0 \\ &= \pi(\delta_a(\pi(q_1 \cdots q_l)) q_0) \\ &= \pi(\widehat{\delta}(\pi(s),a)). \end{aligned}$$

Note that the second and the fourth equality make use of the fact that $q_0 \notin \{q_1, \ldots, q_{k-1}\}$, which prevents π from deleting the trailing q_0. □

[5] The strings $v_1, \ldots, v_k$ are well defined because $\varepsilon \notin L(A)$.

Lemma 4. *The automaton $\pi(\widehat{A})$ is a DFA such that $L(\pi(\widehat{A})) = L(A)^{!*}$. In particular, $L^{!*}$ is regular for all regular languages L.*

Proof. To see that $\pi(\widehat{A})$ is a DFA, let $a \in \Sigma$. By the definition of $\pi(\widehat{A})$, its transition function $\widehat{\delta}_\pi$ is given by

$$\widehat{\delta}_\pi(t, a) = \{\pi(\widehat{\delta}(s, a)) \mid s \in \widehat{Q},\ t = \pi(s)\}$$

for all $t \in \pi(\widehat{Q})$. However, by Lemma 3, $\pi(\widehat{\delta}(s, a)) = \pi(\widehat{\delta}(t, a))$ is independent of the choice of s. In other words, $\widehat{A}$ is a DFA. Furthermore, by induction on the length of $w \in \Sigma^*$, Lemma 3 yields $\widehat{\delta}_\pi(w) = \pi(\widehat{\delta}(w))$. Thus, by Claim 1, $L(\pi(\widehat{A})) = L(A)^{!*}$. In particular, for a regular language L, this shows that $L^{!*}$ is regular, by picking A such that $\mathcal{L}(A) = L$. □

We note here that, despite the detour via an infinite automaton, the construction given above can effectively be implemented. Unfortunately, it results in a DFA of size $\mathcal{O}(n!)$, where n is the number of states of the original DFA.

Theorem 5. *For every cut expression E, $\mathcal{L}(E)$ is regular.*

Proof. Follows from combining Lemmas 2 and 4. □

3.2 Succinctness of Cut Expressions

In this section we show that for some languages, cut expressions provide an exponentially more compact representation than regular expressions and NFAs.

Theorem 6. *For every $k \in \mathbb{N}_+$, there exists a cut expression E_k of size $\mathcal{O}(k)$ such that every NFA and every regular expression for $\mathcal{L}(E_k)$ is of size $2^{\Omega(k)}$. Furthermore, E_k does not contain the iterated cut and it contains only one occurrence of the cut.*

Proof. We use the alphabets $\Sigma = \{0, 1\}$ and $\Gamma = \Sigma \cup \{], [\}$. For $k \in \mathbb{N}_+$, let

$$E_k = (\varepsilon \mid [\Sigma^* 0 \Sigma^{k-1} 1 \Sigma^*] \mid [\Sigma^* 1 \Sigma^{k-1} 0 \Sigma^*]) \,!\, [\Sigma^{2k}].$$

Each string in the language $\mathcal{L}(E_k)$ consists of one or two bitstrings enclosed in square brackets. If there are two, the first has at least two different bits at a distance of exactly k positions and the second is an arbitrary string in Σ^{2k}. However, when there is only a single pair of brackets the bitstring enclosed is of length $2k$ and its second half will be an exact copy of the first.

We argue that any NFA that recognizes $\mathcal{L}(E_k)$ must have at least 2^k states. Assume, towards a contradiction, that there is an NFA A with fewer than 2^k states that recognizes $\mathcal{L}(E_k)$.

Since $|\Sigma^k| = 2^k$ there must exist two distinct bitstrings w_1 and w_2 of length k such that the following holds. There exist a state q of A and accepting runs ρ_1 and ρ_2 of A on $[w_1 w_1]$ and $[w_2 w_2]$, resp., such that ρ_1 reaches q after reading $[w_1$ and ρ_2 reaches q after reading $[w_2$. This, in turn, means that there are accepting

runs ρ'_1 and ρ'_2 of A_q on $w_1]$ and $w_2]$, respectively, where A_q is the automaton obtained from A by making q the sole initial state. Combining the first half of ρ_1 with ρ'_2 gives an accepting run of A on $[w_1 w_2]$. This is a contradiction and we conclude that there is no NFA for E_k with fewer than 2^k states.

The above conclusion also implies that every regular expression for $\mathcal{L}(E_k)$ has size $2^{\Omega(k)}$. If there was a smaller regular expression, the Glushkov construction [7] would also yield a smaller NFA. □

Remark 7. The only current upper bound is the one implied by Section 3.1, from which automata of non-elementary size cannot be ruled out as it yields automata whose sizes are bounded by powers of twos.

A natural restriction on cut expressions is to only allow cuts to occur at the topmost level of the expression. This gives a tight bound on automata size.

Lemma 8. *Let E be a cut expression, without iterated cuts, such that no subexpression of the form C^* or $C \cdot C'$ contains cuts. Then the minimal equivalent DFA has $2^{\mathcal{O}(|E|)}$ states, and this bound is tight.*

Proof (sketch). Given any DFAs A, A', using product constructions we get DFAs for $L(A) \mid L(A')$ and $L(A) ! L(A')$ whose number of states is proportional to the product of the number of states in A and A'. (See Lemma 2 for the case $L(A) ! L(A')$.) Thus, one can construct an exponential-sized DFA in a bottom-up manner. Theorem 6 shows that this bound is tight. □

4 Uniform Membership Testing

We now present an easy membership test for cut expressions that uses a dynamic programming approach (or, equivalently, memoization). Similarly to the Cocke-Younger-Kasami algorithm, the idea is to check which substrings of the input string belong to the languages denoted by the subexpressions of the given cut expression. The pseudocode of the algorithm is shown in Algorithm 1. Here, the string $u = a_1 \cdots a_n$ to be matched against a cut expression E is a global variable. For $1 \leq i \leq j \leq n+1$, $\mathit{Match}(E, i, j)$ will check whether $a_i \cdots a_{j-1} \in \mathcal{L}(E)$. We assume that an implicit table is used in order to memoize computed values for a given input triple. Thus, recursive calls with argument triples that have been encountered before will immediately return the memoized value rather than executing the body of the algorithm.

Theorem 9. *The uniform membership problem for cut expressions can be decided in time $\mathcal{O}(m \cdot n^3)$, where m is the size of the cut expression and n is the length of the input string.*

Proof. Consider a cut expression E_0 of size m and a string $u = a_1 \cdots a_n$. It is straightforward to show by induction on $m+n$ that $\mathit{Match}(E, i, j) = \mathit{true}$ if and only if $a_i \cdots a_{j-1} \in \mathcal{L}(E)$, where $1 \leq i \leq j \leq n+1$ and E is a subexpression of E_0. For $E = E_1 ! E_2$, this is because of the fact that $v \in \mathcal{L}(E)$ if and only if v

Algorithm 1 $Match(E, i, j)$

if $E = \emptyset$ **then return** *false*
else if $E = \varepsilon$ **then return** $i = j$
else if $E \in \Sigma$ **then return** $j = i + 1 \wedge E = a_i$
else if $E = E_1 \mid E_2$ **then return** $Match(E_1, i, j) \vee Match(E_2, i, j)$
else if $E = E_1 \cdot E_2$ **then**
  for $k = 0, \ldots, j - i$ **do**
    if $Match(E_1, i, i + k) \wedge Match(E_2, i + k, j)$ **then return** *true*
  return *false*
else if $E = E_1^*$ **then**
  for $k = 1, \ldots, j - i$ **do**
    if $Match(E_1, i, i + k) \wedge Match(E, i + k, j)$ **then return** *true*
  return $i = j$
else if $E = E_1 \,!\, E_2$ **then**
  for $k = j - i, \ldots, 0$ **do**
    if $Match(E_1, i, i + k)$ **then return** $Match(E_2, i + k, j)$
  return *false*
else if $E = E_1^{!*}$ **then**
  for $k = j - i, \ldots, 1$ **do**
    if $Match(E_1, i, i + k)$ **then return** $Match(E, i + k, j)$
  return $i = j$

has a longest prefix $v_1 \in \mathcal{L}(E_1)$, and the corresponding suffix v_2 of v (i.e., such that $v = v_1 v_2$) is in $\mathcal{L}(E_2)$. Furthermore, it follows from this and the definition of the iterated cut that, for $E = E_1^{!*}$, $v \in \mathcal{L}(E)$ if either $v = \varepsilon$ or v has a longest prefix $v_1 \in \mathcal{L}(E_1)$ such that the corresponding suffix v_2 is in $\mathcal{L}(E)$.

Regarding the running time of $Match(E, 1, n+1)$, by memoization the body of *Match* is executed at most once for every subexpression of E and all i, j, $1 \leq i \leq j \leq n + 1$. This yields $\mathcal{O}(m \cdot n^2)$ executions of the loop body. Moreover, a single execution of the loop body involves at most $\mathcal{O}(n)$ steps (counting each recursive call as one step), namely if $E = E_1^*$, $E = E_1 \,!\, E_2$ or $E = E^{!*}$. □

5 Emptiness Testing of Cut Expressions

Theorem 10. *Given a cut expression E, it is PSPACE-hard to decide whether $\mathcal{L}(E) = \emptyset$. This remains true if $E = E_1 \,!\, E_2$, where E_1 and E_2 are regular expressions.*

Proof. We prove the theorem by reduction from regular expression universality, i.e. deciding for a regular expression R and an alphabet Σ whether $\mathcal{L}(R) = \Sigma^*$. This problem is well known to be PSPACE-complete [12].Given R, we construct a cut expression E such that $\mathcal{L}(E) = \emptyset$ if and only if $\mathcal{L}(R) = \Sigma^*$.

We begin by testing if $\varepsilon \in \mathcal{L}(R)$. This can be done in polynomial time. If $\varepsilon \notin \mathcal{L}(R)$, then we set $E = \varepsilon$, satisfying $\mathcal{L}(E) \neq \emptyset$. Otherwise, we set $E = R \,!\, \Sigma$. If R is universal, there is no string ua such that $u \in \mathcal{L}(R)$ but $ua \notin \mathcal{L}(R)$. Thus $\mathcal{L}(E)$ is empty. If R is not universal, since $\varepsilon \in \mathcal{L}(R)$ there are $u \in \Sigma^*$ and $a \in \Sigma$ such that $u \in \mathcal{L}(R)$ and $ua \notin \mathcal{L}(R)$, which means that $ua \in \mathcal{L}(E) \neq \emptyset$. □

Lemma 11. *For cut expressions E the problems whether $\mathcal{L}(E) = \emptyset$ and $\mathcal{L}(E) = \Sigma^*$ are LOGSPACE-equivalent.*

Proof. Assume that $\# \notin \Sigma$, and let $\Sigma' = \Sigma \cup \{\#\}$. The lemma then follows from these two equivalences: (i) $E \equiv \emptyset$ if and only if $((\varepsilon \mid E\Sigma^*) \,!\, \Sigma^+) \mid \varepsilon \equiv \Sigma^*$; and; (ii) $E \equiv \Sigma^*$ if and only if $(\varepsilon \mid E\#(\Sigma')^*) \,!\, \Sigma^*\# \equiv \emptyset$. □

6 Related Concepts in Programming Languages

Modern regular expression matching engines have numerous highly useful features, some of which improve succinctness (short-hand operators) and some of which enable expressions that specify non-regular languages. Of interest here is that most regular expression engines in practical use feature at least *some* operation intended to control nondeterminism in a way that resembles the cut. They are however only loosely specified in terms of *backtracking*, the specific evaluation technique used by many regular expression engines. This, combined with the highly complex code involved, makes formal analysis difficult.

All these operations appear to trace their ancestry to the first edition of "Mastering Regular Expressions" [2], which contains the following statement:

> *"A feature I think would be useful, but that no regex flavor that I know of has, is what I would call possessive quantifiers. They would act like normal quantifiers except that once they made a decision that met with local success, they would never backtrack to try the other option. The text they match could be unmatched if their enclosing subexpression was unmatched, but they would never give up matched text of their own volition, even in deference to the overall match."[2]*

The cut operator certainly fits this somewhat imprecise description, but as we shall see implementations have favored different interpretations. Next we give a brief overview of three different operations implemented in several major regular expression engines, that exhibit some control over nondeterminism. All of these operators are of great practical value and are in use. Still, they feature some idiosyncrasies that should be investigated, in the interest of bringing proper regular behavior to as large a set of regular expression functionality as possible.

Possessive Quantifiers Not long after the proposal for the possessive quantifier, implementations started showing up. It is available in software such as Java, PCRE, Perl, etc. For a regular expression R the operation is denoted R^{*+}, and behaves like R^* except it never backtracks. This is already troublesome, since "backtracking" is poorly defined at best, and, in fact, by itself $\mathcal{L}(R^{*+}) = \mathcal{L}(R^*)$, but $\mathcal{L}(R^{*+} \cdot R') = \mathcal{L}(R^* \,!\, R')$ for all R'. That is, extending regular expressions with possessive quantifiers makes it possible to write expressions such that $\mathcal{L}(E \cdot E') \neq \mathcal{L}(E) \cdot \mathcal{L}(E')$, an example being given by $E = a^{*+}$ and $E' = a$. This violates the compositional spirit of regular expressions.

Next, consider Table 1. The expression on the first row, call it R, is tested in each of the given implementations, and the language recognized is shown. The

Table 1. Some examples of possessive quantifier use.

Expression	Perl 5.16.2	Java 1.6.0u18	PCRE 8.32
$(aa)^{*+}a$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^{*+}a)^*$	$\{\varepsilon, a, aaa, aaaaa, \ldots\}$	$\{\varepsilon, a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^{*+}a)^*a$	$\{a\}$	$\{a\}$	$\{a\}$

Table 2. Comparison between Perl and PCRE when using the (*PRUNE) operator.

Expression	Perl 5.10.1	Perl 5.16.2	PCRE 8.32
$(aa)^*$(*PRUNE)a	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$	$\{a, aaa, aaaaa, \ldots\}$
$((aa)^*$(*PRUNE)$a)^*$	$\{\varepsilon, a, aa, aaa, \ldots\}$	$\emptyset$	$\emptyset$
a^*(*PRUNE)a	$\{a, aa, aaa, \ldots\}$	$\emptyset$	$\emptyset$

results on the first row are easy to accept from every perspective. The second row however has the expression R^*, and despite $a \in \mathcal{L}(R)$ no implementation gives $aa \in \mathcal{L}(R^*)$, which violates the classical compositional meaning of the Kleene closure (in addition, in PCRE we have $\varepsilon \notin \mathcal{L}(R^*)$). The third row further illustrates how the compositional view of regular expressions breaks down when using possessive quantifiers.

Independent Groups or Atomic Subgroups A practical shortcoming of the possessive quantifiers is that the "cut"-like operation cannot be separated from the quantifier. For this reason most modern regular expression engines have also introduced atomic subgroups ("independent groups" in Java). An atomic subgroup containing the expression R is denoted $(?\rangle R)$, and described as "preventing backtracking". Any subexpression $(?\rangle R^*)$ is equivalent to R^{*+}, but subexpressions of the form $(?\rangle R)$ where the topmost operation in R is not a Kleene closure may be hard to translate into an equivalent expression using possessive quantifiers.

Due to the direct translation, atomic subgroups suffer from all the same idiosyncrasies as possessive quantifiers, such as $\mathcal{L}(((?\rangle(aa)^*)a)^*a) = \{a\}$.

*Commit Operators and (*PRUNE)* In Perl 6 several interesting "commit operators" relating to nondeterminism control were introduced. As Perl 5 remains popular they were back-ported to Perl 5 in version 5.10.0 with different syntax. The one closest to the pure cut is (*PRUNE), called a "zero-width pattern", an expression that matches ε (and therefore always succeeds) but has some engine side-effect. As with the previous operators the documentation depends on the internals of the implementation. *"[(*PRUNE)] prunes the backtracking tree at the current point when backtracked into on failure"*[13].

These operations are available both in Perl and PCRE, but interestingly their semantics in Perl 5.10 and Perl 5.16 differ in subtle ways; see Table 2. Looking at the first two rows we see that Perl 5.10 matches our compositional understanding of the Kleene closure (i.e., row two has the same behavior as $((aa)*!a)*$). On the other hand Perl 5.10 appears to give the wrong answer in the third row example.

7 Discussion

We have introduced cut operators and demonstrated several of their properties. Many open questions and details remain to be worked out however:

- There is a great distance between the upper and lower bounds on minimal automata size presented in Section 3.2, with an exponential lower bound for both DFA and NFA, and a non-elementary upper bound in general.
- The complexity of uniform membership testing can probably be improved as the approach followed by Algorithm 1 is very general. (It can do complementation, for example.)
- The precise semantics of the operators discussed in Section 6 should be studied further, to ensure that all interesting properties can be captured.

Acknowledgments. We thank Yves Orton who provided valuable information about the implementation and semantics of (*PRUNE) in Perl.

References

[1] A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, January 1981.
[2] Jeffrey E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, California, January 1997.
[3] M. Fürer. The complexity of the inequivalence problem for regular expressions with intersection. In *ICALP*, pages 234–245, 1980.
[4] W. Gelade. Succinctness of regular expressions with interleaving, intersection and counting. *Theor. Comput. Sci.*, 411(31-33):2987–2998, 2011.
[5] W. Gelade, W. Martens, and F. Neven. Optimizing schema languages for XML: Numerical constraints and interleaving. *SIAM J. Comput.*, 38(5):2021–2043, 2009.
[6] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. *ACM Trans. Comput. Logic*, 13(1), 2012. Article 4.
[7] V. M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.
[8] H. Gruber and M. Holzer. Tight bounds on the descriptional complexity of regular expressions. In *DLT*, pages 276–287, 2009.
[9] P. Kilpeläinen and R. Tuhkanen. Regular expressions with numerical occurrence indicators - preliminary results. In *SPLST*, pages 163–173, 2003.
[10] O. Kupferman and S. Zuhovitzky. An improved algorithm for the membership problem for extended regular expressions. In *MFCS*, pages 446–458, 2002.
[11] A. J. Mayer and L. J. Stockmeyer. Word problems - this time with interleaving. *Inform. and Comput.*, 115(2):293–311, 1994.
[12] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *SWAT (FOCS)*, pages 125–129, 1972.
[13] Perl 5 Porters. perlre, 2012. http://perldoc.perl.org/perlre.html, accessed January 16th, 2013.
[14] H. Petersen. The membership problem for regular expressions with intersection is complete in LOGCFL. In *STACS*, pages 513–522, 2002.
[15] J. M. Robson. The emptiness of complement problem for semi extended regular expressions requires c^n space. *Inform. Processing Letters*, 9(5):220–222, 1979.

IV

Analyzing Catastrophic Backtracking Behavior in Practical Regular Expression Matching

Martin Berglund
Department of Computing Science,
Umeå University,
Umeå, Sweden
mbe@cs.umu.se

Frank Drewes
Department of Computing Science,
Umeå University,
Umeå, Sweden
drewes@cs.umu.se

Brink van der Merwe
Department of Mathematical Sciences,
Computer Science Division,
University of Stellenbosch,
Stellenbosch, South Africa
abvdm@cs.sun.ac.za

We consider in some detail how regular expression matching happens in Java[1], as a popular representative of the category of regex-directed matching engines. We extract a slightly idealized algorithm for this scenario. Next we define an automata model which captures all the aspects needed to perform matching, of the Java style, in a formal way. Finally, two types of static analysis, which take a regular expression and tells whether there exists a family of strings which make Java-style matching run in exponential time, are done.

1 Introduction

Regular expressions constitute a concise, powerful, and useful pattern matching language for strings. They are commonly used to specify token lexemes for scanner generation during compiler construction, to validate input for web-based applications, to recognize meaningful patterns in natural language processing and data mining, for example, locating e-mail addresses, and to guard against computer system intrusion. Libraries for their use are found in most widely-used programming languages.

There are two fundamentally different types of regex matching engines: DFA (Deterministic Finite Automaton) and NFA (Non-deterministic Finite Automaton) matching engines. DFA matchers are used in (most versions of) awk, egrep, and in MySQL, and are based on the NFA to DFA subset conversion algorithm. This paper deals with NFA engines, which is found in GNU Emacs, Java, many command line tools, .NET, the PCRE (Perl compatible regular expressions) library, Perl, PHP, Python, Ruby and Vim. NFA matchers make use of an input directed depth first search on an NFA, and thus the matching performed by NFA engines are referred to as backtracking matching. NFA engines have made it possible to extend regular expressions with captures, possessive quantifiers, and backreferences.

Theory has however not kept pace with practice when it comes to understanding NFA engines. We now have NFA matchers that are more expressive and succinct than the originally developed DFA matchers, but are also in some cases significantly slower. Although it is known that in the worst case, the matching time of NFA matchers are exponential in the length of input strings [7], their performance characteristics and operational matching semantics are poorly understood in general. Exponential matching time, also referred to as catastrophic backtracking (by NFA matchers), can of course be avoided by using the DFA matchers, but then a less expressive pattern matching language has to be used. Catastrophic backtracking has potentially severe security implications, as denial-of-service attacks are possible in any application which matches a regular expression to data not carefully controlled by the application.

This work was motivated by the exponential time algorithm presented by Kirrage et. al. in [7], which for regular expressions with catastrophic backtracking comes up with a family of strings exhibiting this

[1] Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

exponential matching time behavior. However, they only consider the case where the exponential matching behavior can be exhibited by strings that are rejected. We investigate the complexity of deciding exponential backtracking matching on strings that are rejected (which we refer to as deciding exponential failure backtracking) further, and in addition we consider exponential backtracking in general.

To formally capture backtracking matching, we introduce prioritized NFA (pNFA). These automata make non-deterministic choices ordered, prioritizing some over others, in a way very reminiscent of parsing expression grammars (PEGs), which introduce ordered choice to context-free grammars [5]. An interesting algorithm bridging the two areas is given in Medeiros et. al. in [8], this algorithm translates extended regular expressions to PEGs.

We then define both general backtracking and failure backtracking. The latter is concerned with only failed matches. By linking failure backtracking with ambiguity in NFA, we show that catastrophic failure backtracking can be decided in polynomial time, and in the case of polynomial failure backtracking, the degree of the polynomial can be determined in polynomial time. General backtracking is shown decidable in EXPTIME by associating a tree transducer with the expression and applying a result from [4].

2 Preliminaries

For a set A, we denote by $\mathscr{P}(A)$ the power set of A. The constant function $f\colon A \to B$ with $f(a) = b \in B$ for all $a \in A$ is denoted by b^A. Also, given any function $f\colon A \to B$ and elements $a \in A$, $b \in B$, we let $f_{a\mapsto b}$ denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ for all $x \in A \setminus \{a\}$. For an alphabet Σ, we denote the set of all strings over Σ by Σ^*. In particular, it contains the empty string ε. To avoid confusion, it is assumed that $\varepsilon \notin \Sigma$. The length of a string w is denoted by $|w|$, and the number of occurrences of a symbol a in w is denoted by $|w|_a$. The union of disjoint sets A and B is denoted by $A \uplus B$.

As usual, a regular expression over an alphabet Σ (where $\varepsilon, \emptyset \notin \Sigma$) is either an element of $\Sigma \cup \{\varepsilon, \emptyset\}$ or an expression of one of the forms $(E \,|\, E')$, $(E \cdot E')$, or (E^*), where E and E' are regular expressions. Parentheses can be dropped using the rule that * (Kleene closure) takes precedence over $\cdot$ (concatenation), which takes precedence over $|$ (union). Moreover, outermost parentheses can be dropped, and $E \cdot E'$ can be written as EE'. The language $\mathscr{L}(E)$ denoted by a regular expression is obtained by evaluating E as usual, where $\emptyset$ stands for the empty language and $a \in \Sigma \cup \{\varepsilon\}$ for $\{a\}$.

A *tree* with labels in a set Σ is a function $t\colon V \to \Sigma$, where $V \subseteq \mathbb{N}_+^*$ is a non-empty set of vertices (or nodes) which are such that (i) V is prefix-closed, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $vi \in V$ implies $v \in V$; and; (ii) V is closed to the left, i.e., for all $v \in \mathbb{N}_+^*$ and $i \in \mathbb{N}_+$, $v(i+1) \in V$ implies $vi \in V$.

The vertex ε is the root of the tree and vertex vi is the ith child of v. We let $|t| = |V|$ denote the size of t. t/v denotes the tree t' with vertex set $V' = \{w \in \mathbb{N}_+^* \mid vw \in V\}$, where $t'(w) = t(vw)$ for all $w \in V'$. If V is not explicitly named, we may denote it by $V(t)$. The *rank* of a tree t is the maximum number of children of vertices of t. Given trees $t_1, \ldots, t_n$ and a symbol α, we let $\alpha[t_1, \ldots, t_n]$ denote the tree t with $t(\varepsilon) = \alpha$ and $t/i = t_i$ for all $i \in \{1, \ldots, n\}$. The tree $\alpha[]$ may be abbreviated by α.

Given an alphabet Σ, the set of all trees of the form $t\colon V \to \Sigma$ is denoted by T_Σ. Moreover, if Q is an alphabet disjoint with Σ, we denote by $T_\Sigma(Q)$ the set of all trees $t\colon V \to \Sigma \cup Q$ such that only leaves may be labeled with symbols in Q, i.e., $t(v) \in Q$ implies that $v \cdot 1 \notin V$.

Finally, we recall the definition of non-deterministic finite automata and string-to-tree transducers.

A *non-deterministic finite automaton* (NFA) is a tuple $A = (Q, \Sigma, q_0, \delta, F)$ where Q is a finite set of states, Σ is an alphabet with $\varepsilon \notin \Sigma$, $q_0 \in Q$, $F \subseteq Q$ and $\delta\colon Q \times (\{\varepsilon\} \cup \Sigma) \to \mathscr{P}(Q)$ is the transition function. A string $w \in \Sigma^*$ is accepted by A if and only if there exist $\alpha_1, \ldots, \alpha_m \in \Sigma \cup \{\varepsilon\}$ and $p_1 \cdots p_{m+1} \in Q^*$, the latter being the *accepting run*, such that; $p_{i+1} \in \delta(p_i, \alpha_i)$ for all $i \in \{1, \ldots, m\}$; $\alpha_1 \cdots \alpha_m = w$;

and, finally; $p_1 = q_0$ and $p_m \in F$. The set of strings in Σ^* accepted by A is denoted by $\mathscr{L}(A)$. The fact that $p \in \delta(q,\alpha)$ may also be denoted by $q \xrightarrow{\alpha} p$.

A *string-to-tree transducer* is a tuple $stt = (Q,\Sigma,\Gamma,q_0,\delta)$, where Σ and Γ are the input and output alphabets respectively, Q is the set of states, $q_0 \in Q$ is the initial state, and $\delta\colon Q \times \Sigma \to T_\Gamma(Q)$ is the transition function. When $\delta(q,\alpha) = t$ we also write $q \xrightarrow{\alpha} t$.

For $\alpha_1,\ldots,\alpha_n \in \Sigma$, $stt(\alpha_1 \cdots \alpha_n)$ is the set of all trees $t \in T_\Gamma$ such that there exists a sequence of trees $t_0,\ldots,t_n$ which fulfill the requirement that $t_0 = q_0$ and $t_n = t$; and; for every $i \in \{1,\ldots,n\}$, t_i is obtained from t_{i-1} by replacing every leaf v for which $t_{i-1}(v) \in Q$ with a tree in $\delta(t_{i-1}(v),\alpha_i)$, i.e., it holds that $t_i/v \in \delta(t_{i-1}(v),\alpha_i)$.

3 Regular Expression Matching in Java

Here we will take a look at the algorithm used for matching regular expressions in Java, using the default `java.util.regex` package, and describe in pseudocode roughly how matching is accomplished in this package. The Java implementation is a good representative of the class of NFA search matchers. It is both fairly typical and very consistent. Java 6, 7 and 8 all function the same (Java 1.6.0u27 is used to generate figures here), and many other implementations behave similarly, e.g. the popular Perl Compatible Regular Expressions library (PCRE).

The meat of the implementation lives in `java.lang.regex.Pattern` which for a regular expression constructs an object graph of subclasses of the class `java.lang.regex.Pattern$Node` (we abbreviate this as just `Node`, assuming all classes to be inner classes of `java.lang.regex.Pattern` unless otherwise stated). `Node` objects correspond to states, encapsulating their transitions in addition, and have one relevant method, `boolean Node.match(Matcher m, int i, CharSequence s)`, which we will closely mimic later. The implicit `this` pointer of the method call corresponds to the state, `s` is the entire string, `i` is the index of the next symbol to be read. The *m* argument contains a variety of book-keeping, notably it contains variables corresponding to *C* in Algorithm 4, as well as the information on what the accepting run was after the fact (`match` instead returns `true` if and only if the node can (potentially recursively) match the remainder of the string. Every `Node` contains at least a pointer `next` which serves as the "default" next transition out of the node. Let us look at the object graph on the left in Figure 1. As can be seen there are quite a few nodes even for a small expression like ab^*, but most are needed for fairly minor book-keeping, and in general implement features tangential to our concerns here. For example LastNode checks that all symbols are read by the matching, but can be made to do other things using additional features in `java.util.regex` which we do not deal with.

The matching starts with a call to `match` on Begin with the full string (i.e., `i` set to one and the string in `s`). See Figure 2 for pseudo-code for the behavior of Begin, Single and Curly. Begin (and LastNode) are trivial, they just check that we are in the expected position of the string, and in the case of Begin calls into its `next`. Single reads a single symbol (equal to its internal `c`) and continues to `next`. Accept is even more trivial and always returns true. Curly is where things get more complex. Curly handles the Kleene closure, and, since it has to resolve non-determinism (i.e. how many repetitions to perform), it is a bit more complex. The values `type`, `cmin`, and `cmax` are irrelevant for our concerns, they implement the counted repetition extension. In reality the statement on line 2 on the right works by updating values in the `m` in-out argument left unspecific here. Curly starts by trying to match the atom node `atom` to a prefix of the string, if it succeeds Curly calls itself recursively (call match on `this`) with the remainder. When `atom` fails to match any further Curly instead continues to `next` (backtracking as needed). In reality Curly uses imperative loops for efficiency, but it only serves to achieve a constant speedup and

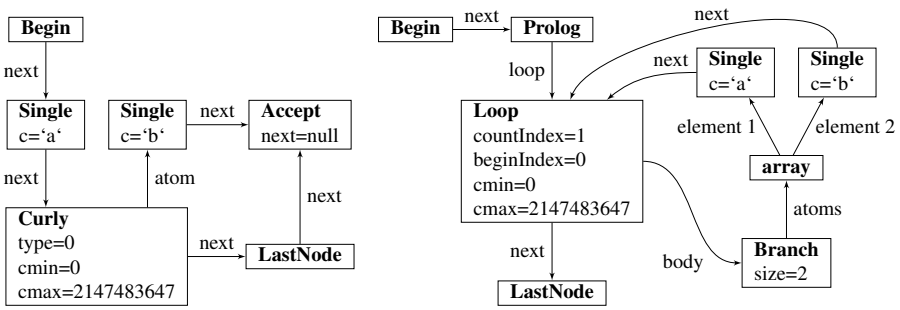


Figure 1: The left diagram shows essentially the complete internal object graph of subclasses of `Node` Java constructs for ab^*. On the right we show a simplified version of the corresponding object graph for $(a|b)^*$. In the latter all nodes without matching effect (in our limited expressions) are removed (e.g. the node Accept seen in the more complete example on the left).

```
1: if i = 0 then
2:     return next.match(i,w)
3: else
4:     return false
5: end if
```

```
1: if α_i = c then
2:     return next.match(i+1,w)
3: else
4:     return false
5: end if
```

```
1: if atom.match(i,w) then
2:     b := #symbols read above
3:     if this.match(i+b,w) then
4:         return true
5:     end if
6: end if
7: return next.match(i,w)
```

Figure 2: The code for a call of the form `match`$(i, w = \alpha_1 \cdots \alpha_n)$ on a Begin (left), Single (middle) and Curly (right) node. Single has a member variable `c` identifying the symbol it should read. Curly tries to recursively repeat `atom`, calling `next` when that fails.

is as such irrelevant for us. Curly is not used for all Kleene closures, if $b = 0$ it would loop forever, so the construction procedure for the object graph only uses Curly when it (with a fairly limited decision procedure) can tell that the contents looped is of constant non-zero length.

Next we look at the more general example on the right side of Figure 1. Here there are some additional nodes to consider. Branch implementing the union, and Prolog and Loop implementing the Kleene closure together (with Prolog calling `matchInit` on Loop to initialize the loop). Let us look at each function in Figure 3. Notably `match` in Branch starts by letting the first subexpression match, continuing with the second and so on if the first attempts fail. The symbiotic relationship between Prolog and Loop is trickier. Where all other nodes calls into Loop with `match`(i,w) as usual Prolog calls in with `matchInit`(i,w) (on the left in Figure 3). This serves only one purpose: it eliminates ε-cycles. That is, it prevents Loop from recursively matching `body` to the empty string, making no progress. In `matchInit` the current value of i is stored, and in `match` (in the middle in Figure 3) the node `body` will only get a match attempted if at least one symbol has been read since the last attempt.

As an additional example, consider the regular expression $(a|a)^*$, which has an object graph almost like on the right of Figure 1, except the second Single *also* has `c` set to a. Matching this against $aa\cdots ab$ will take exponential time in the number of as, as all ways to match each a to each Single in $a|a$ will be tried as the matching backtracks trying to match the final b. In an experiment on an authors desktop PC

```
1: this.sp := i
2: if body.match(i,w) then
3:     return true
4: else
5:     return next.match(i,w)
6: end if
```

```
1: if i > this.sp then
2:     if body.match(i,w) then
3:         return true
4:     end if
5: end if
6: return next.match(i,w)
```

```
1: for e ∈ array do
2:     if e.match(i,w) then
3:         return true
4:     end if
5: end for
6: return false
```

Figure 3: On a call of the form $\texttt{match}(i, w = \alpha_1 \cdots \alpha_n)$ to, on the left to `Loop.matchInit` (called by Prolog in lieu of `match`), in the middle to `Loop.match` (called by all other nodes), and on the right to Branch. Notice that the loop in Branch is *in array order*.

an attempt to match $(a|a)^*$ to $a^{35}b$ using Java took roughly an hour of CPU time.

As was already mentioned space does not permit all details to be considered, and a formal proof of Java semantics is in general outside of our scope. Let us explain some details skipped however.

The object graph on the right in Figure 1 is, as is noted in the caption, a bit of creative editing of reality. A number of nodes not affecting the search behavior or matching are removed, in total the Accept node, which is just a `next` placeholder with no effect, GroupHead and GroupTail, which tracks what part of the match corresponds to a parenthesized subexpression, and finally BranchConn, which is placed in relation to Branch in the right of Figure 1 and records some information for the optimizer (which is responsible e.g. for choosing whether Curly or Loop should be used).

In general all nodes have numerous additional features not discussed, and there are many additional nodes serving similar purposes. For example Single may be replaced with Slice (matches multiple symbols at once) or BnM (matches multiple symbols using Boyer-Moore matching [2]). However, the optimizations are too minor to matter for our concerns (e.g. replacing a sequence of Slice and BnM by Single nodes will be at most a linear slow-down), and the additional features are outside our scope.

To wrap this section up we take the above together and assemble the snippets of matching code into a function which takes a regular expression and a string as input and decides if the expression matches the string. A regular expression is represented by its parse tree, $T : \mathbb{N}_+^* \to \{|, *, *?, \cdot, \varepsilon\} \cup \Sigma$. The operator $*?$ is the lazy Kleene closure, the same as $*$ except it attempts to make as *few* repetitions as possible. We will, naturally, assume that $\cdot$ and $|$ have two children, $*$ and $*?$ one, and each $\alpha \in \Sigma \cup \{\varepsilon\}$ zero.

Define the function $next : \mathbb{N}_+ \to \mathbb{N}_+$ on the nodes of T as follows (compare to the `cont` pointers in Kirrage at al. [7]). Let $next(\varepsilon) = nil$, and

1. If $T(v) = |$ then $next(v \cdot 1) = next(v \cdot 2) = next(v)$.
2. If $T(v) = \cdot$ then $next(v \cdot 1) = v \cdot 2$ and $next(v \cdot 2) = next(v)$.
3. If $T(v) = *$ or $T(v) = *?$ then $next(v \cdot 1) = v$.

Then, collapsing the object graph and ignoring precise node choices in Java we get Algorithm 1.

Algorithm 1. *This is a collapsed fragment of the Java matching algorithm. The implicit regular expression parse tree is T. There are three call-by-value input parameters, the node of the tree currently processed, the remainder of the string to match, and the set of nodes that we should not revisit until at least one symbol has been read. This prevents ε-cycles in a way similar to what was discussed above. The initial call made is* $\textsc{Match}(\varepsilon, w, \emptyset)$.

```
1: function MATCH(v, w = a_1 ··· a_n, C)
2:     if v = nil then
3:         return n = 0
4:     else if T(v) = ε then
5:         return MATCH(next(v), w, C)
6:     else if T(v) ∈ Σ then
```

```
 7:      if n ≥ 1 ∧ T(v) = a_1 then
 8:          return MATCH(next(v), a_2 ··· a_n, ∅)
 9:      end if
10:      return false
11:  else if T(v) = | then
12:      if MATCH(v·1, w, C) then
13:          return true
14:      end if
15:      return MATCH(v·2, w, C)
16:  else if T(v) = · then
17:      return MATCH(v·1, w, C)
18:  else if T(v) = * then
19:      if v·1 ∉ C then
20:          if MATCH(v·1, w, C ∪ {v·1}) then
21:              return true
22:          end if
23:      end if
24:      return MATCH(next(v), w, C)
25:  else if T(v) = *? then
26:      if MATCH(next(v), w, C) then
27:          return true
28:      else if v·1 ∉ C then
29:          return MATCH(v·1, w, C ∪ {v·1})
30:      else
31:          return false
32:      end if
33:  end if
34: end function
```

Notice how the code for the two Kleene closure variants *only* differ in what they try first: * tries to repeat its body first, whereas $^{*?}$ tries to not repeat the body.

Notice also how C is used to prevent ε-loops in the Kleene closure cases (lines 19–20 and 28–29). If the node we *would* go to is already in C this means that no symbol has been read since last time we tried this, meaning repeating it would be a loop making no progress.

4 Prioritized Non-Deterministic Finite Automata

Here we will define a new type of non-deterministic finite automaton, with some additions. These modifications have no impact on the language accepted, but makes the automaton "run-deterministic", every string in the language accepted has a well-defined unique accepting run, a property brought about by ordering every non-deterministic choice, giving some alternatives higher-priority than others, and letting the unique accepting run be the highest priority run for that string.

Definition 2. *A* prioritized non-deterministic finite automaton *(or* pNFA*) is a tuple of seven elements, $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, where $Q_1 \cap Q_2 = \emptyset$; Σ is a finite alphabet; $q_0 \in Q_1 \cup Q_2$ is the initial state; $\delta_1 \colon Q_1 \times \Sigma \to (Q_1 \cup Q_2)$ is the deterministic transition function; $\delta_2 \colon Q_2 \to (Q_1 \cup Q_2)^*$ is the non-deterministic prioritized transition function; and $F \subseteq Q_1 \cup Q_2$ are the final states.*

The NFA corresponding to the pNFA A is given by $\overline{A} = (Q_1 \cup Q_2, \Sigma, q_0, \bar{\delta}, F)$, where

$$\bar{\delta}(q,\alpha) = \begin{cases} \{\delta_1(q,\alpha)\} & \text{if } q \in Q_1 \text{ and } \alpha \in \Sigma, \\ \{q_1, \ldots, q_n\} & \text{if } q \in Q_2,\ \alpha = \varepsilon, \text{ and } \delta_2(q) = q_1 \cdots q_n. \end{cases}$$

The language accepted by *A, denoted by $\mathscr{L}(A)$, is $\mathscr{L}(\overline{A})$.*

Next, we define the so-called backtracking run of a pNFA on an input string w. This run takes the form of a tree which, intuitively, represents the attempts a matching algorithm such as Algorithm 1 would make until accepting the input string (or eventually rejecting it).

Definition 3. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA, $q \in Q_1 \cup Q_2$, $w = \alpha_1 \cdots \alpha_n \in \Sigma^*$, and $C \colon Q_2 \to \mathbb{N}$. Then the* (q,w,C)-backtracking run *of A is a tree over $Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. A backtracking run* succeeds *if and only if Acc occurs in it. We denote the (q,w,C)-backtracking run by $btr_A(q,w,C)$ and inductively define it as follows. If $q \in F$ and $w = \varepsilon$ then $btr_A(q,w,C) = q[Acc]$. Otherwise, we distinguish between two cases:*

1. *If $q \in Q_1$, then*

$$btr_A(q,w,C) = \begin{cases} q[btr_A(\delta_1(q,\alpha_1),\alpha_2\cdots\alpha_n,0^{Q_2})] & \text{if } n>0 \text{ and } \delta_1(q,\alpha_1) \text{ is defined,} \\ q[Rej] & \text{otherwise.} \end{cases}$$

2. *If $q \in Q_2$ with $\delta_2(q) = q_1\cdots q_k$, let $i_0 = C(q)+1$ and $r_i = btr_A(q_i,w,C_{q\mapsto i})$ for $i_0 \le i \le k$. Then*

$$btr_A(q,w,C) = \begin{cases} q[Rej] & \text{if } i_0 > k, \\ q[r_{i_0},\ldots,r_k] & \text{if } i_0 \le k \text{ but no } r_j \text{ succeeds,} \\ q[r_{i_0},\ldots,r_j] & \text{if } j \in \{i_0,\ldots,k\} \text{ is the least index such that } r_j \text{ succeeds.} \end{cases}$$

The backtracking run of A on w *is* $btr_A(w) = btr_A(q_0,w,0^{Q_2})$. *If $btr_A(w)$ succeeds, then the* accepting run of A on w *is the sequence of states on the right-most path in $btr_A(w)$.*

Notice that the third parameter C in $btr_A(q,w,C)$ fulfills a similar purpose as the set C in Algorithm 1. It is used to track transitions that must not be revisited to avoid cycles.

Clearly, for a pNFA A and a string w, $w \in \mathscr{L}(A)$ if and only if $btr_A(w)$ succeeds, if and only if the accepting run of A on w is an accepting run of the NFA $\overline{A}$. Backtracking runs capture the behavior of the following algorithm which generalizes Algorithm 1 to arbitrary pNFAs to deterministically find the accepting run of A on w if it exists.

Algorithm 4. *Let $A = (Q_1,Q_2,\Sigma,q_0,\delta_1,\delta_2,F)$ be a pNFA. The call* MATCH$(q_0,w,0^{Q_2})$[2] *of the following procedure yields the accepting run of A on w if it exists, and $\perp \notin Q_1 \cup Q_2$ otherwise. The third is similar to the C in Definition 3. For every state $q \in Q_2$ with out-degree d, $C(q)$ ranges between 0 and d.*

```
1:  function MATCH(q,w = a_1 ··· a_n,C)
2:    if q ∈ Q_1 then
3:      if n = 0 then
4:        if q ∈ F then
5:          return q
6:        else
7:          return ⊥
8:        end if
9:      else
10:       return q · MATCH(δ_1(q,a_1),a_2 ··· a_n,0^{Q_2})
11:     end if
12:   else
13:     if n = 0 ∧ q ∈ F then
14:       return q
15:     else
16:       q_1 ··· q_k := δ_2(q)
17:       for i = C(q)+1,...,k do
18:         r := MATCH(q_i,w,C_{q↦i})
19:         if r ≠ ⊥ then
20:           return q · r
21:         end if
22:       end for
23:       return ⊥
24:     end if
25:   end if
26: end function
```

Notice especially line 10 where a symbol gets read, and C reset to 0^{Q_2} in the recursive call. The case for Q_2 starts at line 13, the loop at 17 tries all not yet tried *transitions for that state. If no transition succeeds we fail on line 23.*

We note here that the running time of Algorithm 4 is exponential in general, just like Algorithm 1. This can be remedied by means of memoization, but potentially with a significant memory overhead, due to the fact that memoization needs to keep track of each possible assignment to all of $C(q)$, for $q \in Q_2$.[3]

Depending on how one turns a given regular expression into a pNFA, Algorithm 4 will run more or less efficiently. For example, if the pNFA is built in a way that reflects Algorithm 1, analyzing the efficiency of Algorithm 4 or, equivalently, the size of backtracking runs, yields a (somewhat idealized) statement about the efficiency of the Java matcher.

[2] Recall that 0^{Q_2} denotes the function $C\colon Q_2 \to \mathbb{N}$ such that $C(q) = 0$ for all $q \in Q_2$.

[3] Apparently, starting from version 5.6 Perl uses memoization in its regular expression engine in order to speed up matching.

4.1 Two Constructions for Turning Regular Expressions into pNFA

In this section we give two examples of constructions that can be used to turn a regular expression E into a pNFA A such that $\mathscr{L}(A) = \mathscr{L}(E)$. The first is a prioritized version of the classical Thompson construction [9], whereas the second follows the Java approach.

Recall that the classical Thompson construction converts the parse tree T of a regular expression E to a NFA, which we denote by $Th(E)$, by doing a postorder traversal on T. A NFA is constructed for each subtree T' of T, equivalent to the regular expression represented by T'. We do not repeat this well-known construction here, assuming that the reader is familiar with it. Instead, we define a prioritized version, which constructs a pNFA denoted by $Th^p(E)$ such that $\overline{Th^p(E)} = Th(E)$.

Just as the construction for $Th(E)$, we define $Th^p(E)$ recursively on the parse tree for E. For each subexpression F of E, $Th^p(F)$ has a single initial state with no ingoing transitions, and a single final state with no outgoing transitions. The constructions of $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, and $Th^p(F_1 \cdot F_2)$, given that $Th^p(F_1)$ and $Th^p(F_2)$ are already constructed, are defined as for $Th(E)$, splitting the state set into Q_1 and Q_2 in the obvious way. It is only when we construct $Th^p(F_1|F_2)$ from $Th^p(F_1)$ and $Th^p(F_2)$, and $Th^p(F_1^*)$ from $Th(F_1)$, where the priorities of introduced ε-transitions require attention. We also consider the lazy Kleene closure, denoted by $F_1^{*?}$, to illustrate the difference in priorities of transitions between constructions for the greedy and lazy Kleene closure. We only discuss the constructions of $Th^p(F_1|F_2)$, $Th^p(F_1^*)$ and $F_1^{*?}$. In each of the constructions below, we assume that $Th^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $Th^p(E)$. All non-final states in $Th^p(E)$ that are in $Th^p(F_i)$ inherit their outgoing transitions from $Th^p(F_i)$.

- If $E = F_1|F_2$ then $Th^p(E)$ is built in precisely the same way as $Th(E)$, thus introducing new initial and final states q_0 and f_0, respectively, and defining $\delta_2(q_0) = q_1q_2$ and $\delta_2(f_1) = \delta_2(f_2) = f_0$.
- If $E = F_1^*$ then we add new initial and final states q_0 and f_0 to Q_2 and define $\delta_2(q_0) = q_1f_0$ and $\delta_2(f_1) = q_1f_0$. The case $E = F_1^{*?}$ is the same, except that $\delta_2(q_0) = f_0q_1$ and $\delta_2(f_1) = f_0q_1$.

Thus, the pNFA $Th^p(F^*)$ tries F as often as possible whereas $Th^p(F^{*?})$ does the opposite.

The second pNFA construction is the one implicit in the Java approach and Algorithm 1. We denote this pNFA by $J^p(E)$. The base cases $J^p(\emptyset)$, $J^p(\varepsilon)$, $J^p(a)$ are identical to $Th^p(\emptyset)$, $Th^p(\varepsilon)$, $Th^p(a)$, respectively. Now, let us consider the remaining operators. Again, we assume that $J^p(F_i)$ ($i \in \{1,2\}$) has the initial state q_i and the final state f_i. Furthermore, δ_2 denotes the transition function for ε-transitions in the newly constructed pNFA $J^p(E)$.

- Assume that $E = F_1 \cdot F_2$. Then $J^p(E)$ is built from $J^p(F_1)$ and $J^p(F_2)$ by identifying f_1 with q_2, adding a new initial state $q_0 \in Q_2$ with $\delta_2(q_0) = q_1$, and making f_2 the final state. Thus, $J^p(E)$ is built like $Th^p(E)$, except that a new initial state is added and connected to the initial state of $J^p(F_1)$ by means of an ε-transition.
- If $E = F_1|F_2$ then $J^p(E)$ is constructed by introducing a new initial state q_0, defining $\delta_2(q_0) = q_1q_2$, and identifying f_1 and f_2, the result of which becomes the new final state.
- Now assume that $E = F_1^*$. Then we add a new final state f_0 to $J^p(F_1)$, make $q_0 = f_1$ the initial state of $J^P(E)$, and set $\delta_2(q_0) = q_1f_0$. The case $E = F_1^{*?}$ is exactly the same, except that $\delta_2(q_0) = f_0q_1$.

Observation 5. *Let E be a regular expression and A a pNFA. Then the running time of Algorithm 4 on w (with respect to E) is $\Theta(|btr_A(w)|)$.*

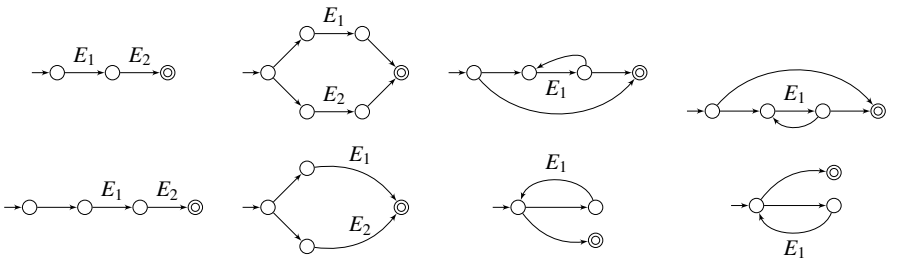
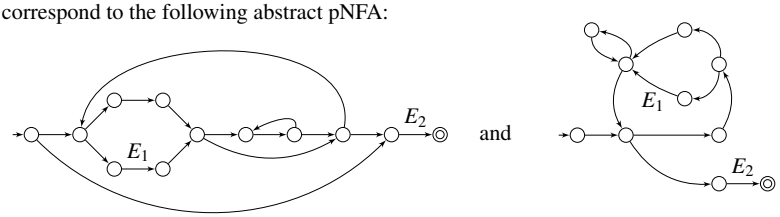


Figure 4: Abstract pNFA corresponding to $E_1 \cdot E_2$, $E_1 \,|\, E_2$, E_1^* and $E_1^{*?}$, from which $Th^p(E)$ (top row) and $J^p(E)$ (bottom row) are constructed.

The two variants of implementing regular expressions by pNFA are closely related. In fact, Kirrage et al. [7] seem to regard them as being essentially identical and write that their reasons for choosing $\overline{J^p(E)}$ is "purely of presentational nature". However, using our notion of pNFA we can show that this is not always the case. For this, note first that the construction of both $Th^p(E)$ and $J^p(E)$ can be viewed in a top-down fashion, where each operation is represented by an abstract pNFA in which zero, one, or two transitions are labeled with regular expressions. Replacing such a transition with the corresponding pNFA yields the constructed pNFA for the whole expression. Figure 4 shows the building blocks for the operations $\cdot$, $|$, *, and $^{*?}$ in both cases. Priorities follow the convention that ε-transitions leaving a state are drawn in clockwise order, starting at noon. Unlabeled edges denote ε-transitions.

Now consider an expression E of the form $((\varepsilon|E_1)\cdot\varepsilon^*)^*\cdot E_2$. When building $Th^p(E)$ and $J^p(E)$, these correspond to the following abstract pNFA:



In $Th^p(E)$, when processing an input string w, the run will first choose the prioritized choice of the union operator (which is ε), iterate the inner loop once, and then return to the initial state of the sub-pNFA corresponding to $\varepsilon|E_1$. Now, the first alternative is blocked, meaning that Algorithm 4 tries to match E_1. Assuming that no failure occurs, it will then proceed by following ε transitions leading to E_2.

Now look at $J^p(E)$. Here, the run first bypasses E_1, similarly to $Th^p(E)$, but this leads to the state following the start state. As the first alternative of transitions leaving this state has already been used, the run drops out of the loop and proceeds with E_2. E_1 will only be tried after backtracking in case E_2 fails.

We thus get several cases by appropriately instantiating E_1 and E_2. Assume first that we choose E_1 in such a way that $Th^p(E_1)$ suffers from exponential backtracking on a set W of input strings over Σ, and $E_2 = \Sigma^*$. Then $Th^p(E)$ causes exponential backtracking on strings in W whereas $J^p(E)$ does not backtrack at all. A concrete example is obtained by taking $\Sigma = \{a,b\}$, $E_1 = (a^*)^*$, and $W = \{a^n b \mid n \in \mathbb{N}\}$.

Conversely, we may choose $E_2 = \varepsilon \,|\, E_2'$ so that $J^p(E_2')$ fails exponentially on W, but $E_1 = \Sigma^*$. Then $Th^p(E)$ will match strings in W in linear time whereas $J^p(E)$ will take exponential time.

Finally, we can of course easily combine two examples of the types above into one, to obtain an

expression such that $Th^p(E)$ shows exponential behavior on a set W of strings on which $J^p(E)$ runs in linear time whereas $J^p(E)$ shows exponential behavior on another set W' of strings on which $Th^p(E)$ runs in linear time.

5 Static Analysis of Exponential Backtracking

We now consider the problem of deciding whether a given pNFA causes backtracking matching similar to Algorithm 1 to run exponentially. More precisely, we ask whether a pNFA has exponentially large backtracking runs. In the case where the considered pNFA is $J^p(E)$, this yields a statement about the running time of Algorithm 1. However, we are interested in the problem in general, because other regular expression engines may correspond to other pNFA. The decision problem comes in two flavors, with very different complexities. Let us start by defining the first.

Definition 6. *Given a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, let $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*, |w| \leq n\}$ for all $n \in \mathbb{N}$. We say that A has* exponential backtracking *if $f \in 2^{\Omega(n)}$,* polynomial backtracking of degree *k for $k \in \mathbb{N}$ if $f \in \Theta(n^{k+1})$, and* finite backtracking *if $f \in \Theta(n)$.*

Definition 7. *Given a pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$, let let A^f be the pNFA $(Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, \emptyset)$, and let $g(n) = \max\{|btr_{A^f}(w)| \mid w \in \Sigma^*, |w| \leq n\}$ for all $n \in \mathbb{N}$. We say that A has* exponential failure backtracking *if $g \in 2^{\Omega(n)}$,* polynomial failure backtracking of degree *$k \in \mathbb{N}$ if $g \in \Theta(n^{k+1})$, and* finite failure backtracking *if $g \in \Theta(n)$.*

Definition 7 provides an upper bound for the general case as defined in Definition 6. Also in cases where we know that the worst-case matching complexity complexity can be exhibited by a family of strings not in $\mathscr{L}(A)$, this analysis is precise. This happens for example if for some $\$ \in \Sigma$, we have $w\$ \notin \mathscr{L}(A)$ for all $w \in \Sigma^*$, or more generally, if for each $w \in \Sigma^*$, there is $w' \in \Sigma^*$, with $|w'| \leq k$ for some fixed constant k, such that $ww' \notin \mathscr{L}(A)$. Definition 7 is of great interest in that it is *much* easier to decide, being in PTIME, instead of being PSPACE-hard. Definition 7 is closely related to the case considered in e.g. [7], where matching complexity of the strings not in $\mathscr{L}(A)$ are considered.

5.1 An Upper Bound on the Complexity of General Backtracking Analysis

Let us first establish an upper bound on the complexity of general backtracking analysis. We will give an algorithm which solves this problem in EXPTIME. Afterwards, we will also note some minor hardness results. The EXPTIME decision procedure relies heavily on a result from [4].

Lemma 8. *Given a string-to-tree transducer $stt = (Q, \Sigma, \Gamma, q_0, \delta)$, it is decidable in deterministic exponential time whether the function $f(n) = \max\{|t| \mid t \in stt(s),\ s \in \Sigma^*,\ |s| \leq n\}$ grows exponentially, i.e. whether $f \in 2^{\Omega(n)}$.*

In short, we will hereafter construct a string-to-tree transducer from a pNFA A which reads an input string (suitably decorated) and outputs the corresponding backtracking run of A (see Definition 3). In this way, we model the running of Algorithm 4 on that string. Then Lemma 8 can be applied to this transducer to decide exponential backtracking. To simplify the construction we first make a small adjustment to the input pNFA in the form of a "flattening", which ensures that δ_2 maps Q_2 to Q_1^*. That is, we remove the opportunity for repeated ε-transitions.

Definition 9. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA. Define $d\colon (Q_1 \cup Q_2) \times (Q_2 \to \mathbb{N}) \to Q_1^*$, and $\bar{r}\colon Q_1^* \to Q_1^*$ as follows:*

$$d(q, C) = \begin{cases} q & \text{if } q \in Q_1, \\ d(q_{i+1}, C_{q \mapsto i+1}) \cdots d(q_n, C_{q \mapsto i+1}) & \text{if } q \in Q_2,\ \delta_2(q) = (q_1 \cdots q_n) \text{ and } C(q) = i. \end{cases}$$

$$\bar{r}(s) = \begin{cases} \bar{r}(uv) & \text{if } s = uqv \text{ for some } u,v \in Q_1^* \text{ and } q \in Q_1 \text{ with } |u|_q \geq 2 \\ s & \text{otherwise.} \end{cases}$$

That is, $\bar{r}$ removes all repetitions of each state q beyond the first two occurrences.

*Now, the δ_2-*flattening of A *is the pNFA $A' = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2', F')$ with $\delta_2'(q) = \bar{r}(d(q, 0^{Q_2}))$ for all $q \in Q_2$, and $F' = \{q \in Q_1 \cup Q_2 \mid d(q, 0^{Q_2}) \cap F \neq \emptyset\}$.*

First let us note that A' in Definition 9 can be computed in polynomial time. In fact, the size of A' is polynomial in the size of A, as no new states are added and no right-hand side is greater than polynomial in length ($2|Q_1|$ is the maximum length after applying $\bar{r}$). The construction itself can be performed in polynomial time in a straightforward way by computing d incrementally in a left-to-right fashion, and aborting each recursion visiting a state that has already been seen twice to the left.

Before proving some properties of the above construction we clarify a supporting observation.

Lemma 10. *Let σ be a function on trees such that, for $t = f[t_1, \ldots, t_k]$*

$$\sigma(t) = \begin{cases} t & \text{if } k = 0 \\ f[\sigma(t_1)] & \text{if } k = 1 \\ f[\sigma(t_i), \sigma(t_j)] & \text{otherwise, where } t_i, t_j \ (i \neq j) \text{ are largest among } t_1, \ldots, t_k. \end{cases}$$

Let $T_0, T_1, T_2, \ldots$ be sets of trees of rank at most k. Then the function $f(n) = \max\{|t| \mid t \in T_n\}$ grows exponentially if and only if $f'(n) = \max\{|\sigma(t)| \mid t \in T_n\}$ grows exponentially.

The proof of the lemma can be found in the appendix.

Lemma 11. *Let $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ be a pNFA and A' its δ_2-flattening. Then A' can be constructed in polynomial time, $\mathscr{L}(A') = \mathscr{L}(A)$, and the function $f(n) = \max\{|btr_A(w)| \mid w \in \Sigma^*, \ |w| \leq n\}$ grows exponentially if and only if $f'(n) = \max\{|btr_{A'}(w)| \mid w \in \Sigma^*, \ |w| \leq n\}$ grows exponentially.*

Proof sketch. Let $A' = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2', F')$. As noted, A' can be constructed in polynomial time.

The language equivalence of A and A' can be established by induction on the accepting runs of A and A'. δ_2' is a closure on δ_2, such that any accepting run for A of the form $p_1 \cdots p_n$ can be turned into one for A' by replacing each maximal subsequence $p_k \cdots p_{k+i} \in Q_2^*$ with just p_k. The function d in the construction of δ_2 will ensure that p_k is accepting if this was at the end of the run, and that p_k can go directly to the following Q_1 state. The converse is equally straightforward, as a suitable sequence from Q_2 can be inserted into an accepting run for A' to create a correct accepting run for A.

Finally, we argue that A' exhibits exponential backtracking behavior if and only if A does. By the construction of A', we have $btr_{A'}(w) \leq btr_A(w)$. Hence, f grows exponentially if f' does. It remains to consider the other direction. Thus, assume that $f(n)$ grows exponentially. We have to show that $f'(n)$ grows exponentially as well. Let A'' be the pNFA generated by δ_2-flattening A without applying $\bar{r}$. Let $t = btr_A(w)$ and $t'' = btr_{A''}(w)$ for some input string w. Then t'' is obtained from t by repeatedly replacing subtrees of the form $q[s_1, \ldots, s_k, q'[t_1, \ldots, t_l], s_{k+1}, \ldots, s_m]$, where $q, q' \in Q_2$, by $q[s_1, \ldots, s_k, t_1, \ldots, t_l, s_{k+1}, \ldots, s_m]$. Since Definition 3 prevents repeated ε-cycles, this process removes only a constant fraction of the nodes in t.[4] Hence, $f''(n) = \max\{|btr_{A''}(w)| \mid w \in \Sigma^*, \ |w| \leq n\}$ grows exponentially. Now, compare t'' with $t' = btr_{A'}(w)$. If a node of t'' has m children with the same state $q \in Q_2$ in their roots, by the definition of backtracking runs the m subtrees rooted at those nodes will be identical. The application of $\bar{r}$ to A'' means that, in effect, the first two copies of these m subtrees are kept in t'. In particular, the two largest subtrees of the node are kept in t'. According to Lemma 10, this means that g' grows exponentially. □

[4] The constant may be exponential in the size of A, but for the question at hand this does not matter.

It should be noticed that, for the proof above to be valid, it is important that $\bar{r}$ preserves the order of occurrences of states from the left, as a subtree being accepting means that no further subtrees are constructed to the right of it.

We are now prepared to define the construction which for any δ_2-flattened pNFA A produces a string-to-tree transducer stt such that $btr_A(w) = t$ if and only if $t \in stt(w')$. Here, w' is a version of w decorated with extra symbols $\flat$ and \$. The former will serve as padding to be read when δ_2 transitions are taken, and \$ marks the beginning and the end of the string.

Definition 12. *Given a δ_2-flattened pNFA $A = (Q_1, Q_2, \Sigma, q_0, \delta_1, \delta_2, F)$ we construct the string-to-tree transducer $stt = (Q, \Sigma', \Gamma, q_0', \delta)$ in the following way. $Q = \{q_0'\} \cup \{a_q, f_q \mid q \in Q_1 \cup Q_2\}$, $\Sigma' = \Sigma \uplus \{\flat, \$\}$, and $\Gamma = Q_1 \cup Q_2 \uplus \{Acc, Rej\}$. Furthermore, δ consists of the following transitions:*

1. *Let $q_0' \xrightarrow{\$} a_{q_0}$ and $q_0' \xrightarrow{\$} f_{q_0}$. For all $q \in Q$ let $q \xrightarrow{\flat} q$.*
2. *For all $q \in Q_1$ and $\alpha \in \Sigma$:*
   (a) *If $\delta_1(q, \alpha) = q'$ let $a_q \xrightarrow{\alpha} q[a_{q'}]$ and $f_q \xrightarrow{\alpha} q[f_{q'}]$.*
   (b) *If $\delta_1(q, \alpha)$ is undefined let $f_q \xrightarrow{\alpha} q[Rej]$.*
3. *For all $q \in Q_2$, if $q_1 \cdots q_n = \delta_2(q)$, then for all $i \in \{0, \ldots, n-1\}$ let $a_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_i}, a_{q_{i+1}}]$, and let $f_q \xrightarrow{\flat} q[f_{q_1}, \ldots, f_{q_n}]$.*
4. *Finally if $q \in F$ let $a_q \xrightarrow{\$} q[Acc]$, whereas when $q \notin F$:*
   (a) *if $q \in Q_1$ let $f_q \xrightarrow{\$} q[Rej]$, and,*
   (b) *if $q \in Q_2$ and $q_1 \cdots q_n = \delta_2(q)$, then $f_q \xrightarrow{\$} q[q_1[Rej], \ldots, q_n[Rej]]$.*

Definition 13. *The string $w_1\alpha_1 w_2 \alpha_2 \cdots w_n \alpha_n w_{n+1}$ is a* decoration *of $\alpha_1 \cdots \alpha_n \in \Sigma^*$ if $w_i \in \{\$, \flat\}^*$ for each i. $\$\flat\alpha_1\flat\alpha_2 \cdots \flat\alpha_n\$$ is the* correct decoration *of $\alpha_1 \cdots \alpha_n$, denoted $dec(\alpha_1 \cdots \alpha_n)$.*

Lemma 14. *For a δ_2-flattened pNFA A, the string-to-tree transducer stt as constructed by Definition 12, and an input string $w = \alpha_1 \cdots \alpha_n$, it holds that $stt(dec(w)) = \{btr_A(w)\}$. For* all *$u$ which are decorations of w either $stt(u) = \emptyset$ or $stt(u) = \{btr_A(w)\}$.*

Proof. First, notice how A being δ_2-flattened impacts btr_A. The flattening ensures that there is no way to take two ε-transitions in a row in A, meaning that every time case 2 of Definition 3 applies, we have $C(q) = 0$ since the previous step is either the initial call or a call from case 1 where C gets reset. As such we will have $C = 0^{Q_2}$ in every recursive call below. Let stt_q denote the string-to-tree transducer stt with the initial state q (instead of q_0).

Let $v = \$\flat\alpha_1\flat\alpha_2\flat\cdots\flat\alpha_n\$$. Establishing that $stt(dec(w)) = \{btr_A(w)\}$ merely requires a straightforward case analysis that can be found in the appendix. Starting with the case where the backtracking run on w *fails*, the analysis establishes that for rejecting backtracking runs $t = btr_A(q, w, 0^{Q_2})$, we have $t \in stt_{f_q}(v)$, for all q, where v equals $dec(w)$ with the initial \$ removed (we will deal with this at the end) and, vice versa, $t \in stt_{f_q}(v)$ is true for exactly one t, so t must be the tree $btr_A(q, w, 0^{Q_2})$.

The proof for the accepting runs follows very similar lines, but with the extra wrinkle of how Q_2 rules are handled when some path accepts. The invariant that $t \in stt_{a_q}(v)$ is true for at most one t is maintained however, as is, of course, the parallel to btr_A. Again, the proof shows that $stt_{a_q}(v)$ outputs precisely one tree if v is $dec(w)$ with the initial \$ removed. That initial \$ is now used by the initial rules in stt: $q_0' \xrightarrow{\$} a_{q_0}$ and $q_0' \xrightarrow{\$} f_{q_0}$. This means that stt produces exactly one tree for every $dec(w)$, and in both the accepting and rejecting case it matches the tree from btr_A.

Finally, we need to deal with *incorrect* decorations. Let v be a decoration of w which is not $dec(w)$. If v has no leading \$, or no trailing \$, or has a \$ in any other position, $stt(v) = \emptyset$, since stt has no other

possible rules for \$. If v contains *extraneous* $\flat$ we still have $stt(v) = \{btr_A(w)\}$, since they will just be consumed by $q \xrightarrow{\flat} q$ rules. If some $\flat$ is "missing" compared to $dec(w)$ this either causes $stt(v) = \emptyset$, if a Q_2 rule needed it, or $stt(v) = \{btr_A(w)\}$, if it is just removed by a $q \xrightarrow{\flat} q$ rule anyway. □

Theorem 15. *It is decidable in exponential time whether a given pNFA A has exponential backtracking.*

Proof. From A, construct the δ_2-flattened pNFA A' according to Definition 9. According to Lemma 11 A' can be constructed in polynomial time, and it has exponential backtracking if and only if A has. Construct the transducer *stt* for A' according to Definition 12. By Lemma 14 *stt* outputs exponentially large trees if and only if A' has exponential backtracking. The construction of *stt* can clearly be implemented to run in polynomial time. Hence, Lemma 8 yields the result. □

5.2 Hardness of General Backtracking Analysis

It seems likely that general backtracking analysis is difficult. We cannot prove this yet, but here we demonstrate that either it is hard to decide if $J^p(E)$ has exponential backtracking *or* the class of regular expressions E such that $J^p(E)$ does *not* have exponential backtracking has an easy universality decision problem. In the following, we say that E has exponential backtracking if $J^p(E)$ does.

Let us briefly recall the universality problem.

Definition 16. *A regular expression E is* Σ-universal *if* $\Sigma^* \subseteq \mathscr{L}(E)$. *The input of* RE Universality *is a regular expression E over an alphabet* Σ. *The question to be answered is whether* $\mathscr{L}(E)$ *is* Σ*-universal.*

This problem is well-known to be PSPACE-complete. See e.g. [6]. We will now give a simple polynomial reduction which takes a regular expression E and constructs a new regular expression E' such that E' has exponential backtracking if E has exponential backtracking *or* E is not universal.

Lemma 17. *Let E be a regular expression over* Σ, $\alpha \in \Sigma$, *and* $\Gamma = \Sigma \cup \{\$\}$ *for some* $\$ \notin \Sigma$. *If E does not have exponential backtracking then* $E' = ((E \,|\, E\$\Gamma^*) \,|\, (\Sigma^*\$(\alpha^*)^*\$)$ *has exponential backtracking if and only if E is not* Σ*-universal.*

Proof. If E does not have exponential backtracking then neither does $E\$\Gamma^*$, since Γ^* never fails. Now, let $A = J^p(E')$. For every input string, the backtracking run of A will attempt to match $\Sigma^*\$(\alpha^*)^*\$$ to the string only if neither E nor $E\$\Gamma^*$ matches it. If E is universal, i.e. equal to Σ^*, then $\mathscr{L}(E|(E\$\Gamma^*)) = \mathscr{L}(\Sigma^*|(\Sigma^*\$\Gamma^*)) = \Gamma^*$ (since a string in Γ^* is either in Σ^* or has a prefix in Σ^* followed by a suffix in Γ^* that begins with a \$). Hence, in this case E' has exponential backtracking if and only if E does.

If we instead assume that E is *not* universal, then there exists some $w \in \Sigma^*$ such that $w \notin \mathscr{L}(E)$. Consider the string $w\$\alpha^n$ for any $n \in \mathbb{N}$. Neither E nor $E\$\Gamma^*$ matches it, which means that backtracking will proceed into $\Sigma^*\$(\alpha^*)^*\$$, where 2^n backtracking attempts will be made to match the suffix $\alpha^n\$$ to the subexpression $(\alpha^*)^*\$$ (as the final \$ keeps failing to match). □

The previous lemma yields the following corollary.

Corollary 18. *Let $\mathscr{E}$ be the set of all regular expressions that do* not *have exponential backtracking. Then either RE Universality is not PSPACE-hard for inputs in $\mathscr{E}$,* or *deciding whether regular expressions have exponential backtracking is PSPACE-hard.*

5.3 The Complexity of Failure Backtracking Analysis

Now we look at the problem to decide whether a given pNFA has exponential failure backtracking (see Definition 7). For reasons of technical simplicity, assume that parallel ε-transitions are absent from pNFA in this section. To simplify the exposition of proofs in this section, we redefine an accepting run of an NFA, as originally defined in Section 2, to exclude accepting runs of $w = \beta_1 \cdots \beta_m \in \Sigma^*$ of the form $p_1 \cdots p_{m+1}$, where $\beta_i = \ldots = \beta_j = \varepsilon$, for some $1 \leq i < j \leq m$, with $p_i = p_j$, and $p_{i+1} = p_{j+1}$. Thus we do not allow an accepting run to contain a consecutive sequence of ε-transitions of which some are the same. We also talk about a *run*, in the case where $p_1 \cdots p_{m+1}$ satisfies the same conditions as for an accepting run, but p_1 is not necessarily the initial state or p_{m+1} an accepting state.

First we recall definitions from [1] on ambiguity for NFA, but for NFA with ε-cycles, these definitions differ from those in [1], due to way we have defined accepting runs in the paragraph above. We define the *degree of ambiguity* of a string w in N, denoted by $da(N,w)$, as the number of accepting runs in N labeled by w. The *degree of ambiguity* of N is defined as $da(N) = \sup_{w \in \Sigma^*} da(N,w)$. N is said to be *finitely ambiguous* if $da(N) < \infty$, and *infinitely ambiguous* otherwise. N is *polynomially ambiguous* if there exists a polynomial h such that $da(N,w) \leq h(|w|)$ for all $w \in \Sigma^*$. The minimal degree of such a polynomial is the *degree of polynomial ambiguity* of N. When N is infinitely ambiguous but not polynomially ambiguous, it is *exponentially ambiguous*.

For an NFA N, we denote by $|N|_Q$ and $|N|_E$ the number of states and transitions of N respectively.

Theorem 19. *For an NFA N it is decidable in time $O(|N|_E^3)$ if N is finitely, polynomially, or exponentially ambiguous, and if N is polynomial ambiguous, the degree of polynomial ambiguity can be computed in time $O(|N|_E^3)$.*

Proof. If N is ε-cycle free, the result follows from Theorems 5 and 6 in [1]. Now let $N = (Q, \Sigma, q_0, \delta, F)$ be an NFA, potentially with ε-cycles, and define the equivalence relation $\sim$ on Q, where $p \sim q$ if and only if they are in the same strongly connected component determined by using only ε-transitions in N. Let $N' := N/\sim$ be the quotient of N by $\sim$, having as states the equivalence classes of $\sim$.

The correctness of the remainder of the argument requires N not to have equivalence classes with two elements, say p, q, where both p and q do not have ε self-loops. We briefly argue how equivalences classes of this form can be removed without changing the ambiguity properties of N. It is tedious, but straightforward, to verify that this can for example be achieved by replacing p and q (and $p \xrightarrow{\varepsilon} q$, $q \xrightarrow{\varepsilon} p$) with 6 states and the appropriately defined ε-transitions to model the behavior of runs in N that go through one or both consecutively of p and q. Three of the 6 states are used to model incoming transitions to p in runs that after reaching p do not follow $p \xrightarrow{\varepsilon} q$, or follow only $p \xrightarrow{\varepsilon} q$, or follow consecutively $p \xrightarrow{\varepsilon} q$ and $q \xrightarrow{\varepsilon} p$, and the other 3 states are used for q in a similar way.

N' could potentially have (parallel) ε self-loops. Let N'' be N' with ε self-loops removed. Each state in N'' will belong to exactly one of the following categories of equivalence classes: (a) a single state of N without an ε self-loop in N; (b) a single state of N with an ε self-loop in N; (c) at least two states such that, in N, there are at least two distinct ε-runs (staying within the equivalence class) between any two states in the equivalence class (thanks to the modification of N described in the preceding paragraph).

Let Z be the states in N'' having the properties specified in (b) or (c). In N'' there are two possibilities. Either (i) there is a (run which is a) cycle in N'' having at least one state in Z, or (ii) each run in N'' goes through at most k states in Z (k is bounded by the number of states in N''). In case (i), N is exponentially ambiguous, since we have at least two ε-runs in N between any two states in an equivalence class in Z. In case (ii), the number of acceptance runs in N'' (by definition without ε-cycles) and number of acceptance runs in N, differ by a constant factor, and we can apply the ε-cycle free result from [1] to N''. □

Theorem 20. *For a pNFA A we can decide in time $O(|A|_E^3)$ if A has finite, polynomial or exponential failure backtracking, and in the case of polynomial failure backtracking, the degree of backtracking can be computed in time $O(|A|_E^3)$.*

Proof. Recall, A^f is the pNFA obtained from A where we change all states of A so that they are not accepting, and $\overline{A}^f$ denotes the NFA obtained by ignoring priorities on transitions of A^f. For an NFA N, $a(N)$ is obtained from N by adding a new accepting sink state z (having transitions to itself on all input letters), all other states in N are made non-accepting, and we add ε-transitions from all states in N to z. Since $da(a(\overline{A}^f), w) = |btr_{A^f}(w)|$, and thus $\max\{da(a(\overline{A}^f), w) \mid w \in \Sigma^*, |w| \leq n\} = \max\{|btr_{A^f}(w)| \mid w \in \Sigma^*, |w| \leq n\}$, the failure backtracking complexity of A is equal to the ambiguity of $a(\overline{A}^f)$. To complete the proof, apply Theorem 19 to $a(\overline{A}^f)$. □

6 Conclusion/Future Work

Our prioritized NFA model is the only automata model, that we are aware of, which formalizes backtracking regular expression matching. This model is well suited to be extended to describe notions such as possessive quantifiers, captures and backreferences found in practical regular expressions. Backreferences have been formalized in [3], but without eliminating ambiguities due to multiple matches. Trying to improve our current complexity result for deciding backtracking complexity (as in Definition 6), and secondly, to formalize what is meant by equivalence of a regular expression with an pNFA, will provide the impetus for future investigations.

References

[1] Cyril Allauzen, Mehryar Mohri & Ashish Rastogi (2008): *General Algorithms for Testing the Ambiguity of Finite Automata.* In Masami Ito & Masafumi Toyama, editors: *Developments in Language Theory*, *Lecture Notes in Computer Science* 5257, Springer, pp. 108–120.

[2] Robert S. Boyer & J. Strother Moore (1977): *A fast string searching algorithm.* *Communications of the ACM* 20(10), pp. 762–772.

[3] Cezar Câmpeanu, Kai Salomaa & Sheng Yu (2003): *A Formal Study of Practical Regular Expressions.* *International Journal of Foundations of Computer Science* 14(06), pp. 1007–1018.

[4] Frank Drewes (2001): *The Complexity of the Exponential Output Size Problem for Top-Down and Bottom-Up Tree Transducers,.* *Information and Computation* 169(2), pp. 264 – 283.

[5] Bryan Ford (2004): *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation.* In: *Symposium on Principles of Programming Languages (POPL'04)*, ACM Press, pp. 111–122.

[6] Michael R. Garey & David S. Johnson (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA.

[7] James Kirrage, Asiri Rathnayake & Hayo Thielecke (2013): *Static Analysis for Regular Expression Denial-of-Service Attacks.* In: *Network and System Security*, Springer, pp. 135–148.

[8] Sérgio Medeiros, Fabio Mascarenhas & Roberto Ierusalimschy (2012): *From Regexes to Parsing Expression Grammars.* *CoRR* abs/1210.4992.

[9] Ken Thompson (1968): *Regular Expression Search Algorithm.* *Commun. ACM* 11(6), pp. 419–422.

A More Detailed Proofs

In this appendix, we collect some proofs that had to be omitted or substantially shortened in the main part of the paper. For convenience, we re-state the results these proofs belong to.

Lemma 10. *Let σ be a function on trees such that, for $t = f[t_1,\dots,t_k]$*

$$\sigma(t) = \begin{cases} t & \text{if } k = 0 \\ f[\sigma(t_1)] & \text{if } k = 1 \\ f[\sigma(t_i),\sigma(t_j)] & \text{otherwise, where } t_i, t_j\ (i \neq j) \text{ are largest among } t_1,\dots,t_k. \end{cases}$$

Let $T_0, T_1, T_2, \dots$ be sets of trees of rank at most k. Then the function $f(n) = \max\{|t| \mid t \in T_n\}$ grows exponentially if and only if $f'(n) = \max\{|\sigma(t)| \mid t \in T_n\}$ grows exponentially.

Proof. The *if* direction is obvious. Let us consider the *only if* direction. Without loss of generality, we may assume that every node of a tree in $\bigcup_{n\in\mathbb{N}} T_i$ is either a leaf or has at least two children (i.e. the second case of the definition of $\sigma(t)$ never applies). Now, the implication to be proved is equivalent to saying that $g'(n) = \max\{\ell(\sigma(t))| \mid t \in T_n\}$ grows exponentially if $g(n) = \max\{\ell(t)| \mid t \in T_n\}$ does, where $\ell(t)$ denotes the number of leaves of a tree t. However, among all trees t of rank k with a given number of leaves, the balanced k-ary trees t are those which minimize $\ell(\sigma(t))$. Clearly, for such a tree, assuming for simplicity that it is fully balanced, we have $\ell(\sigma(t)) = \ell(t)^c$, where $c = \log_k 2$, which proves the statement. □

Lemma 14. *For a δ_2-flattened pNFA A, the string-to-tree transducer stt as constructed by Definition 12, and an input string $w = \alpha_1 \cdots \alpha_n$, it holds that $stt(dec(w)) = \{btr_A(w)\}$. For* all *$u$ which are decorations of w either $stt(u) = \emptyset$ or $stt(u) = \{btr_A(w)\}$.*

Proof. First, notice how A being δ_2-flattened impacts btr_A. The flattening ensures that there is no way to take two ε-transitions in a row in A, meaning that every time case 2 of Definition 3 applies, we have $C(q) = 0$ since the previous step is either the initial call or a call from case 1 where C gets reset. As such we will have $C = 0^{Q_2}$ in every recursive call below. Let stt_q denote the string-to-tree transducer stt with the initial state q (instead of q_0).

Let $v = \$\flat\alpha_1\flat\alpha_2\flat\cdots\flat\alpha_n\$$. The proof will simply be a lengthy case analysis. For the most part the cases in the definition of btr_A have a very direct one-to-one relationship with what is done in stt, with some details requiring clarification. We work our way backwards, starting with the subtrees generated when the empty string remains, $w' = \varepsilon$ and $v = \$$.

We start with the case where the backtracking run on w *fails*. We divide this into two cases.

- Let the remaining suffixes of w and v be $w' = \varepsilon$ and $v' = \$$. For every state q of A we have:
  - When $q \in Q_1 \setminus F$, let $t = q[Rej]$. Then we have $btr_A(q, w, 0^{Q_2}) = t$ and $t \in stt_{f_q}(v')$.
  - When $q \in Q_2 \setminus F$ and $q_1 \cdots q_n = \delta_2(q)$, let $t = q[q_1[Rej],\dots,q_n[Rej]]$. Then it holds that $btr_A(q, w', 0^{Q_2}) = t$ and $t \in stt_{f_q}(v')$.

  These are by construction. Notice that in both cases t is the *only* tree in $stt_{f_q}(v)$.
- Let the remaining suffixes of w and v be $w' = \alpha_1 \cdots \alpha_n$ for $n > 0$ and $v' \in \{\alpha_1\flat\cdots\flat\alpha_n\$, \flat\alpha_1\flat\cdots\flat\alpha_n\$\}$. For every state q of A we have:
  - When $q \in Q_1$ and $\delta_1(q,\alpha_1) = q'$ then $btr_A(q, w', 0^{Q_2}) = q[btr_A(q', \alpha_2 \cdots \alpha_n, 0^{Q_2})]$ and, if $t' \in stt_{f_{q'}}(\flat\alpha_2\cdots\flat\alpha_n\$)$ then $q[t'] \in stt_{f_q}(v')$. The rule used in the transducer is given by case 2(a) of Definition 12, preceded by $f_q \xrightarrow{\flat} f_q$ if v' has a leading $\flat$.

- When $q \in Q_1$ and $\delta_1(q,\alpha_1)$ is undefined we have $btr_A(q,w',0^{Q_2}) = q[Rej]$ and $q[Rej] \in stt_{f_q}(v')$. The rule used in the transducer is given by case 2(b) of Definition 12, preceded by $f_q \xrightarrow{\flat} f_q$ if v' has a leading $\flat$.
- When $q \in Q_2$ and $\delta_2(q) = q_1 \cdots q_n$, since we assume the backtracking run to fail all q_i paths will fail and we will have $btr_A(q,w',0^{Q_2}) = q[btr_A(q_1,w',0^{Q_2}),\ldots,btr_A(q_n,w',0^{Q_2})]$. In the transducer a rule from case 3 is applied to get $q[t_1,\ldots,t_n] \in stt_{f_q}(v')$ where $t_i \in stt_{f_{q_i}}(\alpha_1\flat\cdots\flat\alpha_n)$ for each i. Notice that here a leading $\flat$ in v is *required*, but one will always be available (as we cannot have two Q_2 states in a row due to δ_2-flattening). Since none of the q_i are in Q_2 the next step will read the leading α_1 and a new $\flat$ will be available in the next recursive step.

Notice that each step in *stt* creates precisely one tree if only a single subtree is the recursive call generates only a single tree, and as shown above the base case produces only a single tree. As such $t \in stt_{f_q}(v)$ is by induction true for exactly one tree for each (properly decorated) v. This property will be maintained for a_q states as well.

This establishes that for rejecting backtracking runs $t = btr_A(q,w,0^{Q_2})$, we have $t \in stt_{f_q}(v)$, for all q, where $v = dec(w)$ with the initial \$ removed (we will deal with this at the end) and, vice versa, $t \in stt_{f_q}(v)$ is true for exactly one t, so t must be the tree $btr_A(q,w,0^{Q_2})$.

The proof for the accepting runs follows very similar lines, but with the extra wrinkle of how Q_2 rules are handled when some path accepts. The invariant that $t \in stt_{a_q}(v)$ is true for at most one t is maintained however, as is, of course, the parallel to btr_A. Take a w on which the backtracking run of A *succeeds*. We divide this into the same two cases.

- Let the remaining suffixes of w and v be $w' = \varepsilon$ and $v' = \$$ respectively. For every $q \in F$ we have $t = q[Acc]$ and $btr_A(q,w',0^{Q_2}) = t$ and $t \in stt_{a_q}(v')$. This completes all cases for the empty input string.
- Let the remaining suffixes of w and v be $w' = \alpha_1 \cdots \alpha_n$ for $n > 0$, and $v' \in \{\flat\alpha_1\cdots\flat\alpha_n\$, \alpha_1\flat\cdots\flat\alpha_n\$$ respectively. For every state q of A we have:
  - The case where $q \in Q_1$ and $\delta_1(q,\alpha_1) = q'$ is precisely like the failing case except with a_q and $a_{q'}$ in place of f_q and $f_{q'}$.
  - The case where $q \in Q_1$ but $\delta_1(q,\alpha_1)$ cannot give rise to a backtracking run that succeeds.
  - When $q \in Q_2$ and $\delta_2(q) = q_1 \cdots q_n$ we get something slightly more complex. One of the paths q_i will accept, by assumption, so $btr_A(q,w',0^{Q_2}) = q[btr_A(q_1,w',0^{Q_2}),\ldots,btr_A(q_i,w',0^{Q_2})]$ by construction. Case 3 of Definition 12 makes it possible to generate any $q[f_{q_1},\ldots,f_{q_{i-1}},a_{q_i}]$, so here $q[t_1,\ldots,t_i] \in stt_{a_q}(v')$ with $t_j \in stt_{f_{q_j}}(\alpha_1\flat\cdots\flat\alpha_n\$)$ for $j \in \{1,\ldots,i-1\}$ and $t_i \in stt_{a_{q_i}}(\alpha_1\flat\cdots\flat\alpha_n\$)$. This is in fact the *only* possible case, as the transducer, as here sketched out, can only successfully complete its computation from a state a_q if q eventually accepts the string, and can only complete its computation from a state f_q state if q fails.

Again, this shows that $stt_{a_q}(v)$ outputs precisely one tree if v is $dec(w)$ with the initial \$ removed. That initial \$ is now used by the initial rules in *stt*: $q_0' \xrightarrow{\$} a_{q_0}$ and $q_0' \xrightarrow{\$} f_{q_0}$. This means that *stt* produces exactly one tree for every $dec(w)$, and in both the accepting and rejecting case it matches the tree from btr_A.

Finally, we need to deal with *incorrect* decorations. Let v be a decoration of w which is not $dec(w)$. If v has no leading \$, or no trailing \$, or has a \$ in any other position, $stt(v) = \emptyset$, since *stt* has no other possible rules for \$. If v contains *extraneous* $\flat$ we still have $stt(v) = \{btr_A(w)\}$, since they will just be consumed by $q \xrightarrow{\flat} q$ rules. If some $\flat$ is "missing" compared to $dec(w)$ this either causes $stt(v) = \emptyset$, if a Q_2 rule needed it, or $stt(v) = \{btr_A(w)\}$, if it is just removed by a $q \xrightarrow{\flat} q$ rule anyway. □

V

Characterizing Non-Regularity

Martin Berglund

Umeå University, Sweden
mbe@cs.umu.se

In collaboration with Henrik Björklund and Frank Drewes.

Abstract. This paper considers a characterization of the context-free non-regular languages, conjecturing that there for all such languages exists a fixed string that can be pumped to exhibit infinitely many equivalence classes. A proof is given only for a special case, but the general statement is conjectured to hold. The conjecture is then shown to imply that the shuffle of two context-free languages is not context-free.

1 Introduction

This paper is concerned with the characterization of context-free languages, a subject with a long and interesting history. The context-free class of languages is after all very important, it is a both large and interesting class for which parsing is reasonably efficient both in theory and practice. See e.g. [HMU03] for the basics on context-free languages. We will assume a passing familiarity with formal language theory and will only recall the most important definitions we need.

Specifically, we will consider a conjecture, considered likely to be true by the author, which characterizes the *non-regular* context-free languages. Consider Figure 1. Where classic characterizations such as Parikh's theorem and the context-free pump-

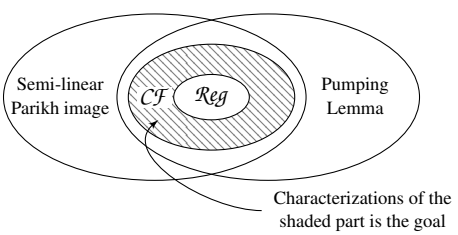


Fig. 1: A Venn diagram illustrating what is being looked at. The set of languages with semi-linear Parikh images are a (strict) superset of the context-free ($\mathcal{CF}$), the languages that fulfill the context-free pumping lemma form a (disjoint and strict) superset of the context-free. The conjecture here studied will restrict itself to just the context-free (by assumption) but will characterize the non-regular (non-$\mathcal{Reg}$) part of the context-free.

ing lemma state requirements that all context-free languages fulfill, this conjecture instead precisely defines which context-free languages are non-regular.

Stating and proving some properties about a conjecture characterizing the inner border of that shaded area, $\mathcal{CF} \setminus \mathcal{Reg}$ in Figure 1, is one part of this paper. The other proves a result that was part of the original motivation for this conjecture, relating to the shuffle. The shuffle of two languages consists of all strings that can be constructed by picking one string from each language and interleaving them. It seems highly probable that the shuffle of any two context-free languages should produce a language which is *not* context-free, *unless* one of those two original context-free languages was actually regular. A more specialized case of this statement will be proven to be true if the conjecture holds.

Overview of the paper. All this will become clearer once some baseline definitions are given, so the intuition about the conjecture is continued in Section 3. In Section 4 the conjecture itself is introduced. In Section 6 the impact it would have on the shuffle of context-free languages if true.

Acknowledgments and citations. The author is indebted to his advisors Henrik Björklund and Frank Drewes for input on the contents of this paper. As most bibliography formats lack a more appropriate field it is suggested that they are listed as author on citations of this paper.

2 Basic Notation

Let $\mathbb{N} = \{0, 1, \ldots\}$. Let ε denote the empty string. Let Σ denote an arbitrary alphabet (finite set of symbols). L^* denotes the Kleene closure (i.e. Σ^* is the language of all strings). We let $w^k = ww^{k-1}$ with $w^0 = \varepsilon$ (i.e. the concatenation of k copies of a string). For a string $w = \alpha_1 \cdots \alpha_n$ (with all $\alpha_i \in \Sigma$) let $w(i) = \alpha_1 \cdots \alpha_i$ and $w(i, j) = \alpha_i \cdots \alpha_j$. As already seen we let $\mathcal{Reg}$ denote the regular languages and $\mathcal{CF}$ denote the context-free languages. We let $\mathcal{L}(A)$ denote the language generated by a grammar/automaton A. $L \propto L'$ denotes the left quotient:

Definition 2 For languages $\mathcal{L}, \mathcal{L}' \subseteq \Sigma^*$ the left quotient $\mathcal{L} \propto \mathcal{L}'$ is defined by letting $w \in \mathcal{L} \propto \mathcal{L}'$ if and only if $vw \in \mathcal{L}'$ for some $v \in \mathcal{L}$. $\diamond$

To simplify the notation in some common cases we, for a string $u \in \Sigma^*$, write just u to mean the language $\{u\}$ in this context. For example $u \propto \mathcal{L}$ and $\mathcal{L} \propto u$ are equivalent to $\{u\} \propto L$ and $\mathcal{L} \propto \{u\}$ respectively. Let $w \equiv_{\mathcal{L}} v$ if and only if $w \propto \mathcal{L} = v \propto \mathcal{L}$, the subclasses of strings induced by $\equiv_{\mathcal{L}}$ are called the *equivalence classes* of $\mathcal{L}$. The relation $\equiv_L$ is also known as the *right congruence* induced by L.

3 Further Introductory Discussion

Among the characterizing results about context-free languages there are few better known than the context-free pumping lemma [BHPS61]. Let us briefly recall it.

Lemma 3 For every context-free language $L \subseteq \Sigma^*$ there exists some constant $k \in \mathbb{N}$ such that every string $w \in L$ with $|w| > k$ can be split into strings $x, v, y, w, z \in \Sigma^*$ (i.e. $w = xvywz$) such that

1. the middle piece is not too long, $|vyw| \leq k$,
2. we can repeat v and w arbitrarily many times, $xv^i yw^i z \in L$ for all $i \in \mathbb{N}$,
3. the repetition part is non-empty, $|vw| > 0$. ◇

As illustrated in Figure 1 this lemma *encapsulates* context-free languages neatly, proving for example that $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free is straightforward as requirements 2 and 3 mean at least one a, b or c must be repeatable, but requirement 1 ensures that we cannot fit at least one each of all three symbols into vw without making the middle too long.

However, what this lemma does *not* tell us is (at least) two-fold. Firstly, for every language L the language Laa^* fulfills the lemma. That is, arbitrarily complex languages can fulfill the lemma if you "attach" them to a language that fulfills it. Secondly, and more importantly for our concerns here, the lemma does not tell us if the language is "just" regular. That is, if the lemma is fulfilled using e.g. $w = \varepsilon$ then it is equivalent to the pumping lemma for regular languages. As shown in Figure 1 we are aiming to instead make a statement about the *non-regular* context-free languages.

The characterization through the semi-linear Parikh image [Par66] is of course similar in the first respect, $L\Sigma^*$ has a semi-linear Parikh image for all L, and the statement of the theorem is precisely that context-free and regular languages have the same class of Parikh images.

It is important to note that there of course *are* strict characterizations of the $\mathcal{CF} \setminus \mathcal{Reg}$ class, simply stating "a language L such that a context-free grammar G exists with $\mathcal{L}(G) = L$ but no finite automaton A exists with $\mathcal{L}(A) = L$" is sufficient. The perhaps most used characterization for $\mathcal{CF} \setminus \mathcal{Reg}$ is to make use of the fact that infinitely many equivalence classes exist, in effect using the converse of the statement of the Myhill-Nerode theorem (see e.g. [HMU03]) to characterize non-regularity.

This is however a matter of how *helpful* the statement is. In short, what we are aiming for is the intuition that every non-regular context-free language has a choice of x and v in the pumping lemma (as sketched in Definition 3) for which the set of choices for y, w, z that fulfill the lemma is *non-empty*, but all have $w \neq \varepsilon$. That is, if the language is non-regular then there exists a choice of string to repeat which *must* be matched by a change in the suffix of the language.

Another way of viewing matters is through the Chomsky-Schützenberger theorem [CS63], which *precisely* defines the context-free languages, basically defining an alternative representation alongside the context-free grammars. To avoid recalling this theorem in full we just note that that the intuition corresponding to the conjecture which follows here would, in terms of the Chomsky-Schützenberger theorem, state that the homomorphism may *not* project away the difference between opening and closing parenthesis in the underlying Dyck language.

4 The Characterizing Conjecture

We now arrive at the main topic of this paper, a conjecture which, if true, characterizes the non-regular context-free languages.

Conjecture 4 For $\mathcal{L} \in \mathcal{CF}$ (over Σ) it holds that $\mathcal{L} \in \mathcal{CF} \setminus \mathcal{Reg}$ *if and only if* there exists some $x, v \in \Sigma^*$ such that for all $n, m \in \mathbb{N}$ with $n \neq m$ it holds that $xv^n \not\equiv_{\mathcal{L}} xv^m$ (i.e., that $(xv^n \propto \mathcal{L}) \neq (xv^m \propto \mathcal{L})$). $\diamond$

Hopefully it is already clear that this statement is of a rather fundamental nature. It characterizes the context-free languages without including the regular languages, in a way that the author believes has not otherwise been done. Unsurprisingly the author is also intuitively convinced that the conjecture holds, as it appears to strike at the core of what context-free languages *do*. It is important to note that this statement is not a trivial consequence of the pumping lemma for context-free languages (or Ogden's lemma or other variations), as those characterize all context-free languages, including the regular. In addition it may be instructive to keep in mind that for $L \in \mathcal{CF}$ it is, in general, undecidable whether $L \in \mathcal{Reg}$, which implies that any proof of Conjecture 4 must be non-constructive.

5 Proving a Fragment of Conjecture 4

There are, luckily, some subclasses of the context-free languages for which it is easier to see that Conjecture 4 holds than it is for the full class. It is, of course, very easy to see that it holds for the language $\{a^n b^n \mid n \in \mathbb{N}\}$, by choosing $x = \varepsilon$ and $v = a$. We can, however, broaden this to a much larger (though somewhat related) subclass of the context-free languages still.

First we recall the definition of push-down automata, mainly in order to name to name some restrictions on the language class precisely.

Definition 5 A *push-down automaton* (PDA) is a tuple $A = (Q, \Sigma, \Gamma, \delta, q_0, \bot, F)$, where

- Q is the finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of final states,
- $\bot \in \Gamma$ is the bottom stack symbol, and, finally,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \times Q \times \Gamma^*$ is the finite set of rules. For all rules $(q, \alpha, \gamma, q', s) \in \delta$ where $\gamma = \bot$ it must hold that $s = s'\bot$ for some $s' \in \Gamma^*$.

The set of configurations of A is $C_A = Q \times (\Gamma^* \cdot \bot)$. A can go from the configuration $(q, \gamma s)$ to $(q', s's)$, for $\gamma \in \Gamma$ and $s, s' \in \Gamma^*$, by reading the symbol α (may be ε in which case nothing is read) if and only if there is a rule $(q, \alpha, \gamma, q', s') \in \delta$. The initial configuration is $(q_0, \bot)$. A *accepts* a string w, i.e. $w \in \mathcal{L}(A)$, if and only if the string

can be read going from the configuration $(q_0, \bot)$ to an accepting configuration. For general push-down automata the accepting configurations are all (q,s) such that $q \in F$ and $s \in \Gamma^*$.

If A is a *stack-emptying PDA* the accepting configurations are the set $\{(q,\bot) \mid q \in F\}$. A is *fully deterministic* if δ is a function $\delta : (Q \times \Sigma \times \Gamma) \to (Q \times \Gamma^*)$. ◇

Next let us note a small, and fairly obvious detail, before we use it in the broader proof. Most readers will probably accept it as true without further argument, but for completeness sake the proof is sketched.

Lemma 6 For every language $L \in \mathcal{CF} \setminus \mathcal{Reg}$ there exists an infinite string ω such that $\omega(i) \not\equiv_L \omega(j)$ for all $i \neq j$. ◇

PROOF: This is necessarily the case, since the Myhill-Nerode theorem dictates that $L \notin \mathcal{Reg}$ if and only if L has infinitely many equivalence classes. Simply construct the tree where the root is marked ε, and for each node v give it a child for each $\alpha \in \Sigma$ such that $v\alpha \not\equiv_L v(i)$ for every i. Each node has a finite number of children, but the infinitely many equivalence classes will necessarily be reached in this fashion, so by Königs lemma some path is infinite, which we take as ω. ■

Theorem 7 If $\mathcal{L}$ is a language accepted by a stack-emptying fully deterministic PDA A then Conjecture 4 holds for $\mathcal{L}$. ◇

PROOF: Clearly, if $\mathcal{L} \in \mathcal{Reg}$ Conjecture 4 holds, since $\mathcal{L}$ has a finite set of equivalence classes according to the Myhill-Nerode theorem, and so for all $x, v \in \Sigma^*$ there must exist $i \neq j$ with $xv^i \equiv_{\mathcal{L}} xv^j$.

The case where $\mathcal{L} \in \mathcal{CF} \setminus \mathcal{Reg}$ remains. Let ω be an infinite word such that $\omega(i) \not\equiv_{\mathcal{L}} \omega(j)$ for all $i, j \in \mathbb{N}$. Let (q_i, s_i) be the configuration A reaches on $\omega(i)$. There will be precisely one since A is fully deterministic. Then let $S : \mathbb{N} \to Q \times \Gamma$ be such that $S(i) = (q_i, \gamma_i)$ where $s_i = \gamma_i s'_i$ for some $\gamma \in \Gamma$. Let $H(i) = |s_i|$ (i.e. the height of the stack of the configuration reached on $\omega(i)$).

All (q_i, s_i) must be different, by the construction of ω and the fact that A accepts L, so H must be ultimately increasing, in the sense that there exists an infinite set $\mathbb{I} \subseteq \mathbb{N}$ such that for all $i \in \mathbb{I}$ and $j > i$ it holds that $H(j) > H(i)$.

Since $\mathbb{I}$ is infinite and $Q \times \Gamma$ finite there must exist some (q, γ) for which $\mathbb{I}_{(q,\gamma)} = \{i \in \mathbb{I} \mid S(i) = (q, \gamma)\}$ is infinite. Let $n = |Q| + 1$ and let $i_1, \ldots, i_n$ be the n smallest elements of $\mathbb{I}_{(q,\gamma)}$.

To give the intuition of the remainder of the proof, we now select a sufficiently long prefix of ω that all of the above configurations are visited. A suffix which is in the language is selected, then a substring of the prefix is found such that pumping it will force an equivalent pumping in the suffix.

Pick any string v such that $\omega(i_n) \cdot v \in \mathcal{L}$ (at least one must exist as $(\omega(i) \propto L) \neq \emptyset$ for all i). Let $w = \omega(i_n) \cdot v$ and let S', H' be functions as above but for running A on w. Let $j_1, \ldots, j_n \in \mathbb{N}$ be such that, for all k, j_k is the smallest number with $j_k > i_k$ and $H'(j_k) \leq H'(i_k)$. More precisely this means that $H'(j_k) = H'(i_k)$, and the configurations reached by A on $w(j_k)$ and $w(i_k)$ have the same stack. This is necessarily the case since A is stack-emptying and a rule can pop at most one stack symbol. Notice

that the j indices end up being "reversed", in that $j_{k+1} < j_k$ for all k. Since $n > |Q|$ the pigeon-hole principle yields that there exists some $p \in Q$ and $k_1, k_2 \in \mathbb{N}$ (with $k_1 < k_2$) such that $(p, \gamma) = S(j_{k_1}) = S(j_{k_2})$. Note that this is the same γ as in $S(i_1)$ through $S(i_n)$.

Next, we partition w into $w = xvyuz$ where

- $x = w(1, i_{k_1})$,
- $v = w(i_{k_1} + 1, i_{k_2})$,
- $y = w(i_{k_2} + 1, j_{k_2})$,
- $u = w(j_{k_2} + 1, j_{k_1})$,
- $z = w(j_{k_1} + 1, |w|)$.

Notice that after A has read x it will be in a configuration of the form $(q, \gamma s)$, and that, by construction, whenever A is in a configuration of the form $(q, \gamma s')$ it can read v to go to a configuration $(q, \gamma s'' \gamma s')$ without ever inspecting s'. In the opposite direction, whenever A is in a configuration of the form $(p, \gamma s'' \gamma s')$ (with the same s' as above) it can read u to go to the configuration $(p, \gamma s')$, without ever inspecting s'. In particular, since $xvyuz \in L$ we have $xv^i yu^i z \in L$ for all $i \geq 1$.

Claim. $xv^i \not\equiv_L xv^j$ for all $1 \leq i < j$.

Assume that there are $1 \leq i < j$ such that $xv^i \equiv_L xv^j$. Let $d = j - i$. Then, since equivalence is closed under concatenation to the right, $xv^i \equiv_L xv^{i+d} \equiv_L xv^{i+2d} \equiv_L xv^{i+3d} \equiv_L \cdots$. However, as argued above, running A on the string xv^{i+cd}, for $c \in \mathbb{N}$, will place it into a configuration

$$(q, \underbrace{\gamma s'' \gamma s'' \cdots \gamma s''}_{i+cd \text{ copies}} \gamma s),$$

which, for a sufficiently large c will give a stack deeper than $|yu^i z|$, but since A must empty its stack before accepting, and must read a symbol for each symbol popped from the stack, this means that $xv^{i+cd} yu^i z \notin L$, but $xv^i yu^i z \in L$, so our assumption that $xv^j \equiv_L xv^i$ must have been incorrect. This contradiction proves the claim, which makes x and v fulfill the conjecture for L. ∎

It seems probable that this proof can be extended to handle slightly less restrictive language classes (notably the stack emptying is a candidate for removal), but introducing full non-determinism appears to require a more advanced approach.

6 Closure Properties of Shuffling Context-Free Languages

To show that Conjecture 4 is not entirely without motivation (although it appears very interesting in its own right), we will here prove a fairly interesting statement about the shuffle of context-free languages that can be made using the conjecture. First let us recall the definition of the *shuffle*, an interleaving operator of great interest in many areas.

Definition 8 For all strings $w = \alpha_1 \cdots \alpha_n$ and $v = \beta_1 \ldots \beta_m$, for $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \in \Sigma$, let $w \odot v$ denote the *shuffle* of w and v. The shuffle is defined by letting $w \odot \varepsilon = \varepsilon \odot w = w$, and letting

$$\alpha_1 \cdots \alpha_n \odot \beta_1 \cdots \beta_m = \alpha_1(\alpha_2 \cdots \alpha_n \odot \beta_1 \cdots \beta_m) \cup \beta_1(\alpha_1 \cdots \alpha_n \odot \beta_2 \cdots \beta_m).$$

For languages $L, L' \subseteq \Sigma^*$ let $L \odot L' = \{w \odot v \mid w \in L, v \in L'\}$. ◇

Next, let us for the sake of completeness recall the statement of Ogden's Lemma [Ogd68], which generalizes the pumping lemma of Definition 3, and which will be useful later.

Lemma 9 If a language $\mathcal{L} \in \Sigma^*$ is context-free there exists a constant $p \in \mathbb{N}$ such that for every string $w \in \mathcal{L}$ and every way of marking at least p of the positions in w there exists a subdivision $p_1 r_1 m r_2 p_2 = w$ such that

1. $r_1 r_2$ contains at least one marked position,
2. $r_1 m r_2$ contains at most p marked positions,
3. $p_1 r_1^i m r_2^i p_2 \in \mathcal{L}$ for all $i \in \{0, 1, \ldots\}$. ◇

We can now go on to prove the following theorem.

Theorem 10 Assume that Conjecture 4 holds. Let a, b be two symbols not in Σ. Let $\mathcal{D} = \{a^n b^n \mid n \in \mathbb{N}\}$. Then for any *context-free* $\mathcal{L} \in \Sigma^*$ it holds that $\mathcal{L} \odot \mathcal{D} \in \mathcal{CF}$ if and only if $L \in \mathcal{Reg}$. ◇

PROOF: If $\mathcal{L} \in \mathcal{Reg}$ it is trivially true that $\mathcal{L} \odot \mathcal{D} \in \mathcal{CF}$, one can directly construct a push-down automaton which recognizes the regular $\mathcal{L}$ entirely in its finite state, and uses the stack to recognize the $\mathcal{D}$ portion.

The remainder of the proof demonstrates the other direction. Using $\mathcal{L}$ let $x, v \in \Sigma^*$ be as in Conjecture 4, which we assumed to be true. Let $\mathcal{L}' = \mathcal{L} \odot \mathcal{D}$ as above. The proof continues by contradiction, let us assume that $\mathcal{L}' \in \mathcal{CF}$ but $\mathcal{L} \in \mathcal{CF} \setminus \mathcal{Reg}$.

Let $\mathcal{R}$ be the regular language corresponding to the regular expression $x(av)^* b^* \Sigma^*$, then let $\mathcal{L}'' = \mathcal{L}' \cap \mathcal{R}$. This $\mathcal{L}''$ must be context-free, since $\mathcal{L}'$ is assumed to be context-free and $\mathcal{CF}$ is closed under intersection with regular languages.

Let p be the constant for which (Ogden's) Lemma 9 holds for $\mathcal{L}''$. For all $\gamma \in \mathbb{N}$ let $W_\gamma = x v^\gamma \propto \mathcal{L}$. Then, for each $\gamma \geq p$ and $w \in W_\gamma$ consider the string $x(av)^\gamma b^\gamma w$, which is in $\mathcal{L}''$ by definition. Apply Lemma 9 by marking all symbols in the substring $(av)^\gamma$ (that is sufficient, as we chose $\gamma \geq p$), and let $p_1 r_1 m r_2 p_2$ be the subdivision that the lemma dictates exists. We know that $r_1 r_2$ must contain at least one symbol from $(av)^\gamma$, and that $p_1 r_1^i m r_2^i p_2 \in \mathcal{L}''$ for all $i \geq 0$ by the lemma, this restricts the choice of r_1 and r_2 severely:

- The language $\mathcal{R}$ used in the construction of $\mathcal{L}''$ ensures that if any symbol in $(av)^\gamma$ is repeated then a complete substring of the form $(av)^k$, for $1 \leq k \leq p$ must be repeated. In actuality $k \leq \left\lfloor \frac{p}{|av|} \right\rfloor$ is dictated by Lemma 9, but we overestimate for simplicity.
- The intersection with the language $\mathcal{R}$ also ensures that neither r_1 nor r_2 can span the border between x and $(av)^\gamma$, the border between $(av)^\gamma$ and b^γ, or the border between b^γ and w, as those borders are distinctly dictated in the regular expression.

- If r_1 contains no a then r_2 must contain the a, but then the repetition $p_1 r_1^i m r_2^i p_2$ increases the number of as without increasing the number of bs, which the shuffled in language $\mathcal{D}$ disallows. As such, r_1 must contain an a, and therefore be of the form $(av)^k$ for some k, as this is the only repetition in this section that $\mathcal{R}$ makes possible.
- This leaves r_2 to be of the form b^k, to preserve the number of bs in correspondence with the number of as.

This leads us to w, which by the above cannot be modified by the pumping $p_1 r_1^i m r_2^i p_2$ (i.e., it necessarily falls entirely within p_2), despite the number of vs changing as part of the pumping of a substring in $(av)^*$. This means that for all $\gamma \geq p$ and $w \in W_\gamma$ (recall, $W_\gamma = xv^\gamma \propto \mathcal{L}$) there exists a constant k such that $xv^{\gamma+ik}w \in \mathcal{L}$ for all $i \geq -1$.

Notice that $p!$ is a multiple of all $1 \leq k \leq p$, meaning that all $w \in W_\gamma$ are such that $xv^{\gamma+i(p!)}w \in \mathcal{L}$ for all $i \geq 0$. It follows from this that $W_\gamma \subseteq W_{\gamma+p!}$, and, since Conjecture 4 is assumed to be true $W_i \neq W_j$ for all $i \neq j$, so the inclusion is strict: $W_\gamma \subsetneq W_{\gamma+p!}$ for all $\gamma \geq p$. Since this holds for all γ we can by simple induction establish that

$$W_p \subsetneq W_{p+p!} \subsetneq W_{p+2(p!)} \subsetneq W_{p+3(p!)} \subsetneq \cdots$$

by simply choosing γ to be p, $p+p!$, $p+2(p!)$ successively. Replacing the uses of W_γ with its left quotient definition we arrive at the statement that

$$(xv^{p+i(p!)} \propto \mathcal{L}) \subsetneq (xv^{p+(i+1)(p!)} \propto \mathcal{L})$$

for all $i \geq 0$. Now, for each $i > 1$ pick some representative $w_i \in (xv^{p+i(p!)} \propto \mathcal{L}) \setminus (xv^{p+(i-1)(p!)} \propto \mathcal{L})$. Notice that this by induction means that $w_i \notin (xv^{p+j(p!)} \propto \mathcal{L})$ for any $0 < j < i$.

Next, let $\hat{\mathcal{R}} = xv^p(av^{p!})^* b^* \Sigma^*$, and construct $\hat{\mathcal{L}} = \hat{\mathcal{R}} \cap (\mathcal{L} \odot \mathcal{D})$. Let $\hat{p}$ be the constant for Lemma 9 for this language. Pick the string $xv^p(av^{p!})^{\hat{p}} b^{\hat{p}} w_{\hat{p}}$. Apply Lemma 9 by marking the substring $(av^{p!})^{\hat{p}}$ and let $p_1 r_1 m r_2 p_2$ be the substring subdivision that the lemma dictates exist. Again notice that r_1 must fall entirely within the substring $(av^{p!})^{\hat{p}}$ due to the intersection with $\hat{\mathcal{R}}$, and r_2 must then fall entirely within the b^* substring (refer to the full argument above). The lemma then further dictates that $p_1 m p_2 \in \hat{\mathcal{L}}$, but this removes (using the same argument for where r_1 and r_2 must be in the original string as above) a non-zero number of the substrings $av^{p!}$ from the string, and by construction $w_{\hat{p}} \notin (xv^{p+k(p!)} \propto \mathcal{L})$ for $k < \hat{p}$.

This contradicts the assumption that $\mathcal{L} \in \mathcal{CF}$ and $\mathcal{L}' \notin \mathcal{Reg}$. ∎

7 Conclusions

We have shown that Conjecture 4 holds for at least one limited case, and proven a follow-up result of an interesting nature. However, the most obvious missing piece is a complete proof of the conjecture, or possibly a counter-example. The author still hopes to manage such a proof, but contributions are most certainly welcome. Secondarily an extension of the proof that the shuffle of a context-free language with $a^n b^n$ (on disjoint alphabets) is context-free if and only if the first language was regular should also be

considered. It appears likely that for all $\mathcal{L}, \mathcal{L}' \in \mathcal{CF}$, on disjoint alphabets, $\mathcal{L} \odot \mathcal{L}' \in \mathcal{CF}$ if and only if either $\mathcal{L} \in \mathcal{Reg}$ or $\mathcal{L}' \in \mathcal{Reg}$. In this case it seems likely that this can be achieved along the lines of the proof of Theorem 10, by making the central claims symmetrical and employing some case analysis, but a counter-example as the final result cannot yet be entirely ruled out.

References

[BHPS61] Yehoshua Bar-Hillel, Micha A. Perles, and Eli Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, (14):143–172, 1961.

[CS63] Noam Chomsky and Marcel Paul Schützenberger. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirshberg, editors, *Computer Programming and Formal Systems*, Studies in Logic, pages 118–161. North-Holland Publishing, Amsterdam, 1963.

[HMU03] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Ed.)*. Pearson Education International, 2003.

[Ogd68] William Ogden. A helpful result for proving inherent ambiguity. *Mathematical systems theory*, 2(3):191–194, 1968.

[Par66] Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

VI

Analyzing Edit Distance on Trees: Tree Swap Distance is Intractable

Martin Berglund

Department of Computing Science, Umeå University
90187 Umeå, Sweden
mbe@cs.umu.se

Abstract. The string correction problem looks at minimal ways to modify one string into another using fixed operations, such as for example inserting a symbol, deleting a symbol and interchanging the positions of two symbols (a "swap"). This has been generalized to trees in various ways, but unfortunately having operations to insert/delete nodes in the tree *and* operations that move subtrees, such as a "swap" of adjacent subtrees, makes the correction problem for trees intractable. In this paper we investigate what happens when we have a tree edit distance problem with *only* swaps. We call this problem tree swap distance, and go on to prove that this correction problem is NP-complete. This suggests that the swap operation is fundamentally problematic in the tree case, and other subtree movement models should be studied.

1 Introduction

String edit distance is an old, well-known and thoroughly studied concept, most commonly used in the context of *string correction problems*. An edit distance (of which there are many kinds) defines some small set of operations on strings. An instance of the string correction problem corresponding to a given edit distance is a question of the form "can the string s be transformed into s' by applying at most k edit operations?" In more complex cases the string correction problem may associate different costs to the edit operations, having k serve as a total budget.

One of the most frequently used types of edit distance is Levenshtein distance [7], which features the three operations `delete`, `insert`, and `replace`. These can be applied to any position in a string, to delete a single symbol, insert a single symbol, and replace a single symbol by another, respectively. A popularly applied extension, called Damerau-Levenshtein distance [3], adds a fourth operation, `swap`, which swaps the position of any two symbols in a string. For both of these distances the string correction problem is very efficiently solvable if all operations have the same cost. A more general variant is called the *extended string-to-string correction problem*, which uses the four Damerau-Levenshtein operations, but allows the problem instance to assign each operator an arbitrary integer cost [11]. In general this makes the correction problem strongly NP-complete [10], a fact that we will make use of later.

As this area is well-explored and successful in the string case it is of great interest to extend the same ideas to the tree case [8, 9]. This work has been very successful for the "insert", "delete" and "replace" operations, but the "swap" operation has most often been left out [12, 5, 2]. This is in fact a necessity, as the problem quickly becomes intractable when subtree movement is introduced as an operation. This follows trivially from the fact that tree edit distance on unordered trees is NP-complete [13], by duplicating nodes one can create a situation where the swaps are so much cheaper than

a `delete`/`insert` operation that the problem becomes equivalent to the unordered one. Still, swaps and other subtree movement operations remain very interesting in practice in very diverse fields such as XML processing, computational biology, natural language processing and many others. Approximations have been considered, for example [1] introduces swaps into tree edit distance but the algorithm as given actually restricts each node to participate in at most one swap, so arbitrary reorderings are not possible.

While much work has been done to restrict the swaps to make the problem tractable we will here instead take a step back and consider the "tree swap distance" problem. In this restriction of tree edit distance *only* the `swap` operation is allowed, reducing the problem to finding the least number of swaps necessary to reorder one tree into another. Unfortunately the end result is that we demonstrate that even this problem is NP-complete, suggesting that the `swap` operation may be a computationally bad choice to model subtree movement operations.

2 Preliminaries

Let $\mathbb{N}$ denote the set of natural numbers $\{0, 1, 2, 3, \ldots\}$. For all $n \in \mathbb{N}$ let $[n]$ denote the set $\{1, \ldots, n\}$. An *alphabet* Σ is a finite set of symbols. Going forward we will simply use Σ to mean some appropriate alphabet without specifying it precisely. The empty string/sequence is denoted by ϵ. The set of all strings over an alphabet Σ is denoted Σ^* and is defined as $\Sigma^* = \{\epsilon\} \cup \{\alpha v \mid \alpha \in \Sigma, v \in \Sigma^*\}$. The length of a string $v \in \Sigma^*$ is denoted $|v|$. The set of sequences over an arbitrary set S is also denoted S^*, the sequence $s_1, \ldots, s_n$ is referred to as an n-tuple. When expedient we may abuse notation and confuse the n-tuple $s_1, \ldots, s_n$ with the string $s_1 \cdots s_n$.

An n by n matrix (all our matrices are square) is an n-tuple of n-tuples $M = ((x_{1,1}, \ldots, x_{1,n}), \ldots, (x_{n,1}, \ldots, x_{n,n}))$ with $x_{i,j} \in \mathbb{N}$ for all $i, j \in [n]$. We say that $x_{i,j}$ is on row i and column j, and denote it by $M_{i,j}$.

A tree t consists of a root node labeled by some symbol $\alpha \in \Sigma$ and a tuple of zero or more direct child subtrees $(t_1, \ldots, t_n)$ (for any $n \in \mathbb{N}$) over the same alphabet. t is denoted by $\alpha[t_1, \ldots, t_n]$. For a tree $\alpha[]$ with zero children we may abbreviate it as simply α. The set of all trees over Σ, denoted by T_Σ, is defined as $T_\Sigma = \Sigma \cup \{\alpha[t_1, \ldots, t_n] \mid \alpha \in \Sigma, n \in \mathbb{N}, t_1, \ldots, t_n \in T_\Sigma\}$.

The set of positions in a tree is defined by a function $pos : T_\Sigma \to 2^{\mathbb{N}^*}$. For any $k \in \mathbb{N}$, including zero, $\alpha \in \Sigma$ and $t_1, \ldots, t_k \in T_\Sigma$ the definition of $pos\big(\alpha[t_1, \ldots, t_k]\big)$ is $\{\epsilon\} \cup \big\{(i, v_1, \ldots, v_n) \mid i \in \{1, \ldots, k\}, (v_1, \ldots, v_n) \in pos(t_i)\big\}$. That is, a position $p \in pos(\alpha[t_1, \ldots, t_n])$ denotes the root note α if $p = \epsilon$, otherwise p is of the form $(i, v_1, \ldots, v_n)$ referring to the position $(v_1, \ldots, v_n)$ in the subtree t_i.

3 The Extended String-to-String Correction Problem

A (pre-existing) problem that we will make use of in the coming proof will now be defined. Later on we will use a reduction from an instance of the *extended string-to-string correction problem* (ESSCP) to our problem to show strong NP-hardness. The ESSCP is known to be NP-complete (problem [SR20] in [4]), shown in the case where the cost of inserts and replacements is made infinite and when swaps and deletes are given a constant cost [10]. The formulation by Wagner in [10] allows arbitrary costs

for deletes and any non-zero cost for swaps, while the formulation in [4] fixes both costs to 1. Here we opt to set the cost of a single swap to 1 and the cost of deletes to 0, this causes no loss of generality, since the number of deletes in a solution is always the difference in length between the source and target strings. The problem definition is divided into three parts, for all $\alpha_1 \cdots \alpha_n \in \Sigma^*$:

Definition 1 (String deletes). *For all* $\{d_1, \ldots, d_m\} \subseteq [n]$ *we define the delete function as* $delete(\alpha_1 \cdots \alpha_n, \{d_1, \ldots, d_m\}) = \alpha_{i_1} \cdots \alpha_{i_{n-m}}$ *where* $i_1 < \ldots < i_{n-m}$ *and* $\{i_1, \ldots, i_{n-m}\} = [n] \setminus \{d_1, \ldots, d_m\}$.

Definition 2 (String swaps). *We define the swap function by letting* $\texttt{swap}(s, \epsilon) = s$ *for all strings* s *and for all* $(s_1, \ldots, s_m) \in [n-1]^*$ *letting*

$$\texttt{swap}(\alpha_1 \cdots \alpha_n, (s_1, \ldots, s_m)) = \texttt{swap}(\alpha_1 \cdots \alpha_{s_1-1}\alpha_{s_1+1}\alpha_{s_1}\alpha_{s_1+2} \cdots \alpha_n, (s_2, \ldots, s_m)).$$

Definition 3 (The delete/swap ESSCP). *An instance of the delete/swap ESSCP (over some alphabet* Σ*) is a tuple* $(S, T, b) \in \Sigma^* \times \Sigma^* \times \mathbb{N}$. *The instance is a "yes" instance (the answer is "yes") if and only if there exists some* $D \subseteq [|S|]$ *and* $W \in [|S| - |D| - 1]^*$ *such that* $\texttt{swap}(\texttt{delete}(S, D), W) = T$ *with* $|W| \leq b$. *We denote the set of all such "yes" instances* ESSCP_{ds}.

There are a couple of important things to notice here.

- The definition is stated so that all deletes happen before any swap. This is not a restriction of the problem, since there is no instance where it is better to delete something after moving it around.
- b is in all interesting instances polynomial in the size of the instance, since all reorderings can be realized in less than n^2 swaps. We therefore, without loss of generality, assume b to be coded in unary in the input, so ESSCP_{ds} is strongly NP-complete.
- Swaps of unrelated symbols can be reordered freely. One recurring example is that if $\texttt{swap}(\alpha_1 \cdots \alpha_n, W)$ is such that the symbol α_i is moved to the end of the string by W we can trivially restructure W to start with the sequence $i, i+1, \ldots, n-1$, without making W longer. That is, if a minimal swap sequence moves the symbol in position i to the last position n then doing this before anything else cannot make the swap sequence longer, since keeping the symbol in the middle of the string for longer serves no purpose.

4 Swap Assignment Problem

Now we will define the first original problem, the swap assignment problem. We will demonstrate that this problem is strongly NP-complete by a reduction from ESSCP_{ds}. This problem will serve as a stepping stone to demonstrate NP-completeness for the tree swap distance problem.

This problem is quite similar to the classical assignment problem [6], except a starting assignment is given, and an optimal assignment is to be reached by swapping adjacent assignments. The swap function is defined exactly as in the string case, when the matrix is viewed as a string of rows.

Definition 4 (Matrix Row Swap). *For an n by n matrix M the swap function is defined by for all $W \in [n-1]^*$ simply viewing the matrix as a string of rows: $(M_{1,1}, \ldots, M_{1,n}) \cdots (M_{n,1}, \ldots, M_{n,n})$ and applying the string swap $\texttt{swap}(M, W)$.*

Definition 5 (The Swap Assignment Problem). *An instance of the swap assignment problem is a tuple (M, b) where $b \in \mathbb{N}$, and M is an n by n matrix. The instance is a "yes" instance if and only if there exists some $W \in [n-1]^*$ such that*

$$b \geq |W| + \sum_{i=1}^{n} \texttt{swap}(M, W)_{i,i}.$$

We denote the set of all such "yes" instances SAP.

Let us look at a small instance to better understand the problem.

Example 6. As an example swap assignment problem instance we can take (M, b) with $b = 9$ and M as below.

$$M = \begin{bmatrix} 4\,5\,16\;\;0 \\ 3\,4\,16\;\;0 \\ 2\,3\;\;0\;\;16 \\ 1\,2\,16\,16 \end{bmatrix} \quad M' = \begin{bmatrix} 4\,5\,16\;\;0 \\ 1\,2\,16\,16 \\ 2\,3\;\;0\;\;16 \\ 3\,4\,16\;\;0 \end{bmatrix}.$$

Since we can use the swaps $W = 3, 2, 3$ to construct $M' = \texttt{swap}(M, W)$ as shown above, it follows that $(M, b) \in$ SAP. M' has the diagonal sum 6 which together with the three swaps adds up to exactly 9. We could also equivalently solve the problem instance using the swap-sequence $W' = 1, 3, 2, 3$ which produces a diagonal cost of $3 + 2 + 0 + 0 = 5$ but, on the other hand, requires 4 swaps, again giving a total of 9.

The ESSCP_{ds} (Definition 3) can be reduced to the swap assignment problem in a slightly tricky to visualize but functionally straightforward way.

Definition 7 (ESSCP to Swap Assignment Reduction). *Take a delete/swap ESSCP instance $(s_1 \cdots s_n, t_1 \cdots t_m, b)$ (we assume that $m \leq n$, otherwise it is trivial). Then construct a swap assignment problem instance (M, b') where the n by n matrix M is constructed by taking:*

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq m \text{ and } s_i = t_j, \\ b' + 1 & \text{if } j \leq m \text{ and } s_i \neq t_j. \\ n + i - j & \text{if } j > m, \end{cases},$$

and $b' = b + n(n - m)$.

This definition is not really intuitive, but a short example should explain the idea of how this represents an ESSCP instance.

Example 8. Let us consider the delete/swap ESSCP instance $(aacb, abc, 1)$. This has a fairly simple solution, delete one of the "a" symbols and swap the "b" and "c". The reduction computes $b' = 1 + 4(4 - 3) = 5$ and the matrix

$$M = \begin{bmatrix} 0\,6\,6\,1 \\ 0\,6\,6\,2 \\ 6\,6\,0\,3 \\ 6\,0\,6\,4 \end{bmatrix}.$$

We will look at the left part first, the part that corresponds to the first two cases of the construction. All these cells are set either to 0 or to $b' + 1$, which means that none of the non-zero cells *may ever* be on the diagonal of a solution, since the sum would always be greater than the budget. So, the first three positions on the diagonal (counting from the upper left) must be made zero in a solution, the three corresponds to the length of the target string. The idea is that a zero on the diagonal in this first part corresponds to a correctly matched symbol. The cells on the right-hand side only come into play on the last part of the diagonal, the bottom few rows of the result. The rows moved to the bottom correspond to symbols that get deleted.

The motivation for the weight $n + i - j$ in case 3 of the reduction is that if we wish to delete some symbol in the original string problem we have a fixed cost (zero), but to move a row to the bottom of the matrix has different cost depending on where the row starts out, since different numbers of swaps need to be used. The cost the rows that end up at the bottom contribute to the diagonal is there to counteract this. Let us look at the two ways to solve this instance, see Figure 1. Here we show the

$$\begin{bmatrix} 0&6&6&1\\ 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&2\\ 0&6&6&1\\ 6&6&0&3\\ 6&0&6&4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&2\\ 6&6&0&3\\ 0&6&6&1\\ 6&0&6&4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4\\ 0&6&6&1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&2\\ 6&0&6&4\\ 6&6&0&3\\ 0&6&6&1 \end{bmatrix}$$

Figure 1: A solution for the the swap assignment problem instance produced by reducing from $(aacb, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$

solution equivalent to deleting the first "a", by swapping the top row down to the bottom with the first three swaps. This row then contributes cost 1 to the diagonal, for a total cost of 4 to get rid of the first symbol. Then we swap the rows that were originally 3 and 4 (going from "acb" to "abc") to move the zeros to the diagonal. The total cost of the solution is 5, which fits the budget b'.

What is key is that the solution can choose to delete any symbol without the cost being different. So let us look at the other possibility, where we delete the second "a" instead, shown in Figure 2. Here we start by swapping the second row, corresponding

$$\begin{bmatrix} 0&6&6&1\\ 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&1\\ 6&6&0&3\\ 0&6&6&2\\ 6&0&6&4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&1\\ 6&6&0&3\\ 6&0&6&4\\ 0&6&6&2 \end{bmatrix} \Rightarrow \begin{bmatrix} 0&6&6&1\\ 6&0&6&4\\ 6&6&0&3\\ 0&6&6&2 \end{bmatrix}$$

Figure 2: An alternative solution for the swap assignment problem instance produced by reducing from $(aacb, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$

to the second "a" into the last position. This takes only 2 swaps, but this row contributes a cost of 2 to the diagonal, again making the delete cost exactly 4. A final swap of the original row three and four again produces a solution with cost 5.

This illustrates the key property of the construction, deletions are substituted with moving the rows in question into bottom positions, and the costs in the rows are

constructed so that a row that is originally far from the bottom gets a proportionally larger "discount" on the diagonal sum to pay for the extra swaps needed to delete them. The formula for the rightmost column is $n + i - j$, the subtraction of j comes into play when multiple symbols are deleted. Since not all rows can go to the bottom position later deletions will have a shorter distance to travel than the first ones, this is counteracted by the costs being greater in the "discount columns" further left. As a final example see the slightly larger instance in Figure 3.

$$\left[\begin{array}{ccc|cc} 12 & 12 & 0 & 2 & 1 \\ 12 & 0 & 12 & 3 & 2 \\ 0 & 12 & 12 & 4 & 3 \\ 0 & 12 & 12 & 5 & 4 \\ 12 & 12 & 0 & 6 & 5 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|cc} 0 & 12 & 12 & 4 & 3 \\ 12 & 0 & 12 & 3 & 2 \\ 12 & 12 & 0 & 6 & 5 \\ 12 & 12 & 0 & 2 & 1 \\ 0 & 12 & 12 & 5 & 4 \end{array}\right]$$

Figure 3: Reducing $(cbaac, abc, 1) \in \mathrm{ESSCP}_{\mathrm{ds}}$ produces the swap assignment problem instance with the left matrix and budget $b' = 11$. "Deleting" a row ends up with a cost of 5 counting swaps and diagonal cost. On the right is the solution which performs the swaps $4, 1, 2, 3, 1$ for a total cost of 11. This solution corresponds to deleting the last "a", deleting the first "c" and finally swapping the remaining "b" and "a".

Lemma 9. *The reduction in Definition 7 produces a swap assignment problem instance that answers "yes" if and only if the original delete/swap ESSCP instance answers "yes".*

Proof (Sketch). Starting with the "if" direction, take some $(s_1 \cdots s_n, t_1 \cdots t_m, b) \in \mathrm{ESSCP}_{\mathrm{ds}}$. Let the deletes and swaps that solves this instance be $\{d_1, \ldots, d_{n-m}\} \subseteq [n]$ and $W \in [m-1]^*$. Construct (M, b') using the reduction. Assume that $d_1 > d_2 > \cdots > d_{n-m}$ then construct the swaps:

$$W_d = d_1, d_1 + 1, \ldots, n-1, d_2, d_2 + 1, \ldots, n-2, \ldots, d_{n-m}, \ldots, m$$

That is, take row d_1, which corresponds to the last (position-wise) symbol deleted in the original string, and swap it into the last position in the matrix. Then swap row d_2 (second to last deleted position) into the second to last position in the matrix and so on. Now construct $W' = W_d W$ (concatenating the two), after applying the swaps W_d the top m rows in the matrix correspond to the positions which are *not* deleted, and we perform the swaps in W on these.

Now we will just demonstrate that $(M, b') \in \mathrm{SAP}$ using W' as the solution. $|W'| = |W_d| + |W|$ and $|W_d|$ contains $(n-i) - d_i$ swaps to place the row initially at d_i into position $n-i$, for each $i \in [n-m]$. So the row (initially at) d_i will contribute $M_{d_i, n-i}$ to the final diagonal sum. The range of i means that $M_{d_i,n-i} = n + d_i - (n-i) = d_i + i$ (since all these positions are filled by the third case in the construction of M in Definition 7). Taking the swaps and diagonal contribution together each of the d_i rows contribute to the total cost by $(n-i) - d_i + d_i + i = n$, meaning that

$$|W_d| + \sum_{i=m+1}^{n} \mathtt{swap}(M, W')_{i,i} = (n-m)n.$$

This establishes that $b' = b + (n-m)n \geq |W'| + \sum_{i=m+1}^{n} \texttt{swap}(M, W')_{i,i} = |W| + (n-m)n$, since $b \geq |W|$ and $|W'| = |W_d| + |W|$.

All that needs to be added is the remainder of the diagonal, so next we show that $\sum_{i=1}^{m} \texttt{swap}(M, W')_{i,i}$ is zero. Take $M' = \texttt{swap}(M, W_d)$ and $S' = \texttt{delete}(s_1 \cdots s_n, D)$ and simply note that if the symbol in position i in S' started out in position l then row i in M' started out in position l in M. The next step for both S' and M' is to apply W, meaning that row $j \in [m]$ in the matrix started out as row i if and only if symbol in position j in the final string was originally s_i. Since this is a solution for the ESSCP instance this means that $s_i = t_j$ which means that row i in M ends up in position j in $\texttt{swap}(M, W')$ if and only if $s_i = t_j$. It follows that the new row contributes $M_{i,j}$ to the diagonal, and the construction of M is such that set $M_{i,j} = 0$ when $s_i = t_j$.

Since we showed that $b' \geq |W'| + \sum_{i=m+1}^{n} \texttt{swap}(M, W')_{i,i}$ above and showed that $\sum_{i=1}^{m} \texttt{swap}(M, W')_{i,i} = 0$ here it follows that $b' \geq |W'| + \sum_{i=1}^{n} \texttt{swap}(M, W')_{i,i}$ so $(M, b') \in \text{SAP}$.

The "only if" direction remains but works in a very similar way. Assume that $(M, b') \in \text{SAP}$ is constructed from some delete/swap ESSCP instance (S, T, b). Let W' be the swaps that solve (M, b'). Notice that if such a solution W' exists then a solution exists which has the structure $W' = W_d W$ (that is, which first swaps all the $n - m$ bottom rows into position), if row i is going to be swapped into position n nothing can be gained by not doing so as the first thing in the swap sequence. Using this we can extract the solution to the string problem instance, deleting the symbols corresponding to rows swapped below the mth row. The solution to (M, b') also cannot do better than the fixed cost $(n-m)(n-1)$ for swaps and diagonal of these bottom rows, and it has to place the top m rows so that they all contribute zero to the diagonal (all other positions being $b' + 1$ which is impossible in a solution), which corresponds directly to matching symbols correctly. □

Corollary 10. *The swap assignment problem is strongly NP-complete.*

This follows since ESSCP_{ds} is strongly NP-complete and the reduction constructs a polynomially sized matrix containing numbers that are all bounded by a polynomial in the original instance (recall that b is polynomial in all relevant cases and assumed to be unary). The problem is in NP since no swap sequence ever needs to be longer than n^2, allowing W' to be guessed.

5 Swap Even-Cost Assignment Problem

Now we will define a very minor restriction on the swap assignment problem. This will turn out to be key to make the final reduction to the tree swap distance problem simple.

Definition 11. *Let $2 \,|\, x$ denote that x is even ($x \in \{0, 2, 4, 6, \ldots\}$), let $2 \nmid x$ denote that x is odd.*

Definition 12 (Swap Even-Cost Assignment Problem). *An instance of the swap even-cost assignment problem is a swap assignment problem instance (M, b) such that $2 \mid M_{i,j}$ for all $i, j \in [n]$. The answer to (M, b) is "yes" if and only if $(M, b) \in \text{SAP}$. We denote the set of all "yes" instances as* SecAP.

We will quickly establish that all swap assignment problem instances have an equivalent swap even-cost assignment problem instance.

Definition 13. *Let* $h(x) = \lceil \frac{x}{2} \rceil$.

Definition 14 (Reducing SAP to SecAP). *Let* (M, b) *be an instance of the swap assignment problem with* M *an* n *by* n *matrix, we then construct* (M', b')*, where* M' *is a* $2n$ *by* $2n$ *matrix, by letting* $b' = b + \frac{n(n-1)}{2}$ *and taking*

$$M'_{i,j} = \begin{cases} M_{i,h(j)} & \text{if } i \leq n,\ 2\nmid j \text{ and } 2|M_{i,h(j)}, \\ b'' & \text{if } i \leq n,\ 2\nmid j \text{ and } 2\nmid M_{i,h(j)}, \\ M_{i,h(j)} - 1 & \text{if } i \leq n,\ 2|j \text{ and } 2\nmid M_{i,h(j)}, \\ b'' & \text{if } i \leq n,\ 2|j \text{ and } 2|M_{i,h(j)}, \\ 0 & \text{if } i > n \text{ and } h(j) = i - n, \\ b'' & \text{if } i > n \text{ and } h(j) \neq i - n, \end{cases}$$

where b'' *is the smallest even number strictly larger than* b'*.*

This definition is also a bit daunting but the underlying thinking is fairly straightforward, let us look at an example.

Example 15. We will start with an instance of the swap assignment problem instance (M, b), where $b = 11$ and M is shown on the left in Figure 4. For this example $b' = 14$,

$$M = \begin{bmatrix} 2 & 3 & 3 \\ 9 & 4 & 12 \\ 1 & 2 & 8 \end{bmatrix} \Rightarrow \begin{bmatrix} 2 & 16 & 16 & 2 & 16 & 2 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 0 & 2 & 16 & 8 & 16 \\ 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{bmatrix}$$

Figure 4: Example of applying the even-cost reduction to a swap assignment problem instance

so $b'' = 16$. Let us look at the upper half of the matrix first. The thing to notice about this part is that for all $i, j \in [n]$ there are for each pair $(M_{2i-1,j}, M_{2i,j})$ only two cases, either the pair is $(M_{i,j}, 16)$ if $M_{i,j}$ was even, or it is $(16, M_{i,j} - 1)$ if $M_{i,j}$ was odd.

This starts making sense when we look at the lower half of the matrix, which is filled with rows such that for each $j \in [n]$ the row at position $n + j$ can only be in either position $2j - 1$ or $2j$ in a valid solution (since that brings the rows zero positions to the diagonal, and b'' is guaranteed to be more than the budget). This means that any valid solution will be structured so that for each $j \in [n]$ one of the positions $2j - 1$ and $2j$ contains the row originally in position $n + j$ (in all other positions it would contribute b'' to the diagonal making the solution impossible) and the other position contains some row originally in the top half (since all rows from the bottom half are already accounted for). The $\frac{n(n-1)}{2}$ part of the budget is exactly enough to pay for the minimal such interspersing (where the row from the top half is the one at the $2j - 1$ position since that is closer).

Let $i \in [n]$ be the initial position of the row from the top that ends up in position $2j-1$ or $2j$, this row is supposed to simulate the cost $M_{i,j}$ on the diagonal. If $M_{i,j}$ is even this is easy, the row can be placed at position $2j-1$ (since it will have $M'_{i,2j-1} = M_{i,j}$), if $M_{i,j}$ contained an odd number however the construction has made $M_{i,2j-1} = b''$, which forces the solution to take an extra swap to bring the row to position $2j$. This extra swap fixes the cost that was lost when the construction rounded down $M'_{i,2j} = M_{i,j} - 1$.

To make this more visual see Figure 5. Since this solution involves a total of

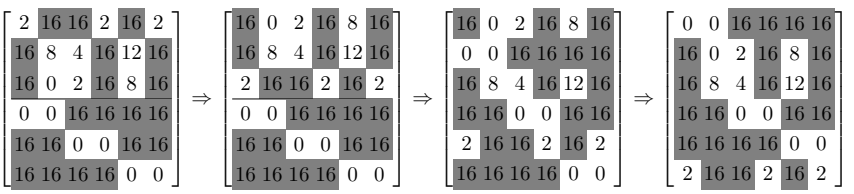


Figure 5: Some steps of the solution of the problem instance in Figure 4

seven swaps several are done in each step. Let us first note that a solution for the original (pre-reduction) instance in Figure 4 is to swap $2, 1, 2$, giving a diagonal sum of $1 + 4 + 3 = 8$ and a total solution cost of 11. In Figure 5 we have the original reduced matrix on the left, in the first step we do the same three swaps $2, 1, 2$. In the next step we intersperse the rows from the bottom half with the top with the swaps $3, 2, 4$. This however leaves us with 16 in two places on the diagonal, and have to finish with the swaps $1, 4$. These last swaps are key. Notice how the diagonal in the original instance ended up being $1 + 4 + 3$, the first and last positions are odd. The construction took these odd numbers, rounded them down to something even and placed this rounded result on the right side of its horizontal "pair" in the top row. This forces the solution to do extra swaps to bring the rows down one step further, paying the cost that was removed by the rounding. In total the solution here makes 8 swaps, and has a diagonal sum of 6, for a total cost of 14, exactly the budget b'.

Lemma 16. *For every swap assignment problem instance (M, b) (M is n by n) the reduction in Definition 14 produces a swap even-cost assignment problem instance (M', b') such that $(M', b') \in$ SecAP if and only if $(M, b) \in$ SAP.*

Proof (Sketch). Assume that $(M, b) \in$ SAP. Let W be a swap sequence that solves (M, b). Then construct a (minimal) swap sequence W_i such that

$$\texttt{swap}(a_1 \cdots a_n b_1 \cdots b_n, W_i) = a_1 b_1 a_2 b_2 \cdots a_n b_n,$$

and, let $W_o = o_1 \cdots, o_m$ be such that $o_1 < \cdots < o_m$ and $2 \nmid \texttt{swap}(M, W)_{i,i}$ if and only if $i \in \{o_1, \ldots, o_m\}$. Then $W' = W W_i W_o$ (the concatenation) is a solution for (M', b'). This sequence of swaps being a solution is quickly established, noting that $|W_i| = \frac{n(n-1)}{2}$ which accounts for the difference between b' and b, and then noting that the construction makes all the swaps in W_o necessary.

The other direction amounts to assuming the existence of W' and then extracting the W part which concerns the internal order of the n first rows. □

Corollary 17. *The swap even-cost assignment problem is strongly NP-complete.*

This follows from the above. The reduction from the strongly NP-complete swap assignment problem is clearly polynomial, the matrix dimensions are doubled and the values in the matrix grow on the order of $\mathcal{O}(n^2)$. The problem is in NP, since SecAP is simply SAP with inputs restricted to even numbers.

6 Tree Swap Distance Problem

This section will reach the goal of the paper, defining the tree swap distance problem and then demonstrating that it is strongly NP-complete by a reduction from SecAP. Let us define the problem.

Definition 18 (Tree Swap). *Take any tree $t = \alpha[t_1, \ldots, t_n] \in T_\Sigma$ and any $P = (p_1, \ldots, p_m) \in pos(t)$ such that $(p_1, \ldots, p_{m-1}, (p_m + 1)) \in pos(t)$. Then define the single-swap function*

$$\mathit{swap}_1(t, P) = \begin{cases} \alpha[t_1, \ldots, t_{p_1-1}, \mathit{swap}_1(t_{p_1}, (p_2, \ldots, p_m)), t_{p_1+1}, \ldots, t_n] & \text{if } m > 1, \\ \alpha[t_1, \ldots, t_{p_1-1}, t_{p_1+1}, t_{p_1}, t_{p_1+2}, \ldots, t_n] & \text{otherwise.} \end{cases}$$

The full swap function is for (appropriate) positions $P_1, \ldots, P_p$ defined as

$$\mathit{swap}(t, (P_1, \ldots, P_p)) = \mathit{swap}_1(\ldots \mathit{swap}_1(\mathit{swap}_1(t, P_1), P_2) \ldots, P_p).$$

The definition of swaps for trees is slightly unwieldy, but the swap function takes a tree and a sequence of tree positions (which are integer sequences). The positions identify, in order, the subtree which should next swap position with its sibling immediately to the right. Notice that P_i for $i > 1$ does not refer to a position in the tree t but to a position in an intermediary tree, it may be that $P_i \notin pos(t)$. An example is shown in Figure 6.

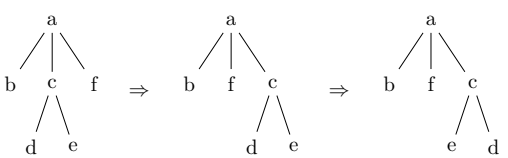


Figure 6: An example of applying the tree swaps $((2), (3, 1))$ to a small tree. That is, going from the first to second tree we swap the position 2, referring to the second child of the root, next the position $(3, 1)$ is swapped, referring to the first child of the rightmost child subtree of the root.

The definition of the tree swap distance problem now follows a familiar formula.

Definition 19 (The Tree Swap Distance Problem). *An instance of the tree swap distance problem is a tuple (t, t', b) where $t \in T_\Sigma$ is the start tree, $t' \in T_\Sigma$ is the target tree and $b \in \mathbb{N}$ is the budget. The instance is a "yes" instance if and only if there exists some $P_1 \in \mathbb{N}^*, \ldots, P_n \in \mathbb{N}^*$ such that $n \leq b$ and $t' = \mathit{swap}(t, (P_1, \ldots, P_n))$. We denote the set of all such "yes" instances* TSwD.

The next definition is used to make it easier to talk about minimal swap sequences.

Definition 20 (Minimal budget for TSwD). *For all $t, t' \in T_\Sigma$ let $mincost(t, t') = b$, where $b \in \mathbb{N}$ is the smallest number for which $(t, t', b) \in$ TSwD. If no such number exists let $b = \infty$.*

The reduction from SecAP to TSwD requires some building blocks. A visual example of the different types of notation defined below is shown later in Figure 8.

Definition 21 (Number Tree). *Assume that $0, 1 \in \Sigma$. For some symbol $\alpha \in \Sigma$ and $x, y \in \mathbb{N}$ such that $x \leq y$ we let $\alpha[x : y]$ denote the tree $\alpha[p_1, \ldots, p_{y+1}]$ where $p_i = 0$ for all $i \neq x + 1$ and $p_{x+1} = 1$.*

For example, $\alpha[2 : 3] = \alpha[0, 0, 1, 0]$. We call these trees "number trees". Notice that for all $x, x', y \in \mathbb{N}$ such that $x \leq y$ and $x' \leq y$ it holds that $mincost(\alpha[x : y], \alpha[x' : y]) = |x - x'|$. That is, the minimum number of swaps needed to turn $\alpha[x : y]$ into $\alpha[x' : y]$ is exactly $|x - x'|$. The tree $\alpha[x : y]$ serves the purpose to represent the number x, with the minimal swap distance to any other $\alpha[x' : y]$ being the absolute difference between x and x'.

Definition 22 (Number Trees with Neutral Elements). *Assume that for each $\alpha \in \Sigma$ there exists a distinct $\alpha' \in \Sigma$. Then for all $x, y \in \{0, 2, 4, 6, \ldots\}$ let $\alpha\langle x : y\rangle$ denote the following special tree.*

$$\alpha\langle x : y\rangle = \alpha\left[\alpha\left[\frac{x}{2} : \frac{y}{2}\right], \alpha'\left[\frac{y-x}{2} : \frac{y}{2}\right]\right].$$

Additionally let $\alpha\langle \perp : y\rangle$ denote the special tree $\alpha\left[\alpha\left[0 : \frac{y}{2}\right], \alpha'\left[0 : \frac{y}{2}\right]\right]$, called a "neutral" tree.

So, for example $\alpha\langle 2 : 6\rangle$ is the tree $\alpha[\alpha[0, 1, 0, 0], \alpha'[0, 0, 1, 0]]$. These trees have the property that for all $x, x', y \in \{0, 2, 4, 6, \ldots\}$ it holds that $mincost(\alpha\langle x : y\rangle, \alpha\langle x' : y\rangle) = |x - x'|$. This should not be a surprise, these trees behave like the earlier number trees, only the necessary swaps are split across two subtrees, and we lose the capability to represent odd numbers in the process. The gain lies in the neutral trees, it holds that $mincost(\alpha\langle \perp : y\rangle, \alpha\langle x : y\rangle) = \frac{y}{2}$ completely *independently of the value x*.

Definition 23 (Multi-number Trees). *For some $\alpha \in \Sigma$ and $k \in \mathbb{N}$ assume that we have the distinct symbols $\alpha_1, \ldots, \alpha_k \in \Sigma$. Then, for all $x_1, \ldots, x_k \in \mathbb{N} \cup \{\perp\}$, such that either $x_i \leq y$ or $x_i = \perp$ for all $i \in [n]$, let $\alpha\langle (x_1, \ldots x_k) : y\rangle$ denote the tree*

$$\alpha[\alpha_1\langle x_1 : y\rangle, \ldots, \alpha_k\langle x_k : y\rangle].$$

This means that

$$mincost(\alpha\langle (x_1, \ldots, x_n) : y\rangle, \alpha\langle (x'_1, \ldots, x'_n) : y\rangle) = \sum_{i=1}^{n} |x_i - x'_i|,$$

for all $x_1, x'_1, \ldots, x_n, x'_n, y \in \mathbb{N}$ such that $x_i \leq y$ and $x'_i \leq y$ for all $i \in [n]$.

Now all the building blocks necessary to reduce a swap even-cost assignment problem instance to a tree swap problem instance are ready.

Definition 24 (Reducing SecAP to TSwD). *Let (M, b) be an instance of the swap even-cost assignment problem as in Definition 12. We then construct the instance (t, t', b') of the tree swap distance problem as follows. Assume that M is an n by n matrix, let τ be the largest integer that occurs in M. Then let $b' = b + \frac{n(n-1)\tau}{2}$ and construct*

$$\begin{aligned} t = \alpha[\, &\beta\langle(M_{1,1}, \ldots, M_{1,n}) : \tau\rangle, \\ &\beta\langle(M_{2,1}, \ldots, M_{2,n}) : \tau\rangle, \\ &\vdots \\ &\beta\langle(M_{n,1}, \ldots, M_{n,n}) : \tau\rangle], \end{aligned}$$

and

$$\begin{aligned} t' = \alpha[&\beta\langle(0, \bot, \bot, \ldots, \bot) : \tau\rangle, \\ &\beta\langle(\bot, 0, \bot, \ldots, \bot) : \tau\rangle, \\ &\vdots \\ &\beta\langle(\bot, \bot, \ldots, \bot, 0) : \tau\rangle], \end{aligned}$$

that is, $t' = \alpha[t_1, \ldots, t_n]$ such that for all $i \in [n]$ we have $t_i = \beta\langle(x_1, \ldots, x_n) : \tau\rangle$ where $x_j = \bot$ for all $j \neq i$ and $x_i = 0$.

The dense notation may make this reduction hard to visualize, let us look at an example.

Example 25. Let (M, b) be an instance of the swap even-cost assignment problem, letting $b = 3$ and

$$M = \begin{bmatrix} 4 & 0 \\ 2 & 2 \end{bmatrix}.$$

Now we construct the tree swap distance problem instance (t, t', b') by applying the reduction from Definition 24. From M we see that $\tau = 4$, so the budget becomes $b' = 3 + \frac{2(2-1)4}{2} = 7$. The constructed trees are

$$\begin{aligned} t &= \alpha[\beta\langle(4, 0) : 4\rangle, \beta\langle(2, 2) : 4\rangle], \\ t' &= \alpha[\beta\langle(0, \bot) : 4\rangle, \beta\langle(\bot, 0) : 4\rangle]. \end{aligned}$$

To get past the notation the full tree t is shown in Figure 7, and the tree t' (as well as a breakdown of which subtrees correspond to which piece of notation) is shown in Figure 8.

Using these figures it is not hard to see how the solutions to (M, b) and (t, t', b') correspond to each other. (M, b) has a single solution, swapping the two rows (which gives a diagonal sum of 2, for a total cost of 3, which is exactly the budget), making no swap is not an option since the initial diagonal sum is 6, which is over the budget.

The decision to swap the rows in M or not corresponds to the decision whether or not to swap the $\beta\langle\ldots\rangle$-subtrees in t. The reader can easily verify by inspecting Figure 7 and 8 that it takes 10 swaps to move the 0/1 nodes around to match t' if we do not swap the $\beta\langle\ldots\rangle$-subtrees first, which is over the budget (in fact, it is over the budget by the same amount as the initial order of M is for that instance). If the two $\beta\langle\ldots\rangle$-subtrees are swapped however, we can reorder the 0/1 nodes in the resulting tree in only 6 swaps, for a total cost of 7, exactly the budget b'.

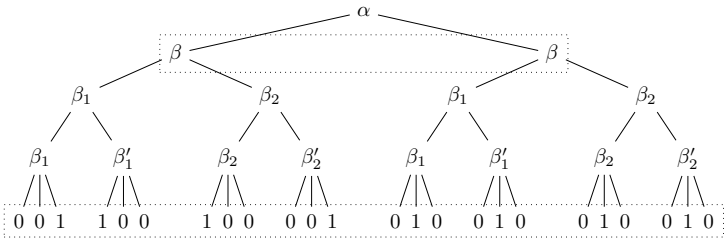


Figure 7: The tree t constructed in the reduction in Example 25. Notice that any solution only needs to perform swaps on the nodes in the dotted rectangles, all other nodes are already in their only possible internal order (compare to t' in Figure 8).

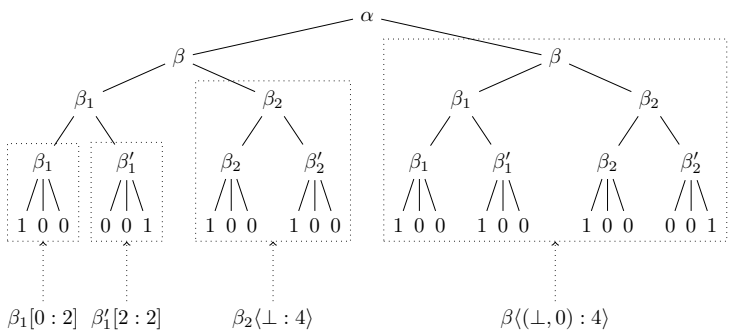


Figure 8: The tree t' constructed in the reduction in Example 25. The dotted arrows shows the notation we use to describe the indicated parts of the tree.

Hopefully the example has clarified the general idea of this reduction, but a proof sketch follows which further illustrates how it functions in the general case.

Lemma 26. *For every swap even-cost assignment problem instance (M, b) and tree swap distance problem instance (t, t', b') constructed from (M, b) by the reduction in Definition 24 it holds that $(t, t', b) \in$ TSwD if and only if $(M, b) \in$ SecAP.*

Proof (Sketch). We reuse the notation of the reduction. First notice that there are only two levels of swapping to consider in t. The immediate subtrees can be reordered since all are of the β multi-number kind, this is the interesting part. In addition the leaves will be swapped to move around the 0/1 sequences that are there to represent numbers, but this is abstracted by our number trees and can only be done in one trivial way once the top-level swaps are decided. The nodes in between are marked with distinct symbols.

Now let us look at the sub-subtrees in t'. There are n^2 of them, organized into n subtrees, each of which represents a row. For each $i \in [n]$ look at position i, i in t', this tree is of the form $\beta_i\langle 0 : \tau\rangle$, whereas for all $i, j \in [n]$ such that $i \neq j$ the subtree at position i, j is of the form $\beta_j\langle \bot : \tau\rangle$. These $n(n-1)$ trees will be matched up with some β_j sub-subtree in t at a constant cost of $\frac{\tau}{2}$ each, incurring a constant and unavoidable cost of $\frac{n(n-1)\tau}{2}$, leaving exactly b of the budget for the remainder.

This leaves the n "diagonal" subtrees of the form $\beta_i\langle 0 : \tau\rangle$ in t'. Assume that W in M moves row i into position j, incurring some swap cost and a diagonal cost of $M_{i,j}$. If we apply W directly to t this would move subtree $\beta\langle M_{i,1}, \ldots, M_{i,n}\rangle$ into position to match the tree in t' that contains the zero number tree $\beta_j\langle 0 : \tau\rangle$ in position j. This means that the cost incurred, beyond the already accounted for constant cost associated with the $n-1$ neutral trees will be $\mathit{mincost}(\beta_j\langle M_{i,j} : \tau\rangle, \beta_j\langle 0 : \tau\rangle)$, which is exactly $M_{i,j}$ by the construction of the number trees. So, to recap, applying W at the top level leaves us with the constant cost of $\frac{n(n-1)\tau}{2}$ plus $|W|$ plus $M_{i,j}$ for each row moved from position i to position j by W. Which is exactly the same cost that applying W in M incurs plus $\frac{n(n-1)\tau}{2}$, and since $b' = b + \frac{n(n-1)\tau}{2}$ this makes the problem instances equivalent. We did the argument starting from W, but we can trivially extract the swaps which deal with the immediate subtrees in t from a solution to (t, t', b'), making the other direction very straightforward. □

Corollary 27. *The tree swap distance problem is strongly NP-complete.*

As before the problem being in NP is trivial since the swap sequence never needs to be longer than n^2 so we may guess it. The reduction being polynomial is not hard to see, though the details become somewhat lengthy. There are on the order of $\mathcal{O}(\tau n^2)$ nodes in the trees, but SecAP is strongly NP-complete so this unary representation is not problematic.

7 Conclusion

Treating a problem where the only conclusion is negative, the problem being intractable, is never quite the ideal outcome. On the other hand it was already known that tree edit distance with subtree movement is problematic, and the efforts to integrate limited forms of swaps have been ongoing for some time. As such it is useful to establish that swaps are *inherently* problematic in trees. This hints that better results may be achieved if one considers simpler measures, such as linear distance, where all subtrees are reordered simultaneously and the cost of moving a subtree from position i to position j is exactly $|i-j|$ independent of whether the trees in between are moved. This would allow the Hungarian algorithm [6] to be leveraged in the tree case, giving a polynomial algorithm.

The problem itself may also be useful for complexity analysis of other swap problems, since it is at its core very simple both to explain and intuitively understand.

Hopefully this rather fundamental problem being proven NP-complete will also serve as a useful stepping stone for other complexity-theoretical work.

References

1. D. T. Barnard, G. Clarke, and N. Duncan: *Tree-to-tree correction for document trees*, Tech. Rep. 1995-372, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, 1995.
2. P. Bille: *A survey on tree edit distance and related problems.* Theor. Comput. Sci., 337(1-3) 2005, pp. 217–239.
3. F. J. Damerau: *A technique for computer detection and correction of spelling errors.* Commun. ACM, 7(3) 1964, pp. 171–176.

4. M. R. Garey and D. S. Johnson: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.

5. P. N. Klein: *Computing the edit-distance between unrooted ordered trees*, in In Proceedings of the 6th annual European Symposium on Algorithms (ESA, Springer-Verlag, 1998, pp. 91–102.

6. H. W. Kuhn: *The hungarian method for the assignment problem.* Naval Research Logistics Quarterly, 2(1-2) 1955, pp. 83–97.

7. V. I. Levenshtein: *Binary codes capable of correcting deletions, insertions, and reversals.* Soviet Physics Doklady, 10(8) 1966, pp. 707–710.

8. S. M. Selkow: *The tree-to-tree editing problem.* Inf. Process. Lett., 6(6) 1977, pp. 184–186.

9. K.-C. Tai: *The tree-to-tree correction problem.* J. ACM, 26 July 1979, pp. 422–433.

10. R. A. Wagner: *On the complexity of the extended string-to-string correction problem*, in STOC '75: Proceedings of seventh annual ACM symposium on Theory of computing, New York, NY, USA, 1975, ACM, pp. 218–223.

11. R. A. Wagner and R. Lowrance: *An extension of the string-to-string correction problem.* J. ACM, 22(2) 1975, pp. 177–183.

12. K. Zhang and D. Shasha: *Simple fast algorithms for the editing distance between trees and related problems.* SIAM J. Comput., 18(6) 1989, pp. 1245–1262.

13. K. Zhang, R. Statman, and D. Shasha: *On the editing distance between unordered labeled trees.* Information Processing Letters, 42(3) 1992, pp. 133 – 139.

Department of Computing Science
Umeå University, SE-901 87 Umeå, Sweden
www.cs.umu.se

ISBN 978-91-7601-047-1
ISSN 0348-0542
UMINF 14.13