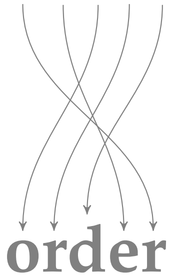
Complexities of Parsing in the Presence of Reordering



Martin Berglund

Department of Computing Science
Licentiate Thesis, Umeå 2012

Complexities of Parsing in the Presence of Reordering

Martin Berglund

LICENTIATE THESIS, APRIL 2012
DEPARTMENT OF COMPUTING SCIENCE
UMEÅ UNIVERSITY
SWEDEN

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

mbe@cs.umu.se

ISBN 978-91-7459-435-5
ISSN 0348-0542
UMINF 12.10

Front cover by Gustaf Bratt (Copyright © 2012, used with permission).
Printed by Print & Media, Umeå University, 2012.

Abstract

The work presented in this thesis discusses various formalisms for representing the addition of order-controlling and order-relaxing mechanisms to existing formal language models. An immediate example is shuffle expressions, which can represent not only all regular languages (a regular expression is a shuffle expression), but also features additional operations that generate arbitrary interleavings of its argument strings. This defines a language class which, on the one hand, does not contain all context-free languages, but, on the other hand contains an infinite number of languages that are not context-free. Shuffle expressions are, however, not themselves the main interest of this thesis. Instead we consider several formalisms that share many of their properties, where some are direct generalisations of shuffle expressions, while others feature very different methods of controlling order. Notably all formalisms that are studied here

- have a semi-linear Parikh image,
- are structured so that each derivation step generates at most a constant number of symbols (as opposed to the parallel derivations in for example Lindenmayer systems),
- feature interesting ordering characteristics, created either by derivation steps that may generate symbols in multiple places at once, or by multiple generating processes that produce output independently in an interleaved fashion, and
- are all limited enough to make the question of efficient parsing an interesting and reasonable goal.

This vague description already hints towards the formalisms considered; the different classes of mildly context-sensitive devices and concurrent finite-state automata.

This thesis will first explain and discuss these formalisms, and will then primarily focus on the associated membership problem (or parsing problem). Several parsing results are discussed here, and the papers in the appendix give a more complete picture of these problems and some related ones.

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 three articles, relating to the subjects discussed in the introduction, are included.

Paper I   This is an as of yet unpublished version combining and updating the content of the following two papers.

Martin Berglund, Henrik Björklund, and Johanna Högberg. Recognizing shuffled languages. *Technical Report UMINF 11.01, Inst. Computing Sci., Umeå University*, Available at `http://www8.cs.umu.se/research/uminf/index.cgi?year=2011&number=1`, 2011.

Martin Berglund, Henrik Björklund, and Johanna Björklund. Recognizing shuffled languages. *Proc. Language and Automata Theory and Applications.* (2011) 142–254.

Paper II   Martin Berglund. The membership problem for the shuffle of two deterministic linear context-free languages is NP-complete. *Technical Report UMINF 12.09, Inst. Computing Sci., Umeå University*, Available at `http://www8.cs.umu.se/research/uminf/index.cgi?year=2012&number=9`, 2012.

Paper III   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, Czech Technical University in Prague, Czech Republic, 2011.

Acknowledgments

All of this would never have been possible without my primary advisor Frank Drewes, whose support, advice and encouragement have been indispensable (he also very notably gave me my current position after I had already quit from it before starting once). I also need to, for too many reasons to enumerate, thank my co-advisor Henrik Björklund, my unofficial advisor Johanna Björklund, and my ex-advisor Michael Minock. Brink van der Merwe helpfully pointed out the semi-linear Parikh image being a link between CFSA and the mildly context-free languages, contributing greatly to the direction of this thesis. Stephen Hegner has given me good advice on general research strategy and direction, though I have not always made a good effort to follow it. I also appreciate the help of the rest of the Formal and Natural Languages group at the Umeå University Computer Science department, and many others at the department and university.

I must of course also thank my family, who have supported me from a distance throughout work on this thesis, while thankfully worrying very little about the trivialities of the actual work, focusing instead on making sure that I am well and happy. I of course owe my good friends a lot of thanks for many things, some of them even related to this thesis. A very partial list would necessarily have to include some Gustaf, John, Josefin, Mårten, Sandra, Sigge for starters, and then there are a couple of Magnus, at least one Tommy, a Johan, and many more. Thank you all!

Finally I would like to dedicate this thesis to the memory of the great philosopher Bertil Larsson.

Contents

Chapter 1
Introduction

This thesis studies the impact of reordering in formal languages in the context of parsing. Specifically, it has its basis in common formal language formalisms like context-free grammars, and adds additional order-controlling and/or order-relaxing mechanisms that allow reorderings to be performed in the derivation procedure. This is of great interest, as many both practical and theoretical processes have properties that are easy to describe in terms of (re)ordering. Consider as a starting point this introductory example.

Example 1.1 (Multi-Process Interleaving) An instance of the computer program P produces as output a sequence of symbols on the communication channel C. We have a context-free grammar G which can recognize whether this symbol sequence corresponds to a valid run of P (that is, the instance of P completing its run correctly).

Now we start n instances of P, all connected to the same communication channel C at the same time, which will arbitrarily interleave their output symbols as they run. Can we modify G to recognize whether the output on C corresponds to all n instances of P running correctly? $\diamond$

This type of problem is very difficult but is of great interest in program verification. Still, this is only one aspect of what will be discussed here. Before we get to the specifics of this however, let us recall some of the basic facts about formal languages.

1.1 Formal Languages

Formal languages is a very large area of study, with innumerable applications. The oldest and most central part of formal languages is concerned with *string languages*. A string is a finite sequence of symbols from an alphabet, a finite set of symbols, usually denoted Σ. We will use the Latin alphabet $\Sigma = \{a, b, c, \ldots\}$ here. A formal (string) language is then a (potentially infinite) set of strings. Trivial examples of formal languages include the empty set $\emptyset$ and the set of *all* possible strings, denoted Σ^*. The first key consideration about formal languages is how they can be formally described. The languages will often be infinite, Σ^* being the trivial example, but they need to be described in finite space to make it possible to work with them computationally.

Formally defined sets of strings are at the core of formal language theory, but this is in itself a rather wide concept. Many diverse computational problems can be phrased in terms of formal languages, and as such the focus lies in defining *classes* of formal languages which can be described by a specific type of formalism. It is easy to describe any finite language (finite set of strings) by simply exhaustively listing the elements, however, for infinite languages some finite description is necessary (for very large finite languages a succinct description is also of interest). In the case of natural languages, for example english, linguists construct grammars which describe how words can be combined into sentences. These grammars are fairly small and attempt to describe how to generate all the sentences in an ostensibly infinite language. Context-free grammars are an example of a formal-language formalism that functions in a similar way. A context-free grammar G specifies rules for combining symbols, generating a potentially infinite language, denoted $\mathscr{L}(G)$. Not all languages can be generated by a context-free grammar, for example $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ (that is, the language $\{\varepsilon, abc, aabbcc, aaabbbccc, \ldots\}$) is not generated by any context-free grammar. It is also common to consider an automaton A, and define the language $\mathscr{L}(A)$ as exactly the strings on which A accepts (halts successfully). That is, $s \in \mathscr{L}(A)$ if and only if A accepts when given s as input. The distinction between automata and grammars is primarily a question of which phrasing is more convenient for the formalism and problem at hand.

1.2 Computational Problems in Formal Languages

With a formalism for generating languages in hand there are various questions that can be asked. For example, the emptiness problem; given an automaton/grammar G, is the language $\mathscr{L}(G)$ empty? This is easy to compute for context-free grammars (the opposite, whether a given grammar accepts *all* strings is undecidable, however). Problems that deal with the languages as a whole are also interesting, for example, does there for any context-free languages L_1 and L_2 (a context-free language is a language that can be generated by a context-free grammar) necessarily exist a context-free grammar that generates $L_1 \cup L_2$? This is in fact the case, but does not hold for $L_1 \cap L_2$. The problem we are primarily concerned with here however is *membership*, determining whether a string belongs to a language or not. This problem comes in three flavors, first the most direct one.

Definition 1.2 (The Uniform Membership Problem) Let $\mathscr{G}$ be a class of formal devices (e.g., grammars or automata) such that each $G \in \mathscr{G}$ defines a formal language $\mathscr{L}(G)$. *The uniform membership problem for* $\mathscr{G}$ is "Given a string w and some $G \in \mathscr{G}$ as input, is w in $\mathscr{L}(G)$?" ◇

It is, however, very common that we are unconcerned about the exact formalism $\mathscr{G}$ by which the language is represented. That is, we are in a situation where the language is known in advance and can be coded into the most efficient representation imaginable, making the size of the string the real concern. This gives rise to the non-uniform case.

Definition 1.3 (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?" ◇

Thirdly there is the problem of *parsing*. This is the problem of, once it has been determined that a string w belongs to a language $\mathscr{L}(G)$, *describing* in terms of G how w was generated (or accepted). In most cases the solution to this problem follows naturally from any algorithm that can solve the problem in Definition 1.2 (or Definition 1.3 with a grammar/automaton fixed). Thanks to this fact the membership problems will be the ones considered throughout this thesis, despite the parsing problem ultimately being of real interest.

1.3 Formalisms Controlling Order

The nature of ordering is central to the difference between the less powerful language classes at the bottom of the Chomsky hierarchy (i.e., smaller classes strictly contained in for example the context-sensitive languages). In a seminal paper from 1966 Rohit J. Parikh [Par66] demonstrated that if the order of the symbols in strings is "ignored" the context-free languages are no more powerful than the regular languages: both are exactly the so-called semi-linear sets. In addition a wide variety of language models have been defined in the "mildly context-sensitive" class, which requires languages to have exactly this unordered semi-linearity property (depending slightly on the source), while providing strictly more power than the context-free languages.

Looking at Parikh's theorem from this perspective intuitively demonstrates just how much the addition of reordering operations can change the structure of a language class and make it more or less powerful. Where taking away the ordering entirely from context-free languages makes them in a sense loose some of their characteristics it is also possible to for example add some reordering operations to regular expressions to make them yield languages that are not even context-free. Finding mechanisms for controlling ordering which do not cause the resulting language class to be intractably hard to parse is not an easy task.

As a straightforward example, the language $a^n b^n c^n$ is famously not context-free, but reordering the symbols conveniently yields the regular language $(abc)^n$, and the language of *all* reorderings of $a^n b^n c^n$ is the set of all strings over $\{a,b,c\}$ with equal numbers of as, bs, and cs. This language is matched by a shuffle expression, an extension of regular expressions that is treated in Paper I.

To make things more concrete there are two ways to affect and control order that will be considered here.

Shuffle languages. For the most basic setting, consider the situation illustrated in Figure 1.4. Here a number of automata instances, let us call them $A_1, \ldots, A_k$, (these might be finite automata, or push-down automata, etc.) work on the same string at the same time, each symbol being non-deterministically read by one of the automata, while the others are unaware that a symbol has been read. If all k automata have reached an accepting state at the end of the string the whole input string is accepted. Another way to view this is to ask if the string can be divided into k subsequences $s_1, \ldots, s_k$ such that s_i is accepted by A_i for each i. Intuitively this simply means that s can be produced by interleaving the strings $s_1, \ldots, s_k$, but a full definition is given in

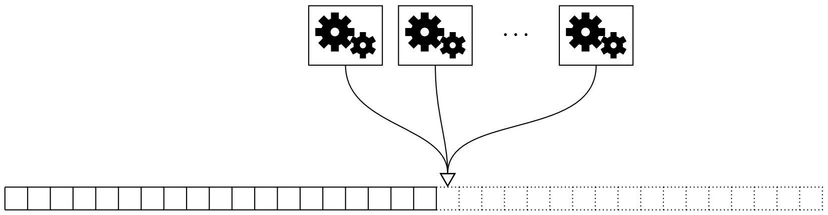


Figure 1.4: An overview of the automata view of shuffling. Several automata share the same read head. Each symbol in the string is non-deterministically read by *one* of the automata, the common read head is then stepped forward (i.e., working left to right), leaving all the other automata effectively unaware of the step.

Definition 2.1 in Chapter 2.

It is a small leap to give a more complete picture of the automata defined in Paper I. In Figure 1.5 the automata are finite-state, but instead feature the ability to spawn child

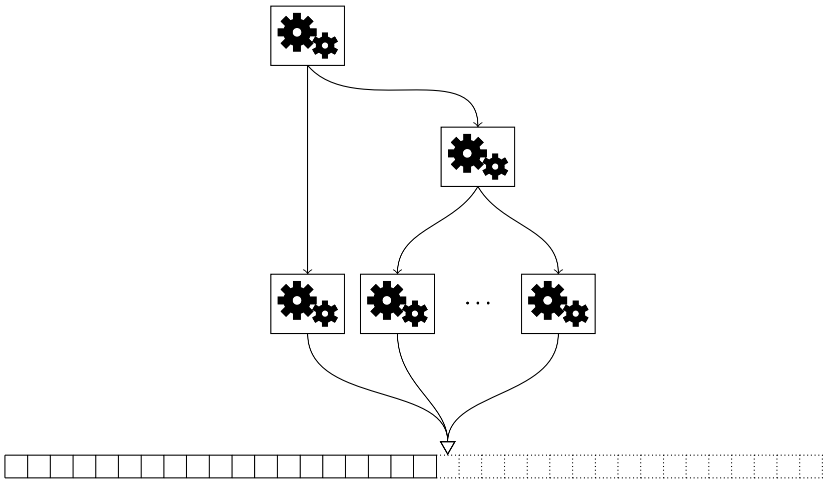


Figure 1.5: Extending the picture in Figure 1.4, Paper I effectively considers a process hierarchy in which the individual processes are (an extended type of) finite automata. The leaf automata all read from the string in the same mode as in Figure 1.4, and automata are able to launch child automata, at which point they are suspended until all children have accepted.

automata. Only "leaf" automata actually run, reading from the string (again non-deterministically ordered) and possibly spawning child automata of their own. When an automaton reaches an accepting state it disappears, possibly making its parent a leaf once more, allowing it to continue running. This allows various types of formalisms to be implemented, many of which can be isolated by syntactical restrictions on one of these automata.

Synchronized substrings. The second type of control over ordering to be studied is almost the opposite, see Figure 1.6. The formalisms intended here are slightly harder

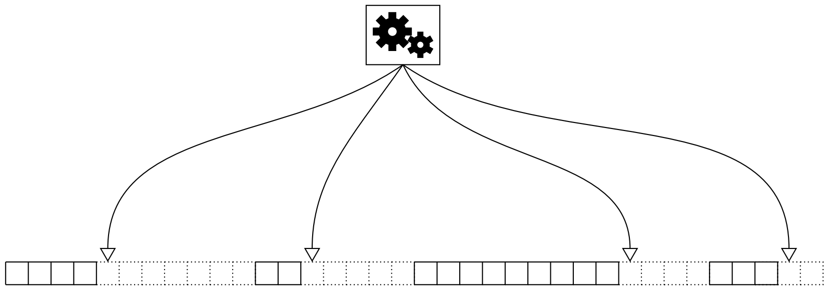


Figure 1.6: A high-level overview of the second type of reordering that will be considered, where one automaton has multiple (a constant number) of reads heads simultaneously operating on separate parts of the string.

to describe in an informal way, but a good starting-point is to consider an automaton with multiple read heads (but still only a constant number) operating on different parts of the string. Intuitively, each head is responsible for processing an isolated substring. Notably a push-down automaton of this type, with k read heads, can accept the language $\{w^k \mid w \in L, k \text{ an integer constant}\}$ for any context-free language L. This description is of course vague (leaving out how the read heads can be created and placed), and the formalisms later covered take a different more precise form. Rather than being featured in-depth in the papers, this type of reordering is the subject of ongoing research and is therefore brought up here to give a more complete overview. Moreover, Paper III considers a somewhat similar case.

Problems considered. In both types of formalisms what is of primary interest is parsing, here simplified into the membership problem (recall Definitions 1.2 and 1.3). The core results from the papers in the appendix that are touched upon here are in the area of shuffle, but the synchronized substrings type of reordering can be considered as a contrast. The general formulations of these problems are unfortunately NP-complete, and are as such not efficiently decidable unless $P = NP$. Luckily restrictions of various kinds yield better results, where polynomial parsing algorithms are possible. Importantly, the non-uniform membership problem is solvable in polynomial time for the synchronized substring formalisms considered, and for several restrictions of the general shuffle formalism (e.g. shuffle expressions).

Another less formal but more overarching question to consider is how the differences between the shuffle formalisms and synchronized substring formalisms show up in the languages generated.

CHAPTER 2

Shuffle Languages

In this chapter a specific type of device for the description of shuffle languages is considered, the Concurrent Finite-State Automaton, or CFSA for short, introduced in Paper I. It will be informally defined and several examples will be considered to prepare for a discussion of the contents of Paper I, as well as to compare CFSA to the synchronized substrings formalisms discussed in the following chapter. Shuffling has a rich history of publications beyond the papers included here however, and Section 2.4 gives a summary of some of the important literature.

2.1 Shuffle Formalisms

These formalisms are named after the *shuffle operation*, an important building-block for describing the languages that can be generated. It is based on the idea of interleaving strings, as was already discussed in Chapter 1. Let us now properly define what it means to divide a string into subsequences.

Definition 2.1 (Dividing a String Into Subsequences) The integer sequence $1,\ldots,n$ can be divided into the subsequences $i_1,\ldots,i_m$ and $j_1,\ldots,j_{n-m}$ if and only if every integer in $\{1,\ldots,n\}$ occurs exactly once in $i_1,\ldots,i_m,j_1,\ldots,j_{n-m}$ and both $i_1 < \cdots < i_m$ and $j_1 < \cdots < j_{n-m}$.

The string $\alpha_1 \cdots \alpha_n$, where each α_i is a symbol, can be divided into the subsequences w and v if and only if $1,\ldots,n$ can be divided into some $i_1,\ldots,i_m$ and $j_1,\ldots,j_{n-m}$ such that $w = \alpha_{i_1} \cdots \alpha_{i_m}$ and $v = \alpha_{j_1} \cdots \alpha_{j_{n-m}}$.

Furthermore, a sequence s can be divided into the subsequences $s_1,\ldots,s_k$ if and only if either $k = 1$ and $s = s_1$; or s can be divided into two subsequences s_1 and s' such that s' can be divided into the subsequences $s_2,\ldots,s_k$. ◇

From here the leap to the shuffle operation is short.

Definition 2.2 (The Shuffle Operation) For two strings w and v the shuffle operation, denoted $\odot$, is defined such that $w \odot v$ is the set of all strings s such that s can be divided into the subsequences w and v. ◇

A short example should help clarify these concepts.

Example 2.3 (Shuffling) To start with, note that the string "*abcbbac*" can for example be divided into the subsequences "*abbba*" and "*cc*", by picking the indices 1, 2, 4, 5, and 6 for the first string, leaving 3 and 7 for the second. It cannot be subdivided into "*aab*" and "*bcbc*", since the first string suggests that there should exist two a symbols before a b symbol in the original string, which is not the case.

In the other direction, the shuffle operation application $abc \odot ad$ yields the set $\{abcad, abacd, aabcd, abadc, aabdc, aadbc, adabc\}$. ◇

This operation is a good starting point for studying many kinds of shuffle, considering for example languages of the form $\{w \odot v \mid w \in L_1, v \in L_2\}$ for some languages L_1 and L_2. Another interesting direction are the so-called shuffle expressions, which are essentially regular expression with the addition of the shuffle operation (and what is known as the shuffle closure, which, applied to a string, generates the language of an arbitrary number of copies of that string shuffled together). These shuffle expressions generate the language class known as the shuffle languages. See the summary of the literature in Section 2.4 for more information on this subject. In this chapter, however, we consider a formalism that can represent an even larger class of languages.

Concurrent Finite-State Automata (CFSA) cover all types of shuffle that are of interest here. This formalism is what is loosely illustrated in Figure 1.5, where finite-state automata gain the ability to

1. spawn child automata, either a fixed number in individually chosen states, or an arbitrary (non-deterministically chosen) number all in the same state, and
2. all automata without children simultaneously non-deterministically read from the string, and "disappear" once they reach an accepting state.

These automata allow all shuffle languages to be recognized, as well as all context-free languages (using the spawning of child automata as a stack), the shuffle of context-free languages, and context-free languages with shuffle. It is important to make a distinction between the latter two, simply shuffling context-free languages together (allowing context-free languages as operands in a shuffle expression) is not equivalent to the full CFSA behavior.

Example 2.4 (A CFSA language) As a trivial example consider the language

$$\{a^n \cdot w \cdot b^n \mid n \in \mathbb{N}, w \in \{c,d,e\}^* \text{ s.t. } w \text{ contains equally many } c\text{s, } d\text{s and } e\text{s}\}$$

which nests a shuffle language inside balanced parentheses (in the sense that "*a*" is an opening parenthesis and "*b*" a closing one), yielding a language that is not recognized by any shuffle expression, context-free grammar or by any shuffle of context-free grammars. It is, however, recognizable by a CFSA in a straightforward way. ◇

To make this more specific we next make a more strict definition of what a CFSA can do. Going forward assume, unless otherwise noted, that $a, b, c, \ldots \in \Sigma$ are the symbols in the alphabet the strings are defined over, and that $q_0, q_1, \ldots$ are states in automata, with q_0 the initial state.

Definition 2.5 (CFSA) A CFSA is a non-deterministic finite-state automaton which in addition to rules of the form $q_1 \xrightarrow{a} q_2$ (going from state q_1 to state q_2 as usual) also has rules of the form $q_1 \xrightarrow{a} q_2[q_3q_4]$. That is, rather than having a "current state" the automaton has a state string consisting of states and (balanced) brackets, for example $q_1[q_2[q_5q_5]q_3]$. Acceptance means turning to the empty string of states, for example $q_2 \xrightarrow{a} \varepsilon$. Once a bracket pair is empty, that is, "[]" occurs as a substring in the state string, it is removed. No transition may be performed on a state which is immediately followed by a left bracket. That is, in the example $q_1[q_2[q_5q_5]q_3]$ there are two occurrences of q_5 and one occurrence of q_3 that may make transitions, whereas q_1 and q_2 are blocked until the brackets are removed. Only one transition at a time is performed, choosing non-deterministically where it happens.

There is one additional kind of rule, of the form $q_1 \xrightarrow{a} q_2[q_6^*]$, which consumes an "$a$" symbol from the input and replaces q_1 by a string of the form $q_2[q_6q_6q_6\ldots]$ with an arbitrary number of instances of q_6, non-deterministically chosen. Notably, q_1 may be replaced by $q_2[]$, which allows the brackets to be immediately dropped. ◇

Some examples should make this definition clear.

Example 2.6 (a^nb^n) Consider a CFSA with the following rules:

$$q_0 \xrightarrow{a} q_1[q_0] \qquad q_0 \xrightarrow{a} q_1 \qquad q_1 \xrightarrow{b} \varepsilon.$$

Then a run of the automaton takes the form shown in Table 2.7. This CFSA accepts the language $\{a^nb^n \mid n \in 1,2,\ldots\}$. Notice that in each step only one state is *not* followed by a left bracket, which forces that state to be the next one a rule is applied on. The "b"-reading transitions replace q_1 by ε, dropping the brackets, allowing the next q_1 to be handled. Once the state has become ε the input string is accepted.

Table 2.7: Run of the automaton in Example 2.6.

State	String read	Next rule
q_0	${}_{\vartriangle}aaaabbbb$	$q_0 \xrightarrow{a} q_1[q_0]$
$q_1[q_0]$	$a_{\vartriangle}aaabbbb$	$q_0 \xrightarrow{a} q_1[q_0]$
$q_1[q_1[q_0]]$	$aa_{\vartriangle}aabbbb$	$q_0 \xrightarrow{a} q_1[q_0]$
$q_1[q_1[q_1[q_0]]]$	$aaa_{\vartriangle}abbbb$	$q_0 \xrightarrow{a} q_1$
$q_1[q_1[q_1[q_1]]]$	$aaaa_{\vartriangle}bbbb$	$q_1 \xrightarrow{b} \varepsilon$
$q_1[q_1[q_1]]$	$aaaab_{\vartriangle}bbb$	$q_1 \xrightarrow{b} \varepsilon$
$q_1[q_1]$	$aaaabb_{\vartriangle}bb$	$q_1 \xrightarrow{b} \varepsilon$
q_1	$aaaabbb_{\vartriangle}b$	$q_1 \xrightarrow{b} \varepsilon$
ε	$aaaabbbb$	*accept*

This illustrates how CFSA can capture context-free languages. It is not hard to add additional rules to this example to make it accept all strings of balanced parenthesis, like for example "((()()(())))". ◇

Example 2.8 (All Reorderings of $a^nb^nc^n$) Consider the CFSA with the rules

$$q_0 \xrightarrow{\varepsilon} q_1[q_2^*] \qquad q_1 \xrightarrow{\varepsilon} \varepsilon \qquad q_2 \xrightarrow{\varepsilon} q_1[q_3q_4q_5] \qquad q_3 \xrightarrow{a} \varepsilon \qquad q_4 \xrightarrow{b} \varepsilon \qquad q_5 \xrightarrow{c} \varepsilon.$$

A run of this automaton can take the form showed in Table 2.9. This automaton accepts all reorderings of the strings $a^nb^nc^n$ (for $n \in \{0,1,2,\ldots\}$). Note that occurrences of the state q_1 only serve as "parent" place-holders for $q_3q_4q_5$ but do not actually read anything when their children have disappeared.

Table 2.9: An example run of the automaton in Example 2.8.

State	String read	Next rule
q_0	${}_{\vartriangle}bbccbacaa$	$q_0 \xrightarrow{\varepsilon} q_1[q_2^*]$
$q_1[q_2q_2q_2]$	${}_{\vartriangle}bbccbacaa$	$q_2 \xrightarrow{\varepsilon} q_1[q_3q_4q_5]$
$q_1[q_2q_2q_1[q_3q_4q_5]]$	${}_{\vartriangle}bbccbacaa$	$q_2 \xrightarrow{\varepsilon} q_1[q_3q_4q_5]$
$q_1[q_2q_1[q_3q_4q_5]q_1[q_3q_4q_5]]$	${}_{\vartriangle}bbccbacaa$	$q_4 \xrightarrow{b} \varepsilon$
$q_1[q_2q_1[q_3q_4q_5]q_1[q_3q_5]]$	$b_{\vartriangle}bccbacaa$	$q_4 \xrightarrow{b} \varepsilon$
$q_1[q_2q_1[q_3q_5]q_1[q_3q_5]]$	$bb_{\vartriangle}ccbacaa$	$q_2 \xrightarrow{\varepsilon} q_1[q_3q_4q_5]$
$q_1[q_1[q_3q_4q_5]q_1[q_3q_5]q_1[q_3q_5]]$	$bb_{\vartriangle}ccbacaa$	$q_5 \xrightarrow{c} \varepsilon$
$q_1[q_1[q_3q_4q_5]q_1[q_3q_5]q_1[q_3]]$	$bbc_{\vartriangle}cbacaa$	$q_5 \xrightarrow{c} \varepsilon$
$q_1[q_1[q_3q_4]q_1[q_3q_5]q_1[q_3]]$	$bbcc_{\vartriangle}bacaa$	$q_4 \xrightarrow{b} \varepsilon$
$q_1[q_1[q_3]q_1[q_3q_5]q_1[q_3]]$	$bbccb_{\vartriangle}acaa$	$q_3 \xrightarrow{a} \varepsilon$
$q_1[q_1q_1[q_3q_5]q_1[q_3]]$	$bbccba_{\vartriangle}caa$	$q_1 \xrightarrow{\varepsilon} \varepsilon$
$q_1[q_1[q_3q_5]q_1[q_3]]$	$bbccba_{\vartriangle}caa$	$q_5 \xrightarrow{c} \varepsilon$
$q_1[q_1[q_3]q_1[q_3]]$	$bbccbac_{\vartriangle}aa$	$q_3 \xrightarrow{a} \varepsilon$
$q_1[q_1q_1[q_3]]$	$bbccbaca_{\vartriangle}a$	$q_3 \xrightarrow{a} \varepsilon$
$q_1[q_1q_1]$	$bbccbacaa$	$q_1 \xrightarrow{\varepsilon} \varepsilon$
$q_1[q_1]$	$bbccbacaa$	$q_1 \xrightarrow{\varepsilon} \varepsilon$
q_1	$bbccbacaa$	$q_1 \xrightarrow{\varepsilon} \varepsilon$
ε	$bbccbacaa$	*accept*

Notice that whereas the automaton in Example 2.6 nests the bracket deeply, but each bracket contains at most one state, this automaton has finite nesting but unbounded branching by using the $q_0 \xrightarrow{\varepsilon} q_1[q_2^*]$ rule. ◇

The above examples illustrate two key facts about concurrent finite-state automata. Example 2.6 shows how they capture the context-free languages by having the state string simulate a stack, whereas Example 2.8 demonstrates how spawning multiple "parallel" states allows them to recognize shuffled together languages, while allowing the shuffling to be limited to syntactically delimited parts of the string (that is, a parent state reading a symbol from the string means that all potential shuffling in spawned child states have already been fully resolved). A combination of the two is also possible, allowing the language in Example 2.4, but the automaton becomes a bit larger and harder to understand.

2.2 CFSA in Relation to Context-free Languages

Recall the shuffle operation $\odot$ from Definition 2.2. Now let us combine it with normal string concatenation to create some simple shuffle expressions. These are still not full shuffle expressions as considered in the literature (see Section 2.4), but are interesting when considered in combination with a context-free grammar.

Definition 2.10 (Shuffle Operations in Expressions) Each $\alpha \in \Sigma$ is an expression that represents the language $\mathscr{L}(\alpha) = \{\alpha\}$. Let S and T be arbitrary expressions, then $(S \odot T)$ is an expression representing the language $\mathscr{L}((S \odot T)) = \{w \mid w \in s \odot t, s \in \mathscr{L}(S), t \in \mathscr{L}(T)\}$, and (ST) is an expression representing the language $\mathscr{L}((ST)) = \{st \mid s \in \mathscr{L}(S), t \in \mathscr{L}(T)\}$.

The parenthesis are used to control the order of evaluation, and may be added freely; (S) is an expression with $\mathscr{L}((S)) = \mathscr{L}(S)$. Parenthesis may be removed if doing so does not change the language, with the addition that concatenation is given priority in otherwise ambiguous expressions, so $(ab \odot c)d(e \odot f)$ generates the language $\{abcdef, acbdef, cabdef, abcdfe, acbdfe, cabdfe\}$, whereas the language generated by $ab \odot cde \odot f$ contains for example $fcabde$. $\diamond$

With this in place it is interesting to note that for each CFSA there exists a "characteristic" context-free grammar which rather than strings generates shuffle expressions that in turn generate the strings in the language of the CFSA.

Definition 2.11 (Characteristic Grammars for CFSA) For any CFSA A accepting strings in Σ^* we can construct the *characteristic* context-free grammar G_A, which contains strings over the alphabet $\{\odot,), (\} \cup \Sigma$, in the following way. For each state q_i the grammar has a non-terminal A_i. A_0 is the initial non-terminal. The rules in the CFSA A are translated into the rules of the grammar G_A in the following way

CFSA rule	Context-free rule
$q_i \xrightarrow{\alpha} \varepsilon$	$A_i \to \alpha$
$q_i \xrightarrow{\alpha} q_j$	$A_i \to \alpha A_j$
$q_i \xrightarrow{\alpha} q_{j_1}[q_{j_2} \cdots q_{j_n}]$	$A_i \to \alpha(A_{j_2} \odot \cdots \odot A_{j_n})A_{j_1}$
$q_i \xrightarrow{\alpha} q_j[q_r^*]$	$A_i \to \alpha(A'_r)A_j \mid \alpha A_j$

where A'_r is an extra non-terminal for each r of the CFSA with the rules

$$A'_r \to A_r, \qquad A'_r \to A_r \odot A'_r. \qquad \diamond$$

Now for each CFSA A the corresponding characteristic context-free language $\mathscr{L}(G_A)$ will be such that each string accepted by A is also a member of the language generated by one or more of the shuffle expressions in $\mathscr{L}(G_A)$. In fact, s is accepted by A if and only if there exists at least one string $E \in \mathscr{L}(G_A)$ such that, by evaluating the shuffle operations in E, the set of strings generated by E contains s. In other words, $\mathscr{L}(A) = \bigcup\{\mathscr{L}(E) \mid E \in \mathscr{L}(G_A)\}$. While it may not be immediately obvious that this is the case, a full proof is beyond the scope of this intuitive explanation of the way CFSA actually work.

The key thing to notice here is that a CFSA behaves in a context-free way with the added ability to *disregard* order. While there is not really any way to *change* ordering in a controlled manner, there always exists a characteristic context-free language at the core, which may generate shuffle operations to "loosen" the language. Notably, while a CFSA has already been seen that can generate all reorderings of $a^n b^n c^n$, which means an interesting superset of $a^n b^n c^n$ can be generated, the language $a^n b^n c^n$ *itself* cannot be generated by a CFSA. It is interesting to view this from the characteristic context-free grammar perspective; a CFSA can generate all reorderings of $a^n b^n c^n$ since there is a context-free language $(abc)^n$ from which to construct it, by simply disregarding all ordering (the characteristic context-free grammar generates a shuffle operation between every symbol generated).

In the next chapter this leads us naturally towards other types of formalisms, which take a very different approach to ordering.

2.3 The Membership Problem

The membership problem for various CFSA variations is covered at length in both Paper I and Paper II, the latter demonstrating that even the non-uniform membership problem is NP-complete for a very restricted CFSA. Paper I handles the opposite direction and demonstrates that the uniform unrestricted problem remains *in* NP, which is surprising seeing how a very restricted CFSA is used to demonstrate NP-hardness. In the next section we take a look at a formalism that is also capable of generating some interesting languages, while giving rise to a more efficiently decidable membership problem.

2.4 Overview of the Literature

Shuffle languages and related questions have been studied for a long time, arguably starting with a definition by S. Ginsburg and E. Spanier in 1965 [GS65]. A main thrust considered here are shuffle expressions, which generate the "shuffle languages", introduced by Gischer [Gis81]. This in turn was based on an 1978 article by Shaw [Sha78] on flow expressions, which were used to model concurrency. Shuffle expressions are in effect regular languages extended with the shuffle operation $\odot$, which was discussed in Section 2.2, as well as the shuffle closure, which iterates the shuffle operation (in analogy to the Kleene closure as iterated concatenation). The membership problem for shuffle expressions is NP-complete in general [Bar85, MS94], but can be decided in polynomial time in the non-uniform case [JS01].

Beyond shuffle expressions numerous other interesting membership problems are considered in the literature. An excellent example is Warmuth and Hausslers 1984 Paper [WH84] that among other things demonstrate 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 $w \in v \odot v \odot \cdots \odot v$. In a similar vein Ogden, Riddle and Rounds demonstrated that the non-uniform membership problem for the shuffle of two deterministic context-free languages is NP-complete [ORR78].

Other interesting directions include shuffle on trajectories [MRS98] and axiomatization of shuffle [EB98]. For a longer list of references, see the introduction of Paper I.

CHAPTER 3
Synchronized Substrings

As a contrast we now consider a very different type of formalism, where the finite control effectively controls multiple positions in a string. Out of the many formalisms that fit this vague description, two will be mentioned here as being of specific interest for ongoing research efforts. As such this chapter starts out with an overview of the literature relevant for the discussion, followed by informal examples, whose purpose it is to explain the nature of the relevant language formalisms.

3.1 Overview of the Literature

The formalisms discussed in the next section belong to a large category defined by Aravind Joshi in [Jos85] called "mildly context-sensitive". Joshi defines a language class $\mathscr{L}$ to be mildly context-sensitive if and only if

1. CF $\subseteq \mathscr{L}$, that is, $\mathscr{L}$ contains all context-free languages (this is left implicit in [Jos85]),
2. at least one language in $\mathscr{L}$ features *limited cross-serial dependencies*,
3. the membership problem is decidable in non-uniform (implicit in [Jos85]) polynomial time for all $L \in \mathscr{L}$,
4. for all $L \in \mathscr{L}$ the set $\{|w| \mid w \in L\}$ is semi-linear. That is, ordering the strings in a language in $\mathscr{L}$ by their length will yield a gradual increase, each string being at most a constant number of symbols longer than the last, with the constant determined by the language.

Requirement 2 specifically refers to a type of substring synchronization, illustrated by Joshi using tree-adjoining grammars, which makes it hard to illustrate here. Suffice it to say that $a^n b^n c^n$ features cross-serial dependencies, and is as such a sufficient addition.

This rather loose set of requirements has given rise to at least two classes of languages, the first being the motivating class, defined equivalently [JSW90] by tree-adjoining grammars [JLT75], linear indexed grammars [Gaz88], combinatorial categorial grammars [Ste87] and head grammars [Pol84]. The second, strictly more powerful class, is the one that is of interest for this section. It can be equivalently (as far as the language class generated is concerned) defined by linear context-free rewriting systems [Wei92], 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]. Since fully defining these formalisms, and defining languages in terms of them, is more complex than what is called for here a more intuitive but imprecise stand-in is used in the next section, which is based on the string-generating hyperedge replacement grammars.

3.2 A Simple Synchronized Substrings Formalism

Rather than attempting to discuss synchronization formally we explain it informally by means of an illustrative example. A natural way to describe intermediary configurations of generating strings using the formalisms listed in Section 3.1 is by a hypergraph, where each instance of a non-terminal is an edge identifying some number of nodes, and the final string is formed as a directed chain. Therefore, the formalism sketched in the examples is given in a visual form resembling string-generating hyperedge replacement grammars. However, modifying the examples into, for instance, simple range concatenation grammars or multiple context-free grammars is not very difficult (the only issue being that the examples are harder to read in symbolic form).

It is important to keep in mind that while the hyperedge replacement grammars here generate only strings (or directed connected chains in graph terms). In general hyperedge replacement grammars can generate a large class of graphs [Hab92, DHK97]. As an illustrative example with some relevance to the topics at hand, a hyperedge replacement grammar can easily generate what amounts to a multi-set of strings, that is, an unconnected graph where each connected subgraph is a directed chain. All the grammar needs to do is break the chain at some points. This language of multi-sets of strings needs to be avoided, since it has an NP-hard non-uniform membership problem, which can be established by a straightforward reduction. There exists a context-free language $L \subseteq \{a,b,\wr\}^*$ such that the non-uniform membership problem for the language $L_\wr = \big\{\{w_1,\ldots,w_n\} \mid w_1 \wr w_2 \wr \cdots \wr w_n \in L, n \in \mathbb{N}, w_1,\ldots,w_n \in \{a,b\}^*\big\}$ (such that $L_\wr$ is a set of multi-sets over $\{a,b\}^*$) is NP-complete [LW87].

Example 3.1 (A Hyperedge Replacement String Grammar) A hyperedge replacement string grammar consists of a finite set of non-terminals $A_0, A_1, A_2, \ldots$ each of which has an arity denoted $arity(A_i)$, and a finite set of rules. These rules identify a non-terminal, A_i, and $arity(A_i)$ positions (in a sentential form) on the left-hand side. The right-hand side then dictates what will be inserted into these $arity(A_i)$ positions. If all non-terminals have $arity(A_j) = 1$ then this is equivalent to a context-free grammar, because then a non-terminal identifies a single position in the sentential string, the position it is *in*. As a first example consider the rule in Figure 3.2. This rule replaces the non-terminal A_2, which has $arity(A_2) = 2$ here. The two arrows illustrate the two positions A_2 "controls". The rule then inserts the string $a \bullet b \bullet$ in the first position A_2 controls, where the first bullet corresponds to the first position known, or controlled, by the *new* instance of A_2 generated on the right-hand side, and the second is the posi-

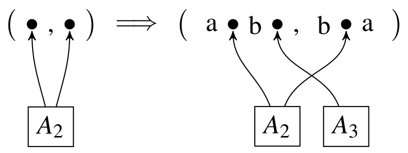


Figure 3.2: A simple rule for a string-generating hyperedge replacement grammar. It replaces the non-terminal A_2 with arity two by a new instance of A_2 and an instance of the non-terminal A_3, while generating some terminal symbols.

tion known by a new non-terminal A_3 (with $arity(A_3) = 1$). In the second position the left-hand side A_2 controls it inserts the string $b \bullet a$ where the bullet denotes the second position controlled by the new A_2 instance on the right-hand side. Notice that the number of positions on the left-hand side has to correspond to the number of strings in the tuple on the right-hand side. The initial non-terminal is A_0, which always has $arity(A_0) = 1$ (intuitively this must be the case since the derivation is supposed to generate one string). This makes the initial configuration (initial sentential string really) appear as in Figure 3.3.

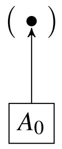


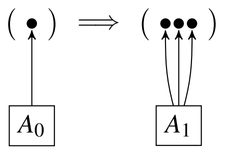
Figure 3.3: The initial configuration for string-generating hyperedge replacement grammars.

To clarify this further let us look at a more complete example. Consider the rules in Figure 3.4. These three rules generate the language $a^n b^n c^n d^n e^n f^n$. Leaving the non-terminals attached to the positions implicit, a derivation of a string in this grammar takes the structure $\bullet \to \bullet\bullet\bullet \to a\bullet bc\bullet de\bullet f \to aa\bullet bbcc\bullet ddee\bullet ff \to aaa\bullet bbbccc\bullet dddeee\bullet fff \to aaabbbcccdddeeefff$ where the first rule applied is (a) (replacing the initial A_0 by an A_1 with three positions), then three applications of rule (c) followed by an application of rule (b) to get rid of the A_1 and create the final string. ◇

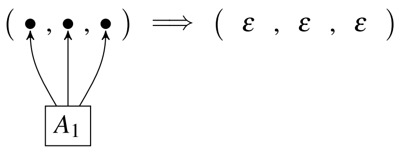
This example is sufficient to illustrate one key aspect of these types of formalisms. By allowing a non-terminal to track multiple positions in the string it is possible to create synchronized substrings. Notably the structure of Example 3.1 is similar to what is needed to generate the language

$$\{(ww^{\mathscr{R}})^k \mid w \in \{a,b\}^*,\ w^{\mathscr{R}} \text{ is } w \text{ reversed}\}$$

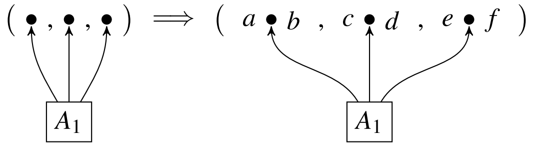
where k is determined by the arity of the non-terminals. That is, there is a grammar using non-terminals with arity at most k which for each palindromatic string p



(a) The "initial" rule, replacing the starting (arity 1) non-terminal by the arity 3 non-terminal A_1, in essence "splitting" the string into three parts controlled simultaneously by A_1.



(b) A finishing rule, which allows A_1 to just generate the empty string in all three of its positions. Taking only the rule in (a) and this one creates a language that generates the empty string.



(c) The third rule is the only one that actually generates symbols, replacing A_1 by a new copy with positions in the middle of a string generated for each position.

Figure 3.4: The three rules for a simple variation of a hyperedge replacement string grammar. These three rules together generate exactly the language $a^n b^n c^n d^n e^n f^n$.

contains the string which results from repeating p k times. The language in Example 3.1 is well known not to be context-free, and the palindrome repetition language more closely illustrates how the formalism allows separate parts which are in themselves (in some vague sense) context-free, but share finite control through the same non-terminal controlling more than one position (as was illustrated in Figure 1.6).

Example 3.5 (More Complex Hyperedge Replacement Rules) To give a more nuanced picture of the formalism, consider also the rule in Figure 3.6. When added to

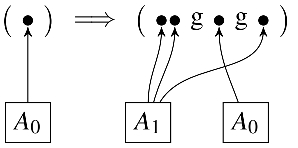


Figure 3.6: Adding this rule to the three in Figure 3.4 creates a more complicated language which illustrates the power of these grammatical formalisms.

rules (a)–(c) from Figure 3.4 this makes it possible to generate a sentential form like the one shown in Figure 3.7.
This new grammar can then generate for example all strings of the form

$$a^n b^n c^n d^n g a^m b^m c^m d^m g a^l b^l c^l d^l e^l f^l g e^m f^m g e^n f^n,$$

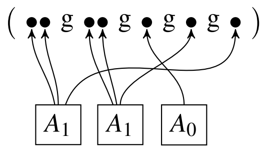


Figure 3.7: A configuration reachable using the rules in Figure 3.4 plus the rule in Figure 3.6.

for independent integers n, m and l. Arbitrarily deep nesting of this form is possible in the grammar. ◇

3.3 The Membership Problem

The membership problem for this type of formalism can, in contrast to the CFSA case, be decided in polynomial time in the non-uniform case. This can be shown using a construction from 2001 by Bertsch and Nederhof [BN01]. This construction (slightly adapted) checks if a string w can be generated by a given hyperedge replacement string grammar G by generating a vast context-free grammar which is non-empty if and only if $w \in \mathscr{L}(G)$. For example, let G be the hyperedge replacement string grammar in Figure 3.4, and pick $w = \alpha_1 \cdots \alpha_n$ as the string to be parsed. Then the context-free grammar will have the non-terminals $\{A_0(i,j) \mid i,j \in \{0,\dots,n\}\} \cup \{A_1((i_1,j_1),(i_2,j_2),(i_3,j_3)) \mid i_1,j_1,i_2,j_2,i_3,j_3 \in \{0,\dots,n\}\}$, meaning that for each non-terminal A_i in G, with arity $a = arity(A_i)$, we construct $(n+1)^{2a}$ non-terminals in the context-free grammar. The construction then adds rules such that the non-terminal $A_1((i_1,j_1),(i_2,j_2),(i_3,j_3))$ can derive ε if and only if G permits the derivation in Figure 3.8 That is, this generated non-terminal represents the statement "A_1 can, in its

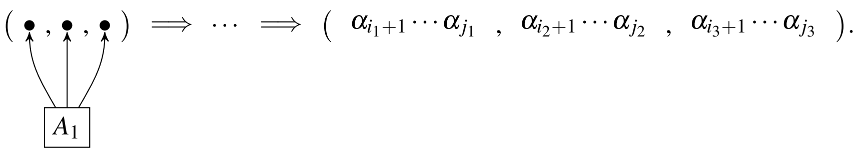

$(\bullet , \bullet , \bullet) \Longrightarrow \cdots \Longrightarrow (\alpha_{i_1+1} \cdots \alpha_{j_1} , \alpha_{i_2+1} \cdots \alpha_{j_2} , \alpha_{i_3+1} \cdots \alpha_{j_3}).$
A_1

Figure 3.8: There exists a derivation (a sequence of rule applications) starting with an instance of the non-terminal A_1 such that it generates the terminal strings $\alpha_{i_1+1} \cdots \alpha_{j_1}$, $\alpha_{i_2+1} \cdots \alpha_{j_2}$, $\alpha_{i_3+1} \cdots \alpha_{j_3}$, in the first, second, and third controlled position, respectively.

three controlled positions, generate the substrings at positions (i_1,j_1), (i_2,j_2), and (i_3,j_3) in the input string w". The rules generated by the construction simply attempt all ways to assign the substrings, so there is, corresponding to rule (a) in Figure 3.4, there are for all $i,j,x_1,x_2 \in \{0,\dots,n\}$ a rule $A_0(i,j) \to A_1((i,x_1),(x_1,x_2),(x_2,j))$.

Turning to rule (c) there is a rule

$$A_1((i_1,j_1),(i_2,j_2),(i_3,j_3)) \to A_1((i_1+1,j_1-1),(i_2+1,j_2-1),(i_3+1,j_3-1))$$

for all $i_1,j_1,i_2,j_2,i_3,j_3 \in \{0,\dots,n\}$ such that $\alpha_{i_1+1}=a$, $\alpha_{j_1}=b$, $\alpha_{i_2+1}=c$, $\alpha_{j_2}=d$, $\alpha_{i_3+1}=e$, and $\alpha_{j_3}=f$. Similarly, hyperedge replacement rules that split a non-terminal into several are represented by just enumerating every possible way to delegate the substrings among them. The initial non-terminal is $A_0(0,n)$, corresponding to the statement that A_0 can generate the entire string w. This should illustrate how this construction works, and shows how the non-uniform membership problem can be decided in polynomial time. Constructing and emptiness-checking a context-free grammar of size $\mathscr{O}(n^c)$, where c is determined by G, can be done in polynomial time when G is considered to be a constant.

Still, this also illustrates how the non-uniform membership problem being efficiently computable does not imply real-world efficiency unless the grammar genuinely is a trivial part of the problem considered. Here the uniform case is indeed NP-hard. This is not difficult to see, for example by a reduction from the longest common subsequence problem (a classic NP-complete problem [GJ90]). Without delving deeply into the reduction, we can for each $k \in \mathbb{N}$ construct a hyperedge replacement string grammar G such that $a^n\$w_1\$\cdots\$w_k \in \mathscr{L}(G)$ if and only if the strings $w_1,\dots,w_k \in \Sigma^*$ have a common subsequence of length n. Thus, the $a^n\$$ prefix of the constructed string represents n in unary form. This construction works by giving G a non-terminal A_1 with $arity(A_1) = k+1$, and starting the derivation by setting up the sentential string $\bullet\$\bullet\$\cdots\$\bullet$, where a single instance of A_1 controls all the positions. This A_1 instance may generate symbols arbitrarily in all its positions *except* the first one, always keeping control of the position to the right of the newly generated symbol (so we can reach for example the sentential string $\bullet\$b\bullet\$ccb\bullet\$\bullet\$\cdots\$bc\bullet$). The derivation will only ever generate a symbol (and then only the symbol a) in the first position, thus effectively increasing n by 1, if it simultaneously generates some symbol α in *all* the other positions (for example reaching the configuration $a\bullet\$bd\bullet\$ccbd\bullet\$d\bullet\$\cdots\$bcd\bullet$). The derivation can terminate at any time by generating ε in all positions. This establishes the NP-completeness of the uniform membership problem for hyperedge-replacement string grammars even for the special case of linear grammars. It is important, however, to notice that this reduction requires the input value k to be embedded in the grammar, meaning that it only works in the uniform case.

The NP-hardness of the uniform membership problem does itself show that these formalisms cannot represent the same languages as a CFSA, unless $P = NP$. The membership problem is also *in* NP, this is easily established by noticing that all strings can be derived in a polynomial number of derivation steps, which allows the entire derivation to be guessed to demonstrate membership.

Chapter 4
Conclusions and Future Work

To wrap up the discussion about the two classes of formalisms considered in Chapters 2 and 3 we return to the aspects already discussed in the introduction to summarize their similarities and look to the future.

4.1 Comparing Formalisms and the Power of Ordering

There are some conclusions to be drawn from this look at shuffle-related formalisms and the synchronized substrings formalism. As was already touched upon in the introduction they share some key properties, first and foremost having a semi-linear Parikh image. Let us fully recall Parikh's definition [Par66].

Definition 4.1 (Parikh Image) Let Σ be an alphabet, fix an arbitrary order of the symbols in $\Sigma = \{\alpha_1, \ldots, \alpha_n\}$. Then for any string w over Σ the Parikh image is the n-vector $[x_1, \ldots, x_n] \in \mathbb{N}^n$ which is such that for all $i \in 1, \ldots, n$ there are exactly x_i occurrences of the symbol α_i in w. The Parikh image of a language L is the set of Parikh images of all strings in L. $\diamond$

All formalisms discussed here generate languages which necessarily have semi-linear Parikh images. That is, the Parikh image of the language is a finite union of linear sets, and a linear set is of the form $\{v_0 + p_1v_1 + \cdots p_mv_m \mid p_1, \ldots, p_m \in \mathbb{N}\}$ for some fixed vectors $v_0, \ldots, v_m \in \mathbb{N}^n$ and integer m.

The original definition of mildly context-sensitive languages does not actually require that the Parikh images are semi-linear [Jos85]. It instead requires that for each language in the class the lengths of the strings form a semi-linear set, as was noted in Section 3.1. The two language different classes representable by these formalisms both feature semi-linear Parikh images however, and this strictly stronger requirement is in many ways more natural.

It is easy to see that the language $a^nb^nc^nd^n$ cannot be generated by a CFSA, but on the other hand, a CFSA *can* generate the language of all strings containing equally many *a*s, *b*s, *c*s, and *d*s (which is the largest possible language which has the same Parikh image as $a^nb^nc^nd^n$). Let us denote this language L_{abcd}. It can be generated by simply repeatedly shuffling the language $a \odot b \odot c \odot d$ with itself. Both classes of mildly context-sensitive language formalisms discussed in Section 3.1 can generate $a^nb^nc^nd^n$, though the weaker class, equivalent to tree adjoining grammars, cannot

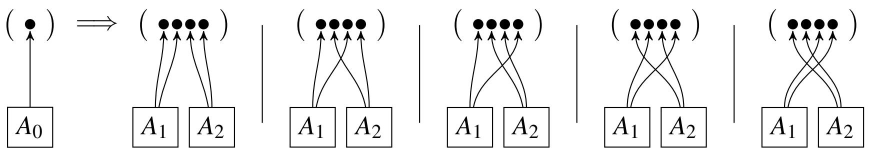


Figure 4.2: A set of five hyperedge replacement rules in the style of Section 3.2 (the five rules have the same left-hand side, each of the right-hand sides are separated by a vertical bar). These rules allow A_0 to generate all possible ways to generate one instance each of A_1 and A_2 with arbitrarily interleaved positions.

generate $a^n b^n c^n d^n e^n$. In the other direction, the author conjectures that L_{abcd} cannot be generated by either of the mildly context-sensitive formalisms discussed in Chapter 3. One argument why it is unlikely that the hyperedge replacement formalism from Section 3.2 can generate arbitrary shuffles is explained by Figure 4.2. Intuitively the hyperedge replacement grammar can, by a large set of rules, arbitrarily interleave positions of non-terminals. Figure 4.2 shows the five rules that allow two non-terminals A_1 and A_2, each with arity 2, to be interleaved. However, after this point the two non-terminal instances no longer have any means of communicating, so a decision which of the non-terminals generates what part of the shuffle has to be encoded in the non-terminal itself. The problem that arises is that it seems unlikely that every shuffle language is such that there exists a constant k, such that every string in the language can be broken into $2k$ pieces, which are then divided into two sets of k substrings each, and those sets have a finite description in the grammar. This is however something that appears to be required for languages that can be represented by the formalisms of Chapter 3, where k is the maximum arity of the non-terminals (which is fixed in every grammar). Making proofs for this type of question for the synchronized substrings formalisms is an interesting direction of future research, it seems probable that much can be achieved using the pumping lemma for hyperedge replacement grammars [DHK97].

This may be raising more questions than it answers. The two classes of formalisms discussed in Chapter 2 and Chapter 3 have a lot in common, both in the way they generate strings (one by allowing multiple independent pieces of control to read from the string in an uncontrolled fashion, another which assigns each independent control several but fixed positions), and are more powerful than context-free languages in an intuitively similar way. They also both have interesting membership problems, being around the edge of what is possible in polynomial time with appropriate restrictions. As such they are an interesting future direction of research.

4.2 Future Plans

The first future direction is investigating the membership problem for the mildly context-sensitive languages, notably attempting to give a nuanced view of the intractability of the uniform membership problem. Similarly, the hunt for classes of shuffle languages for which the membership problem is efficiently decidable continues, with some special languages of particular interest:

- the shuffle of palindromes, $\{ww^{\mathscr{R}} \odot ww^{\mathscr{R}} \mid w \text{ is any string, } w^{\mathscr{R}} \text{ is } w \text{ reversed}\}$,
- the shuffle square $\{w \odot w \mid \text{any string } w\}$.

Both of these languages appear very straightforward, but the difficulty of the membership problem for them remains an open question. To illustrate the problem, consider the following backtracking algorithm for deciding whether a string is the shuffle of two palindromes. It runs reasonably quickly for small examples, but takes exponential time in the worst case.

Algorithm 4.3 (Palindrome Shuffle Membership Test)

```
 1: function IsPalShuffle(string $\alpha_1 \cdots \alpha_n$, optional string $\beta_1 \cdots \beta_m$)
 2:     if $n = 0$ then
 3:         return IsPalindrome($\beta_1 \cdots \beta_m$)
 4:     end if
 5:     if $m > 0$ and $\alpha_1 = \beta_m$ and IsPalShuffle($\alpha_2 \cdots \alpha_n$, $\beta_1 \cdots \beta_{m-1}$) then
 6:         return True
 7:     end if
 8:     for i = n ... 1 do
 9:         if $\alpha_1 = \alpha_i$ and IsPalShuffle($\alpha_2 \cdots \alpha_{i-1}$,$\alpha_{i+1} \cdots \alpha_n \beta_1 \cdots \beta_m$) then
10:             return True
11:         end if
12:     end for
13:     return False
14: end function

15: function IsPalindrome(string $\alpha_1 \cdots \alpha_n$)
16:     return $\forall(i \in \{1, \ldots, \lfloor \frac{n}{2} \rfloor\}) : \alpha_i = \alpha_{n-i}$
17: end function
```

For any string $\alpha_1 \cdots \alpha_n$ the call IsPalShuffle$(\alpha_1 \cdots \alpha_n, \varepsilon)$ returns true if and only if $\alpha_1 \cdots \alpha_n$ is the shuffle of two palindromes. Proving the correctness of the algorithm is not within the scope of the discussion, but a short overview is in order. The algorithm works from the left, attempting to in each call match the first symbol in the string to its "mirror", that is, the other occurrence of the symbol in the palindrome, speculatively removing them, and recursing to check that it was correct (backtracking if it was not).

Consider a top-level call IsPalShuffle(w, ε), assume for now that w *is* the shuffle of two palindromes, call them palindrome A and palindrome B. We assume that the first symbol of w is part of palindrome A (by symmetry). Now assume that we are in a recursive call IsPalShuffle$(\alpha_1 \cdots \alpha_n, \beta_1 \cdots \beta_m)$. Then there exists a string s such that $w \in s \cdot \alpha_1 \cdots \alpha_n(\beta_1 \cdots \beta_m \odot s^{\mathscr{R}})$ where $s^{\mathscr{R}}$ is s reversed. At this point s is the part of the string already processed, and the hypothesis of the current call is that all the symbols $\beta_1 \cdots \beta m$ are part of palindrome B, whereas the substring $\alpha_1 \cdots \alpha_n$ remains to be processed.

Let us look at each possibility at this point. First if $n = 0$ (see line 2), and $\beta_1 \cdots \beta_m$ is a palindrome, then palindrome A has already been consumed, and $\beta_1 \cdots \beta_m$ is the "center" of palindrome B, this means that we are done, w was the shuffle of two palindromes. If $m > 0$ and $\alpha_1 = \beta_m$ (see line 5) it is possible that α_1 belongs to palindrome B by pairing α_1 to β_m (notice that we do not need to check β_i for $i < m$, since that would leave β_m impossible to match). In that case remove both and recursively check if this is part of the solution, otherwise backtrack and check the next part. Next we consider the possibility that α_1 is part of palindrome A, which requires that there exists some i such that $\alpha_1 = \alpha_i$ (see line 8), and all the symbols α_j with $j > i$ have to be part of palindrome B (while $\alpha_2 \cdots \alpha_{i-1}$ remains to be processed).

As an example, the call $\text{IsPalShuffle}(abccdadb, \varepsilon)$ matches the as to each other using the line 8 loop, so palindrome A is of the form axa for some string x. It then recursively calls $\text{IsPalShuffle}(bccd, db)$, which matches the bs on line 5, so palindrome B is on the form byb for some string y. It then recursively calls $\text{IsPalShuffle}(ccd, d)$, which matches the cs (line 8), so palindrome A is $acca$. Finally, the recursive call to $\text{IsPalShuffle}(\varepsilon, dd)$ which runs the check on line 2, confirms that dd is a palindrome, meaning that palindrome B is $bddb$, and indeed $abccdadb \in acca \odot bddb$.

In this way the algorithm attempts all possible ways to divide the string into two shuffled palindromes with backtracking, which unfortunately makes it take exponential time in the worst case. Conveniently it is quite effective in practice, which allows for interesting experimentation.

Finally, the previous section raised some interesting questions about the languages in the intersection between these mildly context-sensitive languages and the shuffle languages. There is a lot of room for investigating this area.

Chapter 5
Summary of Papers

This chapter will give a short overview of each of the three papers, Paper I through Paper III, which are included as appendices. They are all in some way concerned with the way ordering has an impact on formal languages, as well as how to allow only limited changes to a core language (only a fixed number of reordering or derivation modifications), all while focusing on the membership problem for the resulting formalisms.

Each paper is best viewed from the perspective of the membership problem it treats, though other interesting properties are considered as well. The CFSA model discussed at length in Chapter 2 is a the first key direction, but there are other interesting facets. The CFSA membership problem is difficult in general, but this was fully expected, as the general CFSA model is more powerful than what is motivated for most practical cases. It instead serves as a good basis for experimentation, allowing various more limited cases to easily be expressed as restrictions on CFSA. There are both successes and failures (in the sense of negative results being proven) represented in the papers, among the successes are restricted formalisms for which the non-uniform membership problem is solvable in polynomial time, and arguably the result that the membership problem for CFSA is *in* NP is positive as well. On the negative side one paper is dedicated to demonstrating that a restricted CFSA model (the shuffle of two linear deterministic context-free languages) still has an NP-hard non-uniform membership problem. Another direction considered is the problem of allowing only a limited amount of reordering (and other changes) in each string. For this to have any meaning one has to consider some possible reordering of the string to be the "right" one, much like the characteristic context-free grammars for CFSA from Definition 2.11. That is, we have a canonical "correct" grammar, but want to allow strings that are, in some sense, *slightly* wrong into the language. Here the origin of the question is; if the string w can be derived by a given context-free grammar, can w' be derived by swapping the positions of at most k non-terminals in sentential forms in the derivation? It is useful to think about this limited amount of allowed reordering as allowed but undesirable, each reordering operation adds "badness". That is, it is a correction problem, the reorderings are considered errors and a strict grammar is to be loosened such that "reasonably" incorrect strings are allowed. Ideally a formalism similar to the CFSA would be able to play this role, but as will be seen it is unfortunately a difficult problem even for a given single parse tree where swapping adjacent siblings is the reordering allowed.

Table 5.1: The closure properties of a CFSA demonstrated in Paper I.

Operation	**Closed**	**Not closed**
Union	×	
Concatenation	×	
Kleene closure	×	
Shuffle	×	
Shuffle closure	×	
Intersection		×
Complementation		×

The papers do not feature any pure appearance of the synchronized substrings formalisms, they remain lurking in the background as a future direction. Let us get on with summarizing each paper individually.

5.1 Paper I: Recognizing Shuffled Languages

5.1.1 Introduction

Paper I is the starting point used in Chapter 2, introducing the Concurrent Finite-State Automata formalism (which was loosely illustrated in Figure 1.5). The following aspects are then considered

- the closure properties of CFSA under various operations,
- what language classes are obtained by syntactic constraints on a CFSA, and
- the complexity of the membership problem for CFSA, both in the general case and for constrained classes.

5.1.2 CFSA Closure Properties

The closure properties of CFSA are illustrated in Table 5.1 The operations have their usual meanings (for example, the complement of a language $L \subseteq \Sigma^*$ is the set $\bar{L} = \{w \in \Sigma^* \mid w \notin L\}$. The two that may need explanation are the shuffle operations. The shuffle of two languages L and L' is the set of all interleavings of any string in L with any string in L', that is $\bigcup\{w \odot w' \mid w \in L, w' \in L'\}$, where $\odot$ is as in Definition 2.2. The shuffle closure of a language L contains exactly the empty string ε and all strings $w \odot w'$ where $w \in L$ and w' is in the shuffle closure of L. None of the positive cases in Table 5.1 are very surprising, being quite directly expressible in a CFSA (for example, any CFSA can be made to accept its shuffle closure by adding the rule $q_0 \xrightarrow{\varepsilon} q_1[q_0^*]$ and $q_1 \xrightarrow{\varepsilon} \varepsilon$ where q_1 is a new state (and q_0 is the initial state).

5.1.3 CFSA Constraints

It was already illustrated in Example 2.6 how a CFSA can accept the language $a^n b^n$. CFSA of this form, where only a single state can make transitions at any time (the state string forms a monadic tree) can represent exactly the context-free languages. Clearly, the regular languages are exactly the ones that can be recognized by a CFSA where no state string ever contains more than one state. A CFSA that has limited nesting depth, that is, no symbol can be surrounded by arbitrarily many matched pairs of brackets, corresponds directly to the shuffle expressions, covered in Section 2.4. This finite nesting can be enforced by making sure that for each rule $q_0 \xrightarrow{\alpha} q_1[q_2 q_3]$ it holds that no bracketed state (q_2 and q_3 here) can produce the state on the left hand side (q_0) again. This can intuitively be transformed into a syntactical constraint by giving the states a ranking and requiring that no lower-ranked state may produce a higher-ranked one.

5.1.4 CFSA Membership Problems

Finally Paper I discusses the membership problem for various constrained and unconstrained CFSA. It establishes $W[1]$-hardness[1] of the uniform membership problem for shuffle expressions (finitely deeply nesting CFSA). The parameterization chosen is to have the parameter k be the length of the longest state string that can be produced in any run of the CFSA (so fixing k neither infinitely deeply branching or rules which can produce arbitrarily large strings, i.e. right hand sides of the form $q_1[q_2^*]$, are allowed). Second, the uniform membership problem for the shuffle of a regular language and a context-free language is decidable in polynomial time. Third, the non-uniform membership problem for the shuffle of a shuffle expression and a context-free language is decidable in polynomial time (for the uniform case the membership problem for shuffle expressions is already NP-complete [Bar85, MS94]).

It is shown that the uniform membership problem for arbitrary CFSA is *in* NP, a non-trivial result. It is also shown that the problem is NP-hard. This, however, turned out to be a known result [ORR78]. A strengthened version of this result could be obtained in Paper II.

5.2 Paper II: The Membership Problem for the Shuffle of Two Deterministic Linear Context-Free Languages is NP-complete

Paper II builds on the results in Paper I to demonstrate that the non-uniform membership problem for the shuffle of two deterministic linear context-free languages is NP-complete. The proof works by enforcing a strict interleaving by a mix of matching brackets and delimiters, having one deterministic linear context-free grammar generate single computation steps of a universal Turing machine, while the other grammar performs copying linking the steps up into a complete computation. The proof tech-

[1] A fact that strongly suggests that the problem is not *fixed-parameter tractable*. That is, that even if the number of actual shuffling operations involved is limited the membership problem remains hard, unless $W[1] = FPT$, which is believed unlikely. For more on fixed parameter complexity theory, see, e.g., [DF99].

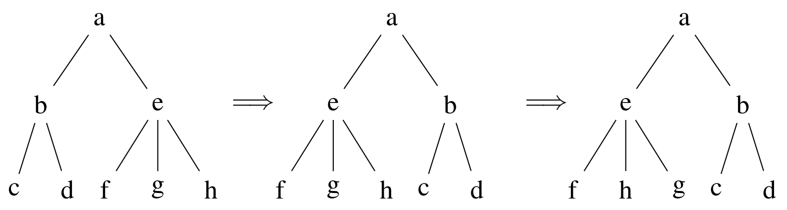


Figure 5.2: An example of tree swaps, transforming the left-most tree into the right-most tree by first swapping the position of the "b" and "c" nodes (being adjacent siblings) and then the "g" and "h" nodes. We say that the right tree is two swaps away from the left tree.

nique is reminiscent of the classic proof that every recursively enumerable set is the homomorphic image of the intersection of two linear context-free languages [BB74].

5.3 Paper III: Analyzing Edit Distance on Trees: Tree Swap Distance is Intractable

Paper III takes a different direction, it is in effect an attempt to find formalism which

1. allows a hierarchically structured mode of reordering,
2. makes it possible to constrain the language to only a limited amount of reordering, and
3. still has an efficiently decidable membership problem.

The specific type of reordering considered in this paper is swapping sibling node positions in a tree. See for example Figure 5.2. This can be viewed as a way to reorder a string by letting the leaf nodes represent the string. We consider each "swap", that is, interchange of adjacent sibling nodes, as having a cost of one. The problem considered is: given an integer k and two trees t and t', can t be transformed into t' using at most k swaps? That is, is t in the finite language defined by applying at most k swaps to t'?

This tree problem does attempt to model both the first and second requirement listed at the start of the section. Unfortunately, the main result of the paper is that this problem is NP-complete, and as such it probably fails to fulfill the third requirement.

Both the shuffle and the synchronized substrings formalism feature possible solutions for both the first and third points, with different ideas of reordering, and different possible restrictions to allow efficient parsing. Neither of those formalisms are helpful when it comes to the second requirement however, and for some very practical problems the second requirement is very important. For example, consider output trees of a statistical natural language parser and how to "fix" these so that they are grammatically correct. In this setting fewer changes are clearly more desirable, as fewer corrections means that it is more likely that the original meaning is preserved.

Since the paper only provides an intractability result for the tree swap distance problem it remains unclear whether all three requirements can be simultaneously be fulfilled (and how).

References

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

[BB74] Brenda S. Baker and Ronald V. Book. Reversal-bounded multipushdown machines. *J. Comput. Syst. Sci.*, 8(3):315–332, 1974.

[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.

[DF99] Rod G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[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.

[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.

[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.

[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.

[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.

[Par66] Rohit Parikh. On context-free languages. *J. ACM*, 13(4):570–581, 1966.

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

[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.

[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.

I

Recognizing Shuffled Languages[☆]

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, i.e., 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 $\{abcd, acbd, acdb, cabd, cadb, cdab\}$. 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. As our starting point, we take the *shuffle languages* considered by Gischer [19] and by Jedrzejowicz and Szepietowski [26]. These are the languages defined by regular expressions augmented with the shuffle and the shuffle closure operators.

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

- In the *modelling* and *verification* of systems, shuffling is useful for reasoning about interleaved or parallel processes [16, 30]. There is a close connection between shuffle languages and Petri nets [19, 16, 7, 5].
- The shuffle operator is used in *XML database systems* for schema definitions, see, e.g., Gelade et al. [17].

[☆]The present article is the full version of [2], 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

- In *plan recognition*, the objective is to identify an agent's goal or plan, based on observations of the agent's actions [10, 35]. In a generalized version, a number independent agents that perform their actions in an interleaved fashion. To model this multi-agent scenario one could combine shuffle operators and context-free grammars [24]. 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 ordering. Recent work in this direction includes parse algorithms for so-called dependency grammars [34, 28].

A number of fundamental questions regarding the membership problem for shuffled languages remain unanswered. We 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 therefor 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 linear-bounded Turing machines. These automata can be viewed as ground tree rewriting systems (see, e.g., [29, 11]) 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 [19, 26]), the *uniform membership problem* is NP-complete [1, 31], while the *non-uniform membership problem* can be decided in polynomial time [26]. 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 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.

Table 1: Summary of results for the membership problem. The shuffle languages are abbreviated by Sh, the regular by Reg, and the context-free by CF. The results of this paper appear in bold face.

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

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. The non-uniform problem is, however, solvable in polynomial time. For the shuffling of two context-free languages, we show that already the non-uniform version of the membership problem is NP-hard.

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. Câmpenau et al. establish $2^{mn} - 1$ as a tight upper bound on the state complexity of the shuffle of two regular languages [9]. Biegler et al. provide a similar result for singleton languages and identify properties that trigger an exponential blow-up in state complexity [4]. 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 [21]. 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. [13]. Brozozowski et al. consider the complexity of *ideal* languages [8], which are regular languages invariant under shuffle with the universal language [22]. Further results for sub-families of the regular languages are found in [20, 23, 3, 12].

Shuffling has also been investigated in a more algebraic setting. The axiomatization of shuffle theory was addressed by Ésik together with Bloom [6] and Bertol [15]. Kari and Sosík consider the effects of shuffling on *trajectories*; sets of binary strings representing positions [27].

Another related formalism is *permutation languages*, considered by Nagy [32, 33], which allow rules of the form $AB \to BA$ in an otherwise normal context-free grammar. That is, there may be rules which can be applied to interchange positions of adjacent non-terminals in intermediary derivation steps. These can be used to simulate some types of shuffling, and we will see that the membership problems for the permutation languages are related to the membership problems for a certain class of shuffle languages.

2. Preliminaries

Sets and numbers. If S is a set, then 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 $\mathit{dom}(f)$.

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 $\alpha \in \Sigma$, $|w|_\alpha = |\{i \in [n] \mid \alpha_i = \alpha\}|$.

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 $1 + \max(|v| \mid v \in \mathit{nodes}(t))$.

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

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

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

For a tree $t \in T_\Sigma$ let $v_1, \ldots, v_k \in \mathit{nodes}(t)$ be the immediate child nodes of the root ordered by numeric value. That is, $\{v_1, \ldots, v_k\} = \{v \in \mathit{nodes}(t) \mid |v| = 1\}$, 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 $\mathit{nodes}(t) = \{\varepsilon\}$), 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 [19, 26].

The *shuffle* operation $\odot : \Sigma^* \times \Sigma^* \to \mathit{pow}(\Sigma^*)$ 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, 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 $|nodes(t)| = 0$,
  - *horizontal* if $|nodes(t)| = 1$, and
  - *vertical* if $|nodes(t)| > 1$.
- $I \subseteq Q$ is a set of *initial* states. □

Remark. For simplicity and without loss of generality, we henceforth assume that the terminal transitions form a subset of $Q \times \{\varepsilon\} \times \{t_\varepsilon\}$. It is easy to see that every CFSA can be rewritten to this normal form in linear time.

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 $v \in \mathit{nodes}(t)$ 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.

Example 1. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be the Dyck languages[2] 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 $M = (\{q_0, q_1, q_1', q_2, q_2'\}, \{\lfloor, \rfloor, \lceil, \rceil\}, \delta, \{q_0\})$, where

$$\begin{array}{llllll}\delta = \{ & (q_0, \varepsilon, q'[q_1, q_2]), & (q_0', \varepsilon, t_\varepsilon), & (q_1, \lfloor, q_1'[q_1]), & (q_1', \rfloor, q_1), & \\ & (q_1, \varepsilon, t_\varepsilon), & (q_2, \lceil, q_2'[q_2]), & (q_2', \rceil, q_2), & (q_2, \varepsilon, t_\varepsilon) & \} \ .\end{array}$$

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. Thus 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.*

[2]A Dyck language consists of all well-balanced strings over a given set of parentheses.

In its initial configuration, M is in the unique initial state q_0 and has yet to consume any input symbol.

$$(\ \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ q_0)$$

To proceed, M must nondeterministically choose the transition $(q_0, \varepsilon, q_0'[q_1, q_2])$.

$$(\ \lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ q_0'[q_1, q_2]\ )$$

By the transition $(q_1, \lfloor, q_1'[q_1])$, M reaches the configuration

$$(\ \lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor,\ q_0'[q_1'[q_1], q_2]\ )$$

and by $(q_1, \lfloor, q_1'[q_1])$ and $(q_2, \lceil, q_2'[q_2])$ the configuration

$$(\ \rfloor\lfloor\rceil\rfloor\rfloor,\ q_0'[q_1'[q_1'[q_1]], q_2'[q_2]]\ )$$

Now M nondeterministically guesses that it is time to read the symbol $\rfloor$. It prepares by deleting the leaf labelled q_1 using transition $(q_1, \varepsilon, t_\varepsilon)$ to get

$$(\ \rfloor\lfloor\rceil\rfloor\rfloor,\ q_0'[q_1'[q_1'], q_2'[q_2]]\ )$$

and then $(q_1', \rfloor, q_1)$ to get

$$(\ \lfloor\rceil\rfloor\rfloor,\ q_0'[q_1'[q_1], q_2'[q_2]]\ )$$

Again, $(q_1, \lfloor, q_1'[q_1])$ lets M read $\lfloor$,

$$(\ \rceil\rfloor\rfloor,\ q_0'[q_1'[q_1'[q_1]], q_2'[q_2]]\ )$$

and $(q_2, \varepsilon, t_\varepsilon), (q_2', \rceil, q_2)$ produces

$$(\ \rfloor\rfloor,\ q_0'[q_1'[q_1'[q_1]], q_2]\ )\ .$$

Thereafter, applying the transition sequence $(q_1, \varepsilon, t_\varepsilon), (q_1', \rfloor, q_1)$ twice yields

$$(\ \varepsilon,\ q_0'[q_1, q_2]\ )\ .$$

Although the entire input has been read, M does not accept until the state tree has been reduced to the empty tree. This can be done by applying $(q_1, \varepsilon, t_\varepsilon), (q_2, \varepsilon, t_\varepsilon)$ to get

$$(\ \varepsilon,\ q_0'\ )\ ,$$

and finally $(q_0', \varepsilon, t_\varepsilon)$ to reach

$$(\ \varepsilon,\ t_\varepsilon\ )\ .$$

Figure 1: The CFSA M of Example 1 accepts the input string $\lfloor\lfloor\lceil\rfloor\lfloor\rceil\rfloor\rfloor$.

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. The classical construction of a nondeterministic automaton for the union of M and M' carries over from the FSA case: a new initial state q is added, together with ε-transitions from q to each of q_0 and q_0'.

Concatenation. For concatenation, we add the new states q, q', and q'', where q becomes the initial state of the new automaton. We also add the vertical transitions $(q, \varepsilon, q'[q_0])$ and $(q', \varepsilon, q''[q_0'])$, and the terminal transition $(q'', \varepsilon, t_\varepsilon)$. This allows the automaton to first simulate a run of M and then a run of M'.

Kleene closure. Next, we construct a CFSA for the Kleene closure of M. All we have to do is add a new, unique, initial state q to Q along with the terminal transition $(q, \varepsilon, t_\varepsilon)$ and the vertical transition $(q, \varepsilon, q[q_0])$. This allows the automaton to simulate any number of runs of M, one after the other.

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 $\{v \mid uvw \in \mathcal{L}$ 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 $G = (N, \Sigma, \gamma, S)$ on 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 acyclicity.

Given a CFSA $M = (Q, \Sigma, \delta, I)$ we show how to construct a shuffle expression s recognizing $\mathcal{L}(M)$. Two states $q, q' \in Q$ 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 [18], we can conclude that acyclic and finitely branching CFSA also recognize the regular languages.

The next theorem shows that CFSA do not provide us with the full power of linear bounded Turing machines.

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 an input string. If there is an accepting run of M on w from some initial state q_0, then a nondeterministic Turing machine can guess and verify this run in linear space by proceeding as follows. (1) The TM simulates a run of M on w starting in q_0, but every time a vertical transition $(q, \alpha, q'[s])$ is used on the top level, where s is a sequence of labels, the TM guesses what part of the subsequent string is to be consumed by the states trees derived from s, marks this segment off with brackets and a pointer to s, and continues in state q' after the closing bracket until it has read all of w. If it accepts what it has seen so far, it goes on to verify each of the bracketed segments.

Let w' be such a segment, annotated with s. 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 $u' \in \mathcal{L}(M_{p'})$. Finally, if $s = p^{\odot}$, the TM guesses an $n \leq |w'|$ and a way to partition w' into n subsequences, and verifies recursively that each such subsequence belongs to $\mathcal{L}(M_p)$. This process continues recursively until no unprocessed bracketed segment has non-zero length. We note that the total amount of information that was recorded in the process 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 it follows that 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 $(q, \alpha, q'[s]) \in \delta$ for some $q' \in F_i$ and some s such that every state that appears in s belongs to F_i, then $q \in F_{i+1}$
5. $\mathcal{F} = \cup_{i=0}^{\infty} F_i$

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 belong 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 $\mathcal{L}(M_q)$. 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 $t_1, \dots, t_m$ is in $\mathcal{F}$. In particular, this means that q belongs to $\mathcal{F}$, since $t = q$. 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. $t_{i-1} = t[\![v \leftarrow q]\!]$, $t_i = t[\![v \leftarrow q'[u_1, \dots, u_n]]\!]$, 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. $\square$

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.*

NP-hardness for the uniform membership problem for shuffle expressions is already known; see, e.g., [1, 31]. We postpone the hardness proof for the non-uniform case until Theorem 9 in Section 4.4, where it is proved for a subclass of CFSA.

To see that the membership problem for CFSA is in NP, we first note that every CFSA can be augmented in polynomial time with "shortcuts", that is, contractions of ε-consuming transition sequences into single transitions.

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

1. For every $q \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, t_\varepsilon)$ then $(\varepsilon, q) \rightarrow (\varepsilon, t_\varepsilon)$.

2. For every choice of $q, q' \in Q$, if $(\varepsilon, q) \xrightarrow{*} (\varepsilon, q')$ then $(\varepsilon, q) \rightarrow (\varepsilon, q')$.

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 trim CFSA in polynomial time.*

Proof (Sketch). A simple procedure based on the emptiness test (recall 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 generate 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. *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 trim. 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.

For every pair of configurations $c = (\varepsilon, t)$, $c' = (\varepsilon, t') \in \Delta(M)$, if there is a sequence of transition steps from c to c', then there is also a sequence of length at most $n \leq |t| + 2|t'|$. Such a short sequence can be found by organizing the transitions as follows: $(\varepsilon, t) \xrightarrow{*} (\varepsilon, \hat{t}) \xrightarrow{*} (\varepsilon, t')$ 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 trim, 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.

Finally, 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 $|Q|(|w| + 1)$. Only $|w|$ symbols are consumed by the transitions, so if there are $|w| + 1$ leaves, then one of them must eventually 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 to have $|w|+2$ or more copies of some state q along some path. By a standard pumping argument the sequence could have chosen not to recognize any q-delimited section of the path (that is, loop once less on q). 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 of transition steps was of minimum length.

In conclusion, 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 [26], that non-uniform parsing for shuffle expressions is polynomial. For the uniform membership problem, membership in NP is obvious – just guess and verify a run of the automaton. NP-hardness follows by an easy adaptation of a result by Barton [1]. □

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, e.g., the similar NFA(&) from [17], 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 [26]. 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 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., [14].

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 $w \in \mathcal{L}(M)$. $\square$

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 [14].

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. $\square$

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 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

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

Finally, we define

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

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.

- 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.
- 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.*

The above theorem actually has a shorter proof than the one given below, based on the fact that a shuffle language shuffled with a context-free language is a context-free language. We give the slightly longer proof because it is a good preparation for the proof of Theorem 8.

Proof. Let $G = (N, \Sigma, \delta, S)$ and $M = (Q, \Sigma, \gamma, I, F)$ be a context-free grammar on Chomsky normal form and an NFA, respectively.

To test membership in $\mathcal{L}(G) \odot \mathcal{L}(M)$, we extend the CYK algorithm for context-free grammars. For every nonterminal $A \in N \cup \{\varepsilon\}$ and every pair of states $q_1, q_2 \in Q$ let $M_{q_1,q_2} = (Q, \Sigma, \gamma, \{q_1\}, \{q_2\})$ and $G_A = (N, \Sigma, \delta, A)$ unless $A = \varepsilon$, in which case $\mathcal{L}(G_A)$ contains only the empty string. Then (A, q_1, q_2) is a *parse triple* for G and M over a string w if and only if $w \in \mathcal{L}(G_A) \odot \mathcal{L}(M_{q_1,q_2})$. In other words, (A, q_1, q_2) is a parse triple for w if w can be partitioned into two subsequences, w_1 and w_2, such that A can produce w_1 (or $w_1 = \varepsilon$ if $A = \varepsilon$)

and M can read w_2 by going from state q_1 to q_2. We note that there are at most $(|N|+1) \cdot |Q|^2$ distinct parse triples.

The idea, like in the CYK algorithm, is to compute the parse triples for each substring, starting with the substrings of length 1, and then combine triples to form new triples for successively longer strings. In the end, $w \in \mathcal{L}(G) \odot \mathcal{L}(M)$ if and only if there is a parse triple (S, q_I, q_F) for the whole of w such that S is the start symbol of G, $q_I \in I$, and $q_F \in F$. Since w has $O(m^2)$ substrings we will compute at most $O(m^2 \cdot |N| \cdot |Q|^2)$ parse triples.

For substrings of length one, computing the triples is trivial. Assume that we have computed all the parse triples for all substrings of length $k-1$. We show how to compute the parse triples for a substring of length k. Let $v = v_1 \cdots v_k$ be such a substring. To find out whether (ε, q_1, q_2) is a parse triple for v, we proceed as follows. We check whether there is an $i \in [k-1]$ and a state q such that (ε, q_1, q) is a parse triple for $v_1 \cdots v_i$, and (ε, q, q_2) is a parse triple for $v_{i+1} \cdots v_k$. If this is the case, (ε, q_1, q_2) is a parse triple for v.

To determine whether (A, q_1, q_2), $A \in N$, is a parse triple for v, we proceed in two steps. First, if there is a rule $A \to a$ in δ, for some $a \in \Sigma$, we check whether there is an $i \in [k]$ and a $q \in Q$ such that $v_i = a$, (ε, q_1, q) is a parse triple for $v_1 \cdots v_{i-1}$, and (ε, q, q_2) is a parse triple for $v_{i+1} \cdots v_k$. If this is the case, (A, q_1, q_2) is a parse triple for v. Second, we check, for each rule $A \to BB'$ whether there is an $i \in [k]$ and a $q \in Q$ such that (B, q_1, q) is a parse triple for $v_1 \cdot v_i$ and (B', q, q_2) is a parse triple for $v_{i+1} \cdot v_k$. In this case too, (A, q_1, q_2) is a parse triple for v. □

Since acyclic and finitely branching CFSA only contribute 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 [26], 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 along the lines of the proof of Theorem 7.

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. $\square$

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. $\square$

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 M_s is *vertically separated.* We write $scp(M_s)$ (for shuffle-closure-parent) for the set of states that can have an unbounded number of children in configuration trees, i.e., $scp(M_s) = \{q \mid \exists p, p', \alpha : (p', \alpha, q[p^{\odot}]) \in \delta\}$. $\square$

Having covered the first step 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 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 $P(v, v')$. $\square$

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

We write $\mathit{se}(v, v')$ 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 therefor 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 $\mathit{nodes}(t)$, 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 $(t_1, w) \overset{*}{\to} (\varepsilon, t_\varepsilon)$ then $(t_2, w) \overset{*}{\to} (\varepsilon, t_\varepsilon)$ □

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

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 $\mathit{scp}(M_s)$ and have children.

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

Lemma 3. *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}

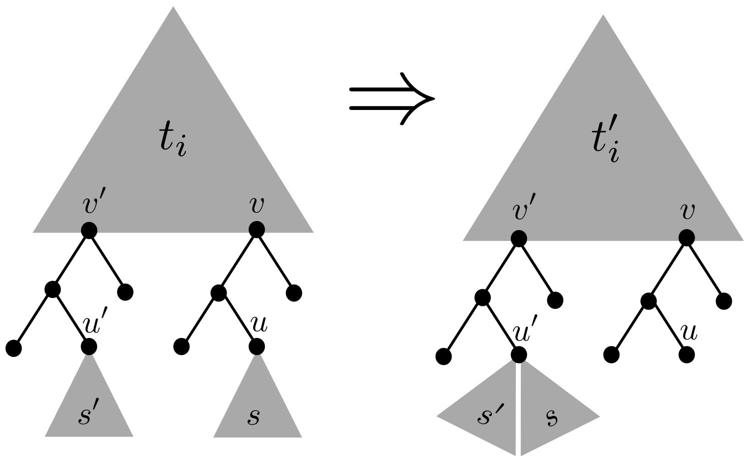


Figure 2: Children obtained through shuffle-closure can be moved between descendants of symmetrically equivalent nodes.

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 might as 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 2 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 3 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 $\mathit{scp}(M_s)$ 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 $q \in Q \setminus \mathit{scp}(M_s)$, then
$$\mathrm{cct}(t) = (q, \langle \underbrace{1, \ldots, 1}_{k} \rangle)[\mathrm{cct}(t_1), \ldots, \mathrm{cct}(t_k)] \ .$$
- If $t = q[t_1, \ldots, t_k]$ and $q \in \mathit{scp}(M_s)$, 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 depend only on the automaton, not on the input word.

Lemma 4. *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 $\mathit{height}(t)$, and thus also $\mathit{height}(\mathrm{cct}(t))$, 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 $\mathit{scp}(M_s)$-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 $|\mathit{SCFree}|$ is finite and depends only on M_s.

Let $\mathit{Layer}(i, t)$ be the tree obtained from t by removing all nodes v such that there are i or more $\mathit{scp}(M_s)$-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 $\mathit{Layer}(i, t)$ 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 $\mathit{Layer}(1, t) \in \mathit{SCFree}$ and thus $\mathit{Layer}(1, t)$ 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 $\mathit{Layer}(i, t)$ is at most e_i. We obtain $\mathit{Layer}(i+1, t)$ from $\mathit{Layer}(i, t)$ by adding trees from *SCFree* as children to $\mathit{scp}(M_s)$-labeled leaves of $\mathit{Layer}(i, t)$. For two nodes v_1 and v_2 in $\mathit{nodes}(\mathit{Layer}(i + 1, t)) \setminus \mathit{nodes}(\mathit{Layer}(i, t))$ not to be symmetrically equivalent, they must either belong to two such subtrees from *SCFree* that are not isomorphic or their closest ancestors in $\mathit{Layer}(i, t)$ belong to different equivalence classes of se. This means that in $\mathit{Layer}(i + 1, t)$ there can be no more than $e_i \cdot |\mathit{SCFree}| \cdot m$ 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 $\mathit{scp}(M_s)$-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 $\mathit{scp}(M_s)$-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 $\mathit{height}(t) \leq |Q|$ this means that the number of nodes of $\mathrm{cct}(t)$ depends only on M_s. □

Lemma 5. *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 trim. 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 4 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 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). As in the proof of Theorem 7, 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. $w' = w'_1 \odot w'_2$,
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 5 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. □

4.4. The membership problem for CF $\odot$ CF

In contrast to Corollary 3, the non-uniform version of the membership problem for $\mathcal{L}(A_1) \odot \mathcal{L}(A_2)$, where A_1 and A_2 are context-free grammars, is NP-complete. This, of course, immediately implies that the same holds for the uniform version of the problem.

Proof outline. First, we recall the definitions of push-down automata, which are equivalent to context-free grammars, and two-stack push-down automata, which are equivalent to Turing machines. These definitions are well known, but for completeness, and because we will use a slightly specialized version of the definitions, we shall include them here.

Next, Definition 16 gives a reduction from an arbitrary two-stack push-down automaton A to a push-down automaton A_{sim}, such that A accepts a string a if and only if A_{sim} accepts some string $a \cdot \$ \cdot s$, where s is a sequence of stack operations that is valid in the sense of Definition 17. The correctness of the reduction is shown as Lemma 6. The idea is that A_{sim} uses its own, single, stack to simulate the first stack in A, and, whenever A would perform an operation on its second stack, for example popping "0", A_{sim} instead reads a string encoding of that operation, for example "[pop$_0$]". This means that as long as the suffix s of stack operations behaves as a stack should (that is, the symbols popped correspond to those pushed) A_{sim} will behave just like A.

The rest of the construction ensures that A_{sim} always reads a valid sequence of stack operations. This is done by constructing the context-free language $\mathcal{L}(A_{comp})$ is constructed that contains strings of the form $\$ \cdot \hat{s}$, where $\hat{s}$ is the *complement* of a valid sequence with respect to a special template input string (see Definition 18). This input string is of the form $a' = a \cdot \$ \cdot \$ \cdot S$, where S is a template repetition of stack operations (see Definition 19). As a consequence, we have $a' \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ if and only if A accepts a in a number of steps bounded by the length of S, since A_{comp} forces A_{sim} to read only valid stack operations from its part of a' (concluded in Theorem 9).

For the remainder of this paper, we represent context-free languages by push-down automata. It is well known that we can convert any context-free grammar into an equivalent push-down automaton (and vice versa) in polynomial time [25]. We choose a simple definition of push-down automata that uses only a binary stack alphabet. No generality is lost, because ε-transitions are allowed, so the automata can simulate a richer stack alphabet by representing each symbol by some fixed-length binary string.

Definition 12 (Push-down automata). A push-down automaton (PDA) is a tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states,
- Σ is a finite alphabet of input symbols ($\varepsilon \notin \Sigma$),
- $\delta \subset Q \times ((\Sigma \cup \{\varepsilon\}) \times \{\varepsilon, 0, 1\}) \times (Q \times \{\varepsilon, 0, 1\})$ is a finite set of transitions,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ are the final states.

We write $(q, a, s, q', s') \in \delta$ as $q \xrightarrow{a,s/s'} q'$. This means that if the automaton is in state q and it can read the symbol a from the input (if $a = \varepsilon$ nothing is read) and can pop the binary value s off the top of the stack (if $s = \varepsilon$ nothing is popped) it may choose to go to state q', pushing s' onto the top of the stack (if $s' = \varepsilon$ nothing is pushed).

As usual, the computation starts in state q_0 and the machine accepts if and only if some sequence of transitions leave the automaton in a state in F when the entire input string has been read. If the stack is empty no transition that would pop a value off the stack can be taken (an automaton can clearly use a sentinel bit sequence to identify the stack bottom). $\square$

When nondeterministic push-down automata are extended to have two independent stacks, they become computationally equivalent to nondeterministic Turing machines, and can simulate each step of a Turing machine run using only a constant number of transitions: the stacks can be used to simulate the work tape by letting the first stack contain the portion of the tape to the left of the head, in reverse order, while the second stack contain the portion to the right of the head. The head can then be moved by popping a symbol from one stack and pushing it onto the other.

Definition 13 (2-PDA). A two-stack push-down automaton (2-PDA) is a tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states,
- Σ is a finite alphabet of input symbols,
- $\delta \subset (Q \times (\Sigma \cup \{\varepsilon\}) \times \{\varepsilon, 0, 1\} \times \{\varepsilon, 0, 1\}) \times (Q \times \{\varepsilon, 0, 1\} \times \{\varepsilon, 0, 1\})$ is a finite set of transitions,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ are the final states.

We write $(q, a, s_1, s_2, q', s_1', s_2') \in \delta$ as $q \xrightarrow{a,s_1/s_1',s_2/s_2'} q'$. This automaton operates just like a PDA, only with two independent stacks. $\square$

To simplify the reduction, we define some additional properties that the input 2-PDA A must exhibit, the first of which is being *input-partitioned.* This means that A starts by reading its entire input without using its second stack and then, when all input has been consumed, switches over to performing arbitrary computations using both stacks.

Definition 14 (Input-partitioning). Given a 2-PDA $A = (Q, \Sigma, \delta, q_0, F)$, an *input-partitioning* of A is a tuple $(Q_{\text{input}}, Q_{\text{compute}})$ with $Q = Q_{\text{input}} \cup Q_{\text{compute}}$, $Q_{\text{input}} \cap Q_{\text{compute}} = \emptyset$, $q_0 \in Q_{\text{input}}$, $F \subseteq Q_{\text{compute}}$, and for all transitions $q \xrightarrow{a,s_1/s_1',s_2/s_2'} q'$ in δ it holds that

- $q \in Q_{\text{input}}$ implies that $s_2 = \varepsilon$, $s_2' = \varepsilon$,
- $q \in Q_{\text{compute}}$ implies that $q' \in Q_{\text{compute}}$ and $a = \varepsilon$,
- $q \in Q_{\text{input}}$ and $q' \in Q_{\text{compute}}$ only for one unique transition, which also has $a = \varepsilon$, $s_1 = \varepsilon$ and $s_1' = \varepsilon$. □

This means that an input-partitioned automaton A starts out in a state in Q_{input}, and no transition from a state in Q_{input} ever touches the second stack, but may read input. It must then, sooner or later, take the unique switching transition $q \xrightarrow{\varepsilon,\varepsilon/\varepsilon,\varepsilon/\varepsilon} q'$, with $q \in Q_{\text{input}}$ and $q' \in Q_{\text{compute}}$, after which it will always be in some state in Q_{compute}. No transitions from states in Q_{compute} may read input, or go to a state in Q_{input}, but they may use both stacks.

Remark. In the sequel, we will, without loss of generality, assume that we have an input-partitioning for any 2-PDA used. A general 2-PDA is equivalent to a Turing machine, and input partitioning simply forces the machine to start by transferring all the input to its work tape. Clearly, every Turing machine can be rewritten into an equivalent TM that accepts the same input strings (when suitably encoded) as the starting content of its work tape. □

For convenience, we fix two alphabets that will be frequently used in the remainder of this section.

Definition 15 (Γ,$\hat{\Gamma}$). $\Gamma = \{\text{push}_0, \text{push}_1, \text{pop}_0, \text{pop}_1\}$ and $\hat{\Gamma} = \Gamma \cup \{], [, \$\}$. □

The next definition contains the key construction of this section. It shows how to, given a 2-PDA A, construct a PDA A_{sim} such that A accepts input string w if and only if there exists a special string s such that $w \cdot \$ \cdot s \in \mathcal{L}(A_{sim})$. The requirement is that s encodes a valid sequence of stack operations. The construction works by letting the stack of A_{sim} simulate the first stack of A, and making A_{sim} read the stack operations for the second stack from s.

Definition 16 (A_{sim}). Given the 2-PDA $A = (Q, \Sigma, \delta, q_0, F)$, with input-partitioning $(Q_{\text{input}}, Q_{\text{compute}})$, we construct the PDA $A_{sim} = (Q', \Delta, \delta', q_0, F)$ as follows.

- $\Delta = \Sigma \cup \hat{\Gamma}$, (we assume $\Sigma \cap \hat{\Gamma} = \emptyset$).

- To construct Q' we use the mapping $f : \{\text{push}, \text{pop}\} \times \{\varepsilon, 0, 1\} \to (\Sigma \cup \hat{\Gamma})^*$ that is defined by

$$f(x, v) = \begin{cases} \varepsilon & \text{when } v = \varepsilon, \\ [\cdot x_v \cdot] & \text{otherwise.} \end{cases}$$

  Now, Q' is the union of Q and the set of states

$$\{\tau_S^{[q']} \mid q \xrightarrow{a, s_1/s_1', s_2/s_2'} q' \in \delta, S \text{ is a suffix of } f(\text{pop}, s_2) \cdot f(\text{push}, s_2')\} \ .$$

Similarly, δ' contains the rules

$$\{\tau_{a_1 \cdots a_n}^{[q]} \xrightarrow{a_1, \varepsilon/\varepsilon} \tau_{a_2 \cdots a_n}^{[q]} \mid \tau_{a_1 \cdots a_n}^{[q]} \in Q' \setminus Q\}$$

and $\{\tau_\varepsilon^{[q]} \xrightarrow{\varepsilon, \varepsilon/\varepsilon} q \mid q \in Q\}$. Also, for every transition $q \xrightarrow{a, s_1/s_1', s_2/s_2'} q'$ in δ,

- if $q, q' \in Q_{\text{input}}$ then $q \xrightarrow{a, s_1/s_1'} q'$ is in δ',
- if $q \in Q_{\text{input}}$ and $q \in Q_{\text{compute}}$ then $q \xrightarrow{\$, \varepsilon/\varepsilon} q'$ is in δ',
- if $q, q' \in Q_{\text{compute}}$ then $q \xrightarrow{\varepsilon, s_1/s_1'} \tau_{f(\text{pop}, s_2) \cdot f(\text{push}, s_2')}^{[q']}$ is in δ'. □

In the sequel, we will often be using strings of symbols to represent stack operation sequences. To simplify this, let us define the set VSR (for Valid Stack Runs) to contain all valid sequences of stack operations for a binary stack alphabet. Note especially that this includes the possibility of push operations with no corresponding pop, corresponding to sequences which end with a non-empty stack. Additionally, we define two functions to format these strings in convenient ways.

Definition 17 (Valid stack run). Define VSR, S_{VSR}, and $\bar{S}_{\text{VSR}}$ as follows.

- VSR $= \mathcal{L}(G)$ where G is the context-free grammar $G = (Q, \Sigma, \delta, q_0)$ with nonterminals $Q = \{q_0, b\}$, alphabet $\Sigma = \Gamma$ and δ containing the rules

$$\begin{array}{rcl} q_0 & \to & \text{push}_0\, q_0 \mid \text{push}_1 q_0 \mid q_0 q_0 \mid b \\ b & \to & \text{push}_0\, b\, \text{pop}_0 \mid \text{push}_1\, b\, \text{pop}_1 \mid bb \mid \varepsilon \end{array}$$

- $S_{\text{VSR}} : \text{VSR} \to \hat{\Gamma}^*$, such that we have $S_{\text{VSR}}(r_1 \cdots r_n) = [r_1] \cdots [r_n]$ for all $r_1 \cdots r_n \in \text{VSR}$. Also, let

$$\bar{S}_{\text{VSR}}(p_1 \cdots p_n) = \begin{cases} \varepsilon \text{ if } n = 0, \\ [\text{push}_1\, \text{pop}_0\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) \text{ if } p_1 = \text{push}_0, \\ [\text{push}_0\, \text{pop}_0\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) \text{ if } p_1 = \text{push}_1, \\ [\text{push}_0\, \text{push}_1\, \text{pop}_1] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) \text{ if } p_1 = \text{pop}_0, \\ [\text{push}_0\, \text{push}_1\, \text{pop}_0] \cdot \bar{S}_{\text{VSR}}(p_2 \cdots p_n) \text{ if } p_1 = \text{pop}_1. \end{cases}$$ □

Observation 3 (Stack runs). Note that for all $s \in$ VSR, all prefixes of s are also in VSR. Also, note that S_{VSR} and $\bar{S}_{\text{VSR}}$ complement each other in the sense that for all $p \in$ VSR we have

$$\underbrace{[[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]] \cdots [[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]]}_{|p| \text{ times}} \in S_{\text{VSR}}(p) \odot \bar{S}_{\text{VSR}}(p) \ .$$

□

Observations 3 completes the toolkit needed to establish a link between A_{sim} and the 2-PDA A.

Lemma 6. *Let $A = (Q, \Sigma, \delta, q_0, F)$ be a 2-PDA with input-partitioning and let $w \in \Sigma^*$ be any input string. Then the following holds.*

1. *A has an accepting run on w if and only if there exists some $p \in VSR$ such that $w \cdot \$ \cdot S_{VSR}(p) \in \mathcal{L}(A_{sim})$ where $A_{sim} = (Q', \Delta, \delta', q_0, F)$ is constructed as in Definition 16.*

2. *If A has an accepting run on w of length n, then there exists a $p \in VSR$ that fulfills the above with $|p| \leq 2n$.*

Proof. Consider the alphabets $\hat{\Gamma}$ and Γ from Definition 15. For all strings $s \in (\Sigma \cup \hat{\Gamma})^*$ we define $\gamma : (\Sigma \cup \hat{\Gamma})^* \to \Gamma^*$ so that $\gamma(s)$ produces s with all non-push/pop symbols removed (notably $\gamma(w \cdot \$ \cdot S_{\text{VSR}}(p)) = p$). Then define $\sigma : \text{VSR} \to \{0,1\}^*$ as the function which for all $s \in \Gamma^*$ produces the stack contents resulting from applying the stack operations in s, in order, to an initially empty stack. Notice that $\sigma(\gamma(s))$ is well-defined for all prefixes of $w \cdot \$ \cdot S_{\text{VSR}}(p)$.

Let $(Q_{\text{input}}, Q_{\text{compute}})$ be the input partitioning of A. Write configurations of A (when in a Q_{compute} states) as tuples of the form

$$(q, B_1, B_2) \in Q_{\text{compute}} \times \{0,1\}^* \times \{0,1\}^*$$

where q is the current state and B_1 and B_2 the current contents of Stack 1 and Stack 2, respectively. Write configurations of A_{sim} as tuples of the form

$$(q, B_1, B_2) \in Q_{\text{compute}} \times \{0,1\}^* \times \{0,1\}^*$$

where q is the current state, B_1 is the current stack contents, and $B_2 = \sigma(\gamma(s))$ where $s \in \Sigma^*$ is the part of the input string already read.

Base case Assume that A has a run on w. Let (q', B_1, ε) be the configuration of A after the unique transition $q \xrightarrow{\varepsilon,\varepsilon/\varepsilon,\varepsilon/\varepsilon} q'$ with $q \in Q_{\text{input}}$ and $q' \in Q_{\text{compute}}$ is taken (this unique transition must be taken at some point and Stack 2 will be empty by Definition 14). Since A_{sim} by construction contains all the same rules for the states in Q_{input} it will be able to, with $q \xrightarrow{\$,\varepsilon/\varepsilon} q'$ as the last transition, reach the configuration (q', B_1, ε) reading the string $w \cdot \$$. This establishes the base case, for all runs of A on the string a we will get to a configuration which A_{sim} can reach on the string $w \cdot \$$, and trivially $\gamma(w \cdot \$) \in$ VSR.

Inductive step Assume that A and A_{sim} are in configuration (q, B_1, B_2), let $s \in \Delta^*$ be the input already read by A_{sim}, and assume that $\gamma(s) \in \text{VSR}$. Let $s_1, s_2 \in \{0,1\}$ be the top elements of the stack contents B_1 and B_2 respectively. This means that A can take a transition $q \xrightarrow{w, s_1/s_1', s_2/s_2'} q'$. Let $w_1 \cdots w_m = f(\text{pop}, s_2) \cdot f(\text{push}, s_2')$ for f as in Definition 16. By construction there must exist transitions

$$q \xrightarrow{\varepsilon, s_1/s_1'} \tau^{[q']}_{w_1 \cdots w_n} \xrightarrow{w_1, \varepsilon/\varepsilon} \tau^{[q']}_{w_2 \cdots w_n} \xrightarrow{w_2, \varepsilon/\varepsilon} \ldots \xrightarrow{w_n, \varepsilon/\varepsilon} \tau^{[q']}_{\varepsilon} \xrightarrow{\varepsilon, \varepsilon/\varepsilon} q'$$

in A_{sim}. The first transition mimics the stack operations on Stack 1 in A, while the remainder makes the input read so far $s' = s \cdot w_1 \cdots w_n$ and $\gamma(w_1 \cdots w_n)$ will be exactly the stack operations performed on Stack 2 by s_2/s_2' in A. We already had $B_2 = \sigma(\gamma(s))$ so $\sigma(\gamma(s'))$ will match the resulting Stack 2 in A, making the configurations match again after the transitions. Since we know that the (possible) pop s_2 was matched in B_2 and we assumed that $\gamma(s) \in \text{VSR}$ we also know that $\gamma(s') \in \text{VSR}$.

Conclusion The other direction is easily shown in the same way, the string read by A_{sim} being mimicked by the operations on Stack 2 by A, and it is necessarily possible by virtue of the stack operation sequence being in VSR. This proves Part 1.

Part 2 follows trivially, the bound on the length of p follows directly from the induction, where p turns out to encode the sequence of stack operations performed on Stack 2 of A during the run. □

Next, we define the PDA A_{comp}, which will serve to read the "complement" of a valid stack run.

Definition 18 (A_{comp}). The language of the PDA A_{comp} is

$$\mathcal{L}(A_{comp}) = \{\$ \cdot \bar{S}_{\text{VSR}}(p) \mid p \in \text{VSR}\} \ .$$

It can be constructed from the context-free grammar $G = (Q, \Sigma, \delta, q_0)$ with nonterminals $Q = \{q_0, s, b\}$, and rules as follows.

$$\begin{array}{rcl} q_0 & \to & \$s \\ s & \to & [\text{push}_1\,\text{pop}_0\,\text{pop}_1]s \ \mid\ [\text{push}_0\,\text{pop}_1\,\text{pop}_2]s \ \mid ss \ \mid\ b \\ b & \to & [\text{push}_1\,\text{pop}_0\,\text{pop}_1]b[\text{push}_0\,\text{push}_1\,\text{pop}_1] \ \mid \\ & & [\text{push}_0\,\text{pop}_0\,\text{pop}_1]b[\text{push}_0\,\text{push}_1\,\text{pop}_0] \ \mid\ bb \ \mid \varepsilon \end{array}$$

□

The last definition of this section shows how to convert an input string for the 2-PDA A into a string to serve as input to the membership problem for the language $\mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$.

Definition 19 (Formatted input). For every string w over the alphabet Σ and every $n \in \mathbb{N}$ we define the function $\textsc{input} : \Sigma^* \times \mathbb{N} \to \Delta^*$, where $\Delta = \Sigma \cup \hat{\Gamma}$ (assume that $\Sigma \cap \hat{\Gamma} = \emptyset$), as

$$\textsc{input}(w) = w \cdot \$ \cdot \$ \underbrace{[[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]] \cdots [[\text{push}_0\,\text{push}_1\,\text{pop}_0\,\text{pop}_1]]}_{n \text{ times}} \ .$$

□

Lemma 7 establishes that A_{comp} will leave only encodings of valid stack runs as the suffix of strings produced by INPUT when shuffled with A_{sim}. In the statement, the PDA A_{sim} is obtained from the TM A using the construction in Definitions 16, and the PDA A_{comp} is as described in Definition 18.

Lemma 7. *Let $\Sigma \cap \Gamma = \emptyset$. Then, for every TM A, every string $w \in \Sigma^*$, and every $n \in \mathbb{N}$ it holds that $\textsc{input}(w, n) \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ if and only if there exists some $p \in VSR$ with $|p| = n$ such that $w \cdot \$ \cdot S_{VSR}(p) \in \mathcal{L}(A_{sim})$.*

Proof. Given $w \in \Sigma^*$ and $n \in \mathbb{N}$, the string produced by $\textsc{input}(w, n)$ is of the form $w \cdot \$ \cdot \$ \cdot [[\mathrm{push}_0\, \mathrm{push}_1\, \mathrm{pop}_0\, \mathrm{pop}_1]] \cdots [[\mathrm{push}_0\, \mathrm{push}_1\, \mathrm{pop}_0\, \mathrm{pop}_1]]$. Both A_{sim} and A_{comp} will, by construction, accept only strings with balanced, non-nested brackets. Additionally, we know that A_{comp} reads only strings of the form $\$ \cdot \bar{S}_{\mathrm{VSR}}(p)$ for some $p \in \mathrm{VSR}$. These facts alone forces A_{sim} to read a string of the form $w \cdot \$ \cdot [p_1] \cdots [p_n]$ for some $p_1, \ldots, p_n \in \Gamma$. As noted in Observation 3 $\bar{S}_{\mathrm{VSR}}$ and S_{VSR} behave as complements, so we will in fact have $p_1 \cdots p_n = p \in \mathrm{VSR}$. Thus, the string will be accepted if and only if the string $w \cdot \$ \cdot S_{\mathrm{VSR}}(p)$ is in $\mathcal{L}(A_{sim})$ □

Finally, the following theorem summarizes the main result of the section, bringing together the results of the previous lemmas.

Theorem 9. *For an input string w it is an NP-complete problem to decide whether or not $w \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$ when $\mathcal{L}(A_{sim})$ and $\mathcal{L}(A_{comp})$ are context-free languages, even when $\mathcal{L}(A_{sim})$ and $\mathcal{L}(A_{comp})$ are fixed. That is, the non-uniform membership problem for the shuffle of two context-free languages is NP-complete.*

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 $w \in w_1 \odot w_2$ and check if $w_1 \in \mathcal{L}(A_{sim})$ and $w_2 \in \mathcal{L}(A_{comp})$.

NP-hardness can now be shown using the tools we have established. Take any nondeterministic input-partitioned 2-PDA A that solves some NP-hard problem in polynomial time. Let $F : \mathbb{N} \to \mathbb{N}$ be a polynomial such that running A on an input string w takes less than $F(|w|)$ steps. These assumptions can be made since we can convert an arbitrary nondeterministic Turing machine into such a 2-PDA, and a nondeterministic TM can of course solve any problem in NP in polynomial time. Modify A so that it may loop indefinitely on all final states.

Construct A_{sim} from A using Definition 16, take A_{comp} as in definition 18. A string w is accepted by A if and only if $\textsc{input}(w, 2F(|w|)) \in \mathcal{L}(A_{sim}) \odot \mathcal{L}(A_{comp})$, by applying Lemma 7 to show that the part of the input left to A_{sim} is restricted to only valid stack runs, and then Lemma 6 to show equivalence with the 2-PDA. Notice also that since the run of A takes at most $F(|w|)$ steps we need at most $2F(|w|)$ stack symbol blocks in the INPUT construction, by Lemma 6. □

From this it also follows that parsing *permutation languages* is NP-complete. For a formal definition of permutation grammars we refer to [32].

Corollary 4. *The non-uniform membership problem for (order 2) permutation languages (defined in [33] with the notation L_{perm_2}, in [32] simply as L_{perm}) is NP-complete.*

Proof. All context-free grammars are trivially permutation grammars, and Theorem 7 in [32] gives a polynomial construction which for two permutation grammars G_1 and G_2 constructs a permutation grammar G' such that $\mathcal{L}(G') = \mathcal{L}(G_1) \odot \mathcal{L}(G_2)$. This establishes NP-hardness through Theorem 9.

The membership problem for a permutation language G is *in* NP since any string $w \in \mathcal{L}(G)$ can be derived in a polynomial number of derivation steps. This can be seen by considering the context-free subset of rules in G, which are applied only a polynomial number of times by the usual argument. Then simply notice that any reordering of an intermediary derivation string (of which there are a polynomial number) can be realized in less that n^2 steps using the interchange rules. □

5. 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 sanity-checked in polynomial time.

To be of practical use, at least the non-uniform membership problem needs to be efficiently decidable. This is known to be true for the shuffle languages, but our analysis shows that the efficiency depends heavily on the number of shuffle operations used. We also obtain that the non-uniform membership problem remains polynomial for the shuffle of a shuffle language and a context-free language. For the shuffle of two context-free languages, however, it is NP-complete.

Ideally, also the uniform membership problem should be solvable in polynomial time. The only language class we studied for which this is the case, unless P=NP, is the interleaving of a regular language and a context-free language.

Future work will strive to determine the complexity of the non-uniform membership problem for further restrictions of CFSA. If even very sparse use of shuffling has a large negative impact on the complexity, one could consider replacing the shuffle operator with weaker alternatives, such as unordered shuffle.

References

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

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

[3] 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

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

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

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

[7] 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

[8] 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

[9] 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

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

[11] 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

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

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

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

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

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

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

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

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

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

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

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

[23] 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

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

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

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

[27] Kari, L., Sosík, P.: Aspects of shuffle and deletion on trajectories. Theoretical Computer Science **332**(1-3) (2005) 47–61

[28] 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

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

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

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

[32] 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

[33] 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

[34] 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

[35] Schmidt, C., Sridharan, N., Goodson, J.: The plan recognition problem: An intersection of psychology and artificial intelligence. Artificial Intelligence **11**(1,2) (1978)

II

The Membership Problem for the Shuffle of Two Deterministic Linear Context-Free Languages is NP-complete

Martin Berglund

Department of Computing Science, Umeå University
90187 Umeå, Sweden
mbe@cs.umu.se

Technical Report UMINF 12.09

Abstract. Formal language models which employ shuffling, or interleaving, of strings are of interest in many areas of computer science. Notable examples include system verification, plan recognition, and natural language processing. Membership problems for the shuffle of languages are especially interesting. It is known that deciding membership for shuffles of regular languages can be done in polynomial time, and that deciding (non-uniform) membership in the shuffle of two deterministic context-free languages is NP-complete. In this paper we narrow the gap by showing that the non-uniform membership problem for the shuffle of two deterministic *linear* context-free languages is NP-complete.

1 Introduction

In this paper we look at a membership problem for a language model based on the shuffle operator, $\odot$, introduced in [GS65]. This operator takes two strings, w and w', and returns the set of all possible interleavings of these strings. For example, $ab \odot cd = \{abcd, acbd, acdb, cabd, cadb, cdab\}$. We generalise the operator to sets in the usual way, $L \odot L' = \{w \odot w' \mid w \in L, w' \in L'\}$. The specific membership problem we consider is the non-uniform membership problem for the shuffle of two deterministic linear context-free languages. Specifically, we show that there exist deterministic linear context-free languages L and L' such that it is NP-complete to decide whether a given input string w is in the set $L \odot L'$, even if L and L' are fixed.

For listings of previous work see for example [BBH11,MRS98], additionally [HZ80] is of special interest, since it draws parallels between two-stack Turing machines and the shuffle of context-free languages. One important omission in [BBH11] (coauthored by the author of this paper) is [ORR78], which shows that the non-uniform membership problem for the shuffle of two deterministic context-free languages is NP-complete. In [BBH11] we show this for general context-free languages, unaware of [ORR78]. In this paper, however, that proof is extended further.

2 Preliminaries

Let $[n] = \{1, \ldots, n\}$ for all $n \in \mathbb{N}$. The cardinality of a set S is denoted $|S|$. An ordered set is a finite set S where the elements have a predetermined order. For $i \in [|S|]$ let $S(i)$ denote the ith element. When the set is stated with numbered elements $S = \{s_1, \ldots, s_n\}$ it is implied that $s_i = S(i)$.

The Kleene closure of a set S is denoted S^*. An alphabet is a finite set of symbols, usually denoted Σ. For strings $w, w' \in \Sigma^*$ let $w \cdot w'$ denote the concatenation, for sets of strings $W, W' \subseteq \Sigma^*$ let $W \cdot W' = \{w \cdot w' \mid w \in W, w' \in W'\}$. We may write the singleton set $\{w\}$ as w for simplicity.

Let $\alpha_1, \ldots \alpha_n \in \Sigma$ and $n \in \mathbb{N}$ (indices such as n being in $\mathbb{N}$ is usually left implicit going forward) in the following. Let ϵ denote the empty string. Let $w^{\mathcal{R}}$ denote the reverse of a string w, that is $(\alpha_1 \cdots \alpha_n)^{\mathcal{R}} = \alpha_n \alpha_{n-1} \cdots \alpha_1$. As usual let $|\alpha_1 \cdots \alpha_n| = n$, and for all $s \in \Sigma$ let $|\alpha_1 \cdots \alpha_n|_s = |\{i \in [n] \mid \alpha_i = s\}|$.

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 (deterministic pushdown automata restricted to a single pushdown reversal for example), no formal definitions will be given here, see instead [HU90]. Instead of full pushdown automata implementations of the DLCF languages constructed (which would be large and hard to read) the strings in the languages are given in an inductive form from which the reader can easily construct automata themselves if desired.

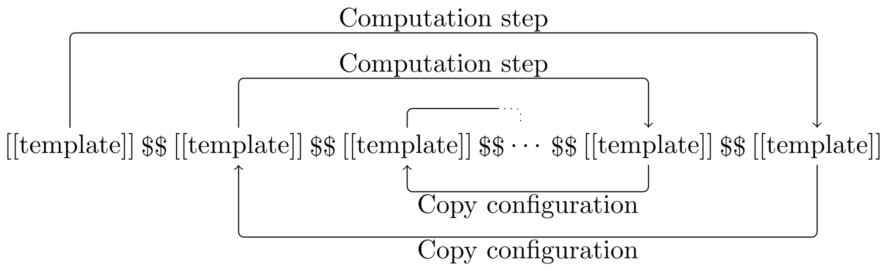
Non-deterministic polynomial time-bounded Turing machines are used heavily in the proofs to demonstrate NP-completeness. Full definitions of the machines are given, but for more complete background information on these topics see [GJ90,Min67].

3 Proof overview

The key building block necessary to make the shuffle of two DLCF languages perform a computation is making the languages communicate. 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], 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 then use this to choose one language make computation steps for a Turing machine, while the other language copies the configuration around to link the computation up. The following figure acts as a visual aid to see how the languages will cooperate to simulate the computation (beware however that many details are left out, the figure only serves as a structural overview).


Computation step
Computation step
[[template]] $$ [[template]] $$ [[template]] $$ · · · $$ [[template]] $$ [[template]]
Copy configuration
Copy configuration

4 Parsing the Shuffle of Deterministic Linear Context-Free Languages

The reduction hinges on representing the computations of Turing machines as strings. To facilitate this we will make a somewhat specialised definition of non-deterministic Turing machine configurations and runs.

Definition 1. *A* non-deterministic Turing machine (NTM) *is a tuple* (Q, Δ) *where*

- Q *is the finite ordered set of states,*
- $\Delta : Q \times \{0,1\} \times \{\leftarrow, \rightarrow\} \times Q \times \{0,1\}$ *is the finite set of rules.*

$Q(1)$ *is the initial state,* $Q(|Q|)$ *is the accepting state.*

The following alphabet has all the symbols needed to complete the reduction.

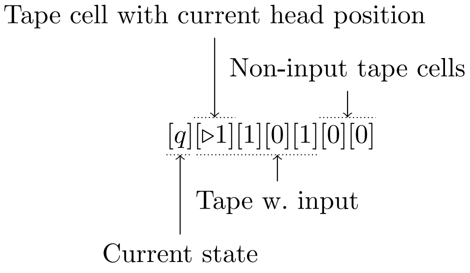
Definition 2. *Define* $\Sigma_M = Q \cup \Delta \cup \{0, 1, \triangleright, [,], \$, \#\}$.

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 3. *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 1. As will be seen in Definition 5 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 4. *For an NTM $M = (Q, \Delta)$ we may* apply rule $r \in \Delta$ *to a configuration* $c \in C_M$ *to produce the configuration* $c' \in C_M$ *under the following conditions. Let* $(q, \alpha, d, q', \alpha') = r$*, then 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 2. For example, in the configuration $[q][0][1][0][\triangleright 1][1][0]$ it is possible to apply the rule $(q, 1, \rightarrow, q', 0)$ to produce the configuration $[q'][0][1][0][0][\triangleright 1][0]$. In the configuration $[q][0][1][\triangleright 0]$ 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 $\triangleright$ is pointing to a 0 and the rule requires a 1.

Next follows the definitions of what it means for an NTM to accept a language in time bounded by some function. It should be obvious that this definition of a time-bounded non-deterministic Turing machine is equivalent to the usual one (see for example [Min67]). Most importantly this means that every problem $L \in$ NP (suitably encoded) is accepted by some NTM M in polynomially bounded time [GJ90].

Definition 5. *Take an NTM* $M = (Q, \Delta)$*, a function* $\psi : \mathbb{N} \rightarrow \mathbb{N}$ *and a string* $\alpha_1 \cdots \alpha_n \in \{0, 1\}^*$*. The* initial configuration *is defined as*

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

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* $c_i \rightarrow c_{i+1}$ *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.*

This definition differs slightly from the usual one 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 (the more usual) *at most* $\psi(n)$ steps can of course simply stay in the accepting state indefinitely to fulfil this condition.

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 6. *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*

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

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

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

Example 3. Let $M = (\{q_1, q_2\}, \Delta)$, and let $\psi(2) = 1$, then $S(M, \psi, 1)$ is

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

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 concept of a shuffle complement with respect to a template.

Definition 7. *For all strings $w, t \in \Sigma_M^*$, let complement(w, t) denote the* shuffle complement of w with respect to t, *defined as*

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

Example 4. *complement*$([q_1][0][\triangleright 1], [[q_1 q_2 q_3]][[\triangleright 01]][[\triangleright 01]]) = \{[q_2 q_3][\triangleright 1][0]\}$.

A very small but important lemma follows.

Lemma 1. *For any configuration $c \in C_M$ and configuration template T (as in Definition 6) if it holds that $|c|_{[} = \frac{1}{2}|T|_{[}$ then*

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

Proof (sketch). If we have a configuration template string T as in Definition 6 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 *complement*$(c, T) = \{c'\}$ 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 *complement*$(c', T) = \{c\}$. □

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

Definition 8. *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 5. Let $M = (\{q_1, q_2\}, \{(q_1, 0, \rightarrow, q_2, 1)\})$, then for example

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

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

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

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 easily 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 simply validates that these differences correspond to a rule in Δ.

Now we turn to the other DLCF language, which is responsible for linking up 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 $\{\bar{c}_i\} = complement(c, T)$ for some configuration c and configuration template T. Compare the constructed strings to those in Example 4.

Definition 9. *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 6. 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 1, 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 1. *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 $S(M,\psi,w) \in L_{step(M)} \odot L_{copy(M)}$.*

This proof is divided into two lemmas, the first showing the "only if" direction, the second the "if" direction.

Lemma 2. *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 $c_1, \dots, c_{\psi(n)+1} \in C_M$ 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 $\{\bar{c}_i\} = \mathit{complement}(c_i, T)$ where T is a configuration template as in Definition 6. 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:

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

w and w_{step} both start with c_1, so cancel that bit. Next w contains two dollar signs, one correponds 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 3. *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 take $w_{step} \in L_{step(M)}$ 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

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

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 1 that *complement*(c_i, T) and *complement*(d_i, T) are singletons for all $i \in \{2, \ldots, \psi(n)\}$. Equations 1 and 2 dictate that $e_i \in$ *complement*(c_i, T) and $e_i \in$ *complement*(d_i, T), which means that *complement*$(c_i, T) =$ *complement*$(d_i, T) = \{e_i\}$. Reversing this (again by Lemma 1) yields *complement*$(e_i, T) = \{c_i\} = \{d_i\}$, 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 $i \in [\psi(n)]$,
- $c_1 = I(M, \psi, \alpha_1 \cdots \alpha_n)$,
- *complement*$(e_{\psi(n)+1}, T) = \{c_{\psi(n)+1}\} \subset F(M)$ (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 $\alpha_1 \cdots \alpha_n$. □

Proof (of Theorem 1). Lemma 2 and Lemma 3 together show both directions of Theorem 1. □

It follows from Theorem 1 that the non-uniform membership problem for the shuffle of DLCF languages is NP-complete.

Corollary 1. *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 1. 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. □

5 Conclusions

Future work. The result in this paper narrows the gap between the cases where the membership problems for shuffled languages are intractable and where they are tractable. Still, there are several further restrictions that could be considered. The proof given here should be possible to modify in such a way that $L_{copy(M)}$ becomes a pure Dyck language, since the initial \$ symbol and the final configuration right after the \$#\$ midpoint marker are the only parts that disqualify it, but both of those could be handled by modifying the template string and changing $L_{step(M)}$. Similarly making both $L_{step(M)}$ and $L_{copy(M)}$ visibly pushdown [AM04] should be possible, since the structure of the construction is such that we know up-front which symbols will be pushed and which will be popped. Finding a language class larger than (or strictly different from) the regular languages for which membership in the shuffle is efficiently decidable remains an elusive but very interesting direction.

Acknowledgements. This paper would not have been possible without my shuffle collaborators Henrik and Johanna Björklund, and my advisor Frank Drewes.

References

[AM04] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[BBH11] Martin Berglund, Henrik Björklund, and Johanna Högberg. Recognizing shuffled languages. In *Language and Automata Theory and Applications*, volume 6638 of *Lecture Notes in Computer Science*, pages 142–154. Springer Berlin / Heidelberg, 2011.

[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.

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

[HU90] John E. Hopcroft and Jeffrey D. Ullman. *Introduction To Automata Theory, Languages, And Computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1990.

[HZ80] David Haussler and H. Paul Zeiger. Very special languages and representations of recursively enumerable languages via computation histories. *Information and Control*, 47(3):201 – 212, 1980.

[Min67] Marvin L. Minsky. *Computation: finite and infinite machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

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

[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.

III

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 $\mathit{pos} : T_\Sigma \to 2^{\mathbb{N}^*}$. For any $k \in \mathbb{N}$, including zero, $\alpha \in \Sigma$ and $t_1, \ldots, t_k \in T_\Sigma$ the definition of $\mathit{pos}\big(\alpha[t_1, \ldots, t_k]\big)$ is $\{\epsilon\} \cup \big\{(i, v_1, \ldots, v_n) \mid i \in \{1, \ldots, k\}, (v_1, \ldots, v_n) \in \mathit{pos}(t_i)\big\}$. That is, a position $p \in \mathit{pos}(\alpha[t_1, \ldots, t_n])$ 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 $\mathtt{swap}(M, W)$.*

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*

$$b \geq |W| + \sum_{i=1}^{n} \mathtt{swap}(M, W)_{i,i}.$$

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 $M' = \mathtt{swap}(M, W)$ 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$_{\text{ds}}$ (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

$$\left[\begin{array}{ccc|c} 0&6&6&1\\ 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&2\\ 0&6&6&1\\ 6&6&0&3\\ 6&0&6&4 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&2\\ 6&6&0&3\\ 0&6&6&1\\ 6&0&6&4 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4\\ 0&6&6&1 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&2\\ 6&0&6&4\\ 6&6&0&3\\ 0&6&6&1 \end{array}\right]$$

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

$$\left[\begin{array}{ccc|c} 0&6&6&1\\ 0&6&6&2\\ 6&6&0&3\\ 6&0&6&4 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&1\\ 6&6&0&3\\ 0&6&6&2\\ 6&0&6&4 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&1\\ 6&6&0&3\\ 6&0&6&4\\ 0&6&6&2 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 0&6&6&1\\ 6&0&6&4\\ 6&6&0&3\\ 0&6&6&2 \end{array}\right]$$

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 $(cbaac, abc, 1) \in \text{ESSCP}_{\text{ds}}$ 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 $(s_1 \cdots s_n, t_1 \cdots t_m, b) \in \text{ESSCP}_{\text{ds}}$. 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 $(M, b') \in \text{SAP}$ 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 $b' = 14$,

$$M = \begin{bmatrix} 2 & 3 & 3 \\ 9 & 4 & 12 \\ 1 & 2 & 8 \end{bmatrix} \Rightarrow \left[\begin{array}{cccccc} 2 & 16 & 16 & 2 & 16 & 2 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 0 & 2 & 16 & 8 & 16 \\ \hline 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{array}\right]$$

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

$$\left[\begin{array}{cccccc} 2 & 16 & 16 & 2 & 16 & 2 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 0 & 2 & 16 & 8 & 16 \\ \hline 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{cccccc} 16 & 0 & 2 & 16 & 8 & 16 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 2 & 16 & 16 & 2 & 16 & 2 \\ \hline 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{cccccc} 16 & 0 & 2 & 16 & 8 & 16 \\ 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 2 & 16 & 16 & 2 & 16 & 2 \\ 16 & 16 & 16 & 16 & 0 & 0 \end{array}\right] \Rightarrow \left[\begin{array}{cccccc} 0 & 0 & 16 & 16 & 16 & 16 \\ 16 & 0 & 2 & 16 & 8 & 16 \\ 16 & 8 & 4 & 16 & 12 & 16 \\ 16 & 16 & 0 & 0 & 16 & 16 \\ 16 & 16 & 16 & 16 & 0 & 0 \\ 2 & 16 & 16 & 2 & 16 & 2 \end{array}\right]$$

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.

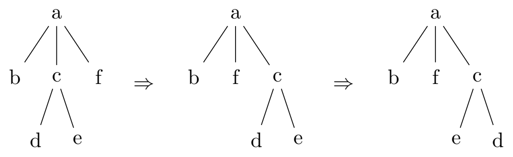


Figure 6: An example of applying the tree swaps $((2), (3, 1))$ to a small tree. That is, going from the first to second tree we swap the position 2, referring to the second child of the root, next the position $(3, 1)$ is swapped, referring to the first child of the rightmost child subtree of the root.

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'.

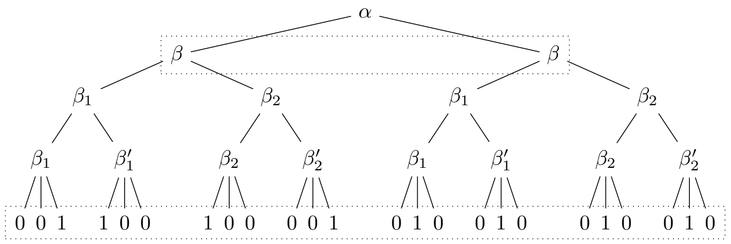


Figure 7: The tree t constructed in the reduction in Example 25. Notice that any solution only needs to perform swaps on the nodes in the dotted rectangles, all other nodes are already in their only possible internal order (compare to t' in Figure 8).

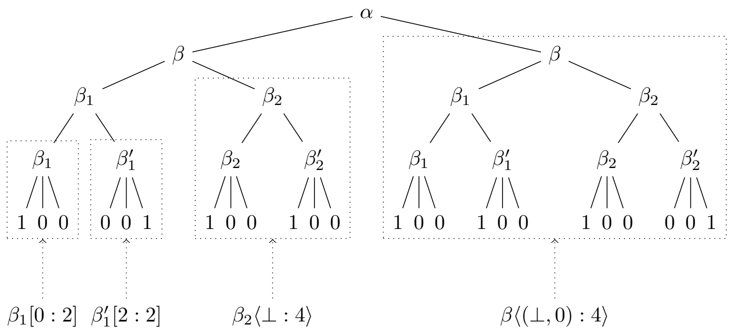


Figure 8: The tree t' constructed in the reduction in Example 25. The dotted arrows shows the notation we use to describe the indicated parts of the tree.

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* $(t, t', b) \in \text{TSwD}$ *if and only if* $(M, b) \in \text{SecAP}$.

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.

ISBN 978-91-7459-435-5
ISSN 0348-0542
UMINF 12.10