Pattern Matching in Trace Monoids[*]

Jochen Messner
Abt. Theoretische Informatik,
Universität Ulm,
89069 Ulm, Germany

February 13, 1997

Abstract

An algorithm is presented solving the factor problem in trace monoids. Given two traces represented by words, the algorithm determines in linear time whether the first trace is a factor of the second one. The space used for this task is linear in the length of the first word. Similar to the Knuth-Morris-Pratt Algorithm for the factor problem on words, the algorithm simulates a finite automaton determined by the first word on the second word. To develop the algorithm, we examine overlaps of two traces, and show that they form a lattice. Finally we investigate the lattice of extensible trace pairs (which represent still extensible prefixes of a searched factor appearing in some other trace), because of their close relations to the structures used by the algorithm.

1 Introduction

The pattern matching problem in free monoids is an extensively studied problem in computer science. For two words $v, x \in A^*$ it is asked whether there are words $u, w \in A^*$ such that $x = uvw$, i.e., it is asked whether v is a factor of x. There are several linear time algorithms solving the problem. The algorithm which was given by Knuth, Morris, and Pratt [14] has close connections to the theory of finite automata (see [1]): First from the first word v a so called failure function is computed as a table in time linear to the length of v; after this first stage the failure function is used to simulate on the second word in linear time a finite automaton accepting the language

[*]An extended abstract of this paper appears in [17].

$A^* \cdot v \cdot A^* = \{uvw \mid u, w \in A^*\}$. Altogether the time used is linear in the input size, and even does not depend on the size of the alphabet, because the automaton is not constructed explicitly.

In this paper we use a quite similar approach to the factor problem in trace monoids. Trace monoids, also called free partially commutative monoids, have been studied in combinatorics in [3]. In [16] Mazurkiewicz considered them as a suitable mathematical model for concurrent systems. Given a finite set of actions (an alphabet), some of the actions are considered independent (e.g., they may use different resources). Because the order of independent actions is irrelevant, one identifies sequences of actions (i.e., words) which can be made equivalent by exchanging adjacent independent actions. This yields an equivalence relation $\sim_I$ on those words which is, in fact, a congruence; a congruence class is called trace, consequently the monoid of the congruence classes is called trace monoid. It is determined uniquely by the generating alphabet and the relation I of independent letters. The free monoid is obtained as a special case, when all letters are dependent. Trace monoids have been studied in many publications. Good starting points are [8], [9], and [7].The factor problem in a trace monoid M is to decide for two words, whether the trace l, represented by the first word, is a factor of the trace t, represented by the second word (where l is called factor of t, when $t = pls$ for some traces p, s). A linear-time algorithm for the problem using space linear in the input size was given in [15]. This space-complexity is not always desirable. So, for example, a control component of a concurrent system may need to recognize certain subsequences in a sequence of actions (modulo independence) without remembering all executed actions. Therefore we consider in this paper an approach more closely related to finite automata. From the first word v, in a first stage, several structures are computed in linear time. These structures are used in the second stage to simulate on the second word a finite automaton recognizing the language $\{x \in A^* \mid x \sim_I uvw$ for some words $u, w\}$ which is the set of words representing the traces in $M \cdot l \cdot M = \{pls \mid p, s \in M\}$. The time used for this simulation is linear in the length of the second word such that altogether the algorithm needs linear time. Because of the similarity we consider the presented algorithm as a generalization of the Knuth-Morris-Pratt Algorithm to trace monoids, although there is in general no correspondent to the failure function for traces (cf. [18] for related results). Already in [13] Hashiguchi and Yamada had this approach. However, the algorithm they proposed to solve the factor problem produces an incorrect answer in some cases as we show by an example. The observation of this error was the reason for the investigations presented in this paper. And in fact, one may see our results as a correction of the results of Hashiguchi and Yamada.

The organization of the paper is as follows: We first introduce basic notions. A representation of traces is obtained by the general embedding theorem using projections. In Section 3 we study the set of prefixes and suffixes of some trace. Both sets form lattices with the prefix (resp. suffix) orders. We observe that projections are morphisms for those lattices which allows us to deduce easily that the intersection of a set of prefixes with a set of suffixes (called overlaps of two traces) still forms a lattice. Using these results, we finally develop a finite automaton recognizing the language of traces containing a given trace l as a suffix. In Section 4 we present an algorithm computing the transition function of this automaton which allows us to simulate the automaton in linear time. This simulation solves the suffix problem (the algorithm was already given in [13], we obtain an improved time-complexity). In Section 5 we obtain a finite automaton recognizing the traces containing a given trace l as a factor. A linear-time simulation of this automaton solves the factor problem. However, for some trace monoids the simulation presented needs time and space exponential in the alphabet-size. In Section 6 we investigate extensible trace pairs which have close relations to the states reached in the automaton. We show that they form a lattice—an observation which may lead to an improvement of the presented algorithm remaining efficient even when the alphabet is a part of the input.

Although the main purpose of this paper is the investigation of the factor problem, the results obtained for overlaps (cf. [18]) and extensible trace pairs (cf. [13]) are interesting on their own.

2 Preliminaries

In the following, the basic notions for trace monoids are given. A suitable representation of traces is obtained by the embedding theorem, which allows us to represent a trace uniquely by a tuple of words. We also give some notations on finite automata and a very brief description of some properties of the pattern-matching algorithm of Knuth, Morris, and Pratt. As we use standard notions of [2], we give no introduction to lattice or poset theory, here.

2.1 Free Partially Commutative Monoids

Let denote A a finite alphabet, A^* the free monoid generated by A, and λ the empty word of A^*. $|A|$ denotes the number of letters in A. For any word $w \in A^*$, $|w|$ denotes its length. The alphabet of a word w is the set $\mathrm{alph}(w) = \{a \in A \mid w = uav\}$ of letters actually appearing in w.

By $D \subseteq A \times A$ we denote throughout this paper a reflexive and symmetric dependence relation. Its complement $I = (A \times A) - D$ is called independence or commutation relation. The graph (A,D) is called dependence alphabet. Let $\sim_I$ be the congruence over A^* which is generated by the equivalences $ab \sim_I ba$ for $(a,b) \in I$. The monoid of the congruence classes is the free partially commutative monoid $\mathrm{M}(A,D) = A^*/\sim_I$. Following Mazurkiewicz [16] a congruence class $t \in \mathrm{M}(A,D)$ is called a trace, consequently $\mathrm{M}(A,D)$ is also called trace monoid. The neutral element, the class of the empty word, is called empty trace and denoted by λ, too; a letter $a \in A$ may denote the trace $[a]_{\sim_I}$. The notions of the length and the alphabet of a word can be transfered to traces, as they are invariant for words representing the same trace. Traces t_1, t_2 are said to be independent when $\mathrm{alph}(t_1) \times \mathrm{alph}(t_2) \subseteq I$; in this case $t_1t_2 = t_2t_1$. For a subset $B \subseteq A$, $D(B) = \{a \in A \mid (a,b) \in D$ for some $b \in B\}$ denotes the set of letters dependent from B; for $a \in A$ we just write $D(a)$ instead of $D(\{a\})$.

A trace $l \in \mathrm{M}(A,D)$ is called factor of $t \in \mathrm{M}(A,D)$, when $t = pls$ for some traces $p, s \in \mathrm{M}(A,D)$; l is called prefix (suffix) of t when p (resp. s) can be chosen to be λ. The set of prefixes (suffixes) of t is denoted by $\mathrm{Pre}(t)$ (resp. $\mathrm{Suf}(t)$). Levi's Lemma on factorizations of words has the following generalization for traces. For a proof of Proposition 2.1 see [7, Proposition 1.3.1] (see also [6]). Note as a consequence that trace monoids are cancellative.

Proposition 2.1 *Let $p, s, p', s' \in M(A,D)$ with $ps = p's'$. Then there are uniquely determined traces $x, y, y', z \in M(A,D)$ with $alph(y) \times alph(y') \subseteq I$ and $p = xy$, $p' = xy'$, $s = y'z$ and $s' = yz$.*

A graph $H = (A',D')$ is called subgraph of (A,D), denoted by $H \subseteq (A,D)$, when $A' \subseteq A$, and $D' \subseteq D$ is a reflexive and symmetric relation. Intersection and union of subgraphs is defined on the components. A family $\mathcal{G}$ of subgraphs of (A,D) is called covering, when $(A,D) = \bigcup_{G \in \mathcal{G}} G$. A subset S of $\mathcal{P}(A)$, where $\mathcal{P}(A)$ is the set of all subsets of $\mathcal{A}$, may denote the family $\{(B, D_{|B}) \mid B \in S\}$ and is called covering accordingly. Cliques are subgraphs of the form $(C, C \times C) = (C, D_{|C})$ for some $C \subseteq A$, they are simply represented by the set $C \subseteq A$. A covering consisting only of cliques, is called clique-covering. The set $\{\{a,b\} \mid (a,b) \in D\}$ is a trivial clique-covering of (A,D).

For a subgraph $H = (A',D')$ of (A,D), π_H denotes the projection of $\mathrm{M}(A,D)$ onto $\mathrm{M}(A',D')$, i.e., $\pi_H : \mathrm{M}(A,D) \to \mathrm{M}(A',D')$ is an homomorphism such that for $a \in A$, $\pi_H(a) = a$ if $a \in A'$, and $\pi_H(a) = \lambda$ else. For $B \subseteq A$, we use π_B to denote $\pi_{(B,D_{|B})}$, where $D_{|B} = D \cap (B \times B)$. The a-length of a trace

$t \in \mathrm{M}(A{,}D)$ is given by the length of its $\{a\}$-projection, $|t|_a = |\pi_{\{a\}}(t)|$, for $a \in A$,

Proposition 2.2 is a generalization of [6, Proposition 1.1] given in [7], where it was called general embedding theorem. It allows us to represent a trace $t \in \mathrm{M}(A{,}D)$ uniquely by a tuple of traces containing fewer letters. $\prod_i M_i$ denotes the direct product of the monoids M_i. An element of $\prod_i M_i$ is uniquely denoted by $(t_i)_i$ with $t_i \in M_i$; the multiplication in $\prod_i M_i$ is defined on the components: $(t_i)_i \cdot (s_i)_i = (t_i \cdot s_i)_i$.

Proposition 2.2 *Let $\mathcal{G}$ be family of subgraphs of $(A{,}D)$. $\mathcal{G}$ is a covering if, and only if, the mapping $\pi : M(A{,}D) \to \prod_{G\in\mathcal{G}} M(G)$ defined by $\pi(t) = (\pi_G(t))_{G\in\mathcal{G}}$ is an embedding (i.e., an injective homomorphism).*

For a covering $\mathcal{G}$ of $(A{,}D)$ we call the tuple $(\pi_G(t))_{G\in\mathcal{G}}$ the tuple-representative of $t \in \mathrm{M}(A{,}D)$. Extending terminology of [4], a tuple which is a tuple-representative of some trace is called reconstructible. Choosing $\mathcal{G}$ to be a clique-covering, we obtain a version of Proposition 2.2 already given in [10] which allows us to represent a trace uniquely by a tuple of words. Clearly, this tuple is computable in time linear to its size, given a word representing the trace (as the model of computation we generally assume a RAM (see [19]) with a uniform cost criteria, i.e., space is determined by the number of used registers and time is the number of operations executed. This is a realistic assumption, as in all algorithms of this paper, the number stored in any register is not bigger than the number of registers used by the algorithm. For a fixed alphabet A the presented algorithms can even be implemented on a multi-tape Turing machine with the stated time- and space-bounds).

2.2 Automata

A (nondeterministic) finite automaton, shortly called automaton, is a tuple $\mathcal{A} = (Q, A, \delta, q_0, F)$ consisting of the finite state-set Q, the alphabet A, the transition relation $\delta \subseteq Q \times A \times Q$, the initial state $q_0 \in Q$, and the set of final states $F \subseteq Q$. $\mathcal{A}$ can be seen as an edge-labeled graph with vertices Q, where $(p, a, q) \in \delta$ is an edge from p to q labeled a. For $w \in A^*$ we write $p \xrightarrow{w}_{\mathcal{A}} q$, when in $\mathcal{A}$ there is a path from p to q labeled w ($w = \lambda$ implies $p = q$). A language $L \subseteq A^*$ is said to be recognized by $\mathcal{A}$, when $L = \bigcup_{q\in F}\{w \in A^* \mid q_0 \xrightarrow{w}_{\mathcal{A}} q\}$.

An automaton $\mathcal{A}$ is called $\mathrm{M}(A{,}D)$-automaton, when $p \xrightarrow{w}_{\mathcal{A}} q$ implies $p \xrightarrow{v}_{\mathcal{A}} q$ for any $v \sim_I w$. In this case we simply write $p \xrightarrow{t}_{\mathcal{A}} q$, where $t = [w]_{\sim_I}$ is the trace represented by w. $L = \bigcup_{q\in F}\{t \in \mathrm{M}(A{,}D) \mid q_0 \xrightarrow{t}_{\mathcal{A}} q\}$ is the trace language recognized by the $\mathrm{M}(A{,}D)$-automaton $\mathcal{A}$. A trace language $L \subseteq \mathrm{M}(A{,}D)$ is called recognizable, when L is recognized by some finite

M(A,D)-automaton. For languages $L_1, L_2 \subseteq$ M(A,D), the concatenation $L_1 \cdot L_2 = \{l_1 l_2 \mid l_1 \in L_1 \text{ and } l_2 \in L_2\}$ is defined on the elements. We also write $L \cdot l$ for $L \cdot \{l\}$. It is known that the concatenation of recognizable trace-languages is constructively recognizable (see Theorem 5.1).

See [11] for a definition of deterministic, complete, and minimal deterministic automata.

2.3 The Algorithm of Knuth-Morris-Pratt

By the Algorithm of Knuth-Morris-Pratt (see [14], [1]) it is decidable in linear time, whether a word $v \in A^*$ is a factor of the word $w \in A^*$ using $\mathcal{O}(|v|)$ space. The basis of the algorithm is the so called failure function $\phi_v : \text{Pre}(v) - \{\lambda\} \to \text{Pre}(v)$, where $\phi_v(p)$ is the longest word $s \neq p$ which is both, a prefix and a suffix of p. The failure function ϕ_v can be calculated as a table in time linear to $|v|$ (notice, a prefix p of v may be uniquely represented by its length $|p|$).

The complete, deterministic automaton $\mathcal{A}_v = (\text{Pre}(v), A, \varphi_v, \lambda, v)$, where for $p \in \text{Pre}(v)$, $a \in A$, $\varphi_v(p, a)$ is the longest word in $\text{Pre}(v) \cap \text{Suf}(pa)$, is the minimal automaton recognizing the language $A^* \cdot v$. Using the failure function ϕ_v, the transition function φ_v is computable efficiently by the following relationship

$$\varphi_v(p,a) \quad = \quad \begin{cases} pa & \text{if } pa \in \text{Pre}(v), \\ \lambda & \text{if } p = \lambda \text{ and } a \notin \text{Pre}(v), \\ \varphi_v(\phi_v(p), a) & \text{else.} \end{cases}$$

Using this relations the computation of $\varphi_v(p, a)$ needs, for any prefix p of v, and any $a \in A$, time linear to the number of times the failure function is used in the computation. Because each use of ϕ_v shortens the input to φ_v, the time used is linearly bounded by $|pa| - |\varphi_v(p, a)|$.

3 Prefixes and Suffixes

We examine now the set of prefixes and the set of suffixes of some trace. For $l, t \in$ M(A,D), we write $l \leq_p t$ ($l \leq_s t$), when l is a prefix (resp. suffix) of t. It is clear that $\leq_s$ and $\leq_p$ are partial orders. See [12] for an investigation of the poset (M(A,D), $\leq_p$).

3.1 Prefix and Suffix Lattices

Lemma 3.1 shows that the structures $(\text{Pre}(l), \leq_p)$ and $(\text{Suf}(l), \leq_s)$ are lattices for any trace l. The statement for the prefix-case can be found in [5], the result for suffixes is obtained by symmetry.

Lemma 3.1 *Let $t, t' \in M(A,D)$ both be prefixes (suffixes) of the same trace. Then there are uniquely determined traces $x, y, y' \in M(A,D)$ with alph$(y) \times$ alph$(y') \subseteq I$, $t = xy$, and $t' = xy'$ (resp. $t = yx$, and $t' = y'x$). Further, x is the greatest lower bound for t and t' in the prefix (resp. suffix) order in $M(A,D)$ and xyy' (resp. $yy'x$) is the least upper bound for t and t' in the prefix (resp. suffix) order.*

In the following the least upper bound of t and t' in the prefix (suffix) order is denoted by $t \sqcup_p t'$ (respectively $t \sqcup_s t'$), the greatest lower bound by $t \sqcap_p t'$ (respectively $t \sqcap_s t'$). We say that $t \sqcup_p t'$ is not defined, when there is no trace s such that $t \leq_p s$ and $t' \leq_p s$ (similar for $\sqcup_s$).

Because projections are monoid homomorphisms, they are poset morphisms with respect to the prefix and the suffix order. Also independence of traces is preserved by projections. We obtain that projections are also morphisms for the lattices of prefixes (resp. suffixes):

Lemma 3.2 *Let $t, t' \in M(A,D)$ such that $t \sqcup_p t'$ (resp. $t \sqcup_s t'$) is defined. Let $G \subseteq (A,D)$. Then $\pi_G(t \sqcup_p t') = \pi_G(t) \sqcup_p \pi_G(t')$, and $\pi_G(t \sqcap_p t') = \pi_G(t) \sqcap_p \pi_G(t')$ (respectively, $\pi_G(t \sqcup_s t') = \pi_G(t) \sqcup_s \pi_G(t')$ and $\pi_G(t \sqcap_s t') = \pi_G(t) \sqcap_s \pi_G(t')$).*

Proof. Let $x, y, y' \in \mathrm{M}(A,D)$ be such that $t = xy$, $t' = xy'$ and alph$(y) \times$ alph$(y') \subseteq I$ (by Lemma 3.1 these values exist if $t \sqcup_p t'$ is defined). Let $G = (A',D') \subseteq (A,D)$, and $I' = (A' \times A') - D'$. Because π_G is an homomorphism, we have $\pi_G(t) = \pi_G(x)\pi_G(y)$, and $\pi_G(t') = \pi_G(x)\pi_G(y')$. Observe $\pi_G(y) \times \pi_G(y') \subseteq I_{|A'} \subseteq I'$. Using Lemma 3.1 in the monoid $M(G)$, we obtain that $\pi_G(x)$ is the greatest lower bound, and $\pi_G(x)\pi_G(y)\pi_G(y')$ is the least upper bound of $\pi_G(t)$ and $\pi_G(t')$ in the prefix order. Notice $t \sqcup_p t' = xyy'$ and $t \sqcap_p t' = x$. Thus, $\pi_G(t) \sqcup_p \pi_G(t') = \pi_G(t \sqcup_p t')$, and $\pi_G(t) \sqcap_p \pi_G(t') = \pi_G(t \sqcap_p t')$. The result for the suffix case is obtained analogously. □

Lemma 3.2 allows us, in combination with Proposition 2.2, to transfer results for the prefix and suffix orders in free monoids to trace monoids. See the proof of Lemma 3.3 as an example. In a similar way, one could, for example, show that $(\mathrm{Pre}(l), \sqcup_p, \sqcap_p)$ is a distributive lattice which is a well known fact (see e.g. [18]).

3.2 The Lattice of Overlaps

The traces in $\mathrm{Pre}(l) \cap \mathrm{Suf}(t)$ are called overlaps of l and t. We show that the prefix and suffix order coincides for the overlaps of two traces and that there is a unique maximal overlap denoted by $\sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(t))$. The notion of overlap used here is a slight generalization of the same notion in [18]. In [18] sets $\mathrm{Pre}(x) \cap \mathrm{Suf}(x)$ for some trace x were examined and it was shown that such a set forms a lattice. This lattice is equal to $(\mathrm{Pre}(l) \cap \mathrm{Suf}(t), \leq_p)$ when choosing $x = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(t))$. Compared to [18] we obtain a very short proof, due to Lemma 3.2.

Observe for words $u, v \in A^*$ that $u \sqcup_p v$ (resp. $u \sqcup_s v$) is just the longest word of both (if defined). Thus, if $u \sqcup_p v$ and $u \sqcup_s v$ both exist, $u \sqcup_p v = u \sqcup_s v$, and $u \sqcap_p v = u \sqcap_s v$. We get the same result for traces.

Lemma 3.3 *Let $l, t \in M(A,D)$, let $r, s \in Pre(l) \cap Suf(t)$. Then $r \sqcup_p s = r \sqcup_s s$ and $r \sqcap_p s = r \sqcap_s s$.*

Proof. Let $\mathcal{C}$ be a clique-covering of (A,D) and let $\pi : \mathrm{M}(A,D) \to \prod_{C \in \mathcal{C}} C^*$ be an embedding like in Proposition 2.2. We have by Lemma 3.2

$$\begin{aligned} \pi(r \sqcup_p s) &= (\pi_C(r \sqcup_p s))_{C \in \mathcal{C}} = (\pi_C(r) \sqcup_p \pi_C(s))_{C \in \mathcal{C}}, \quad \text{and} \\ \pi(r \sqcup_s s) &= (\pi_C(r \sqcup_s s))_{C \in \mathcal{C}} = (\pi_C(r) \sqcup_s \pi_C(s))_{C \in \mathcal{C}}. \end{aligned}$$

Note that for each clique $C \in \mathcal{C}$ both suprema $\pi_C(r) \sqcup_p \pi_C(s)$ and $\pi_C(r) \sqcup_s \pi_C(s)$ are defined and thus equal, as $\pi_C(r), \pi_C(s) \in \mathrm{Pre}(\pi_C(l)) \cap \mathrm{Suf}(\pi_C(t))$ is implied by the assumption $r, s \in \mathrm{Pre}(l) \cap \mathrm{Suf}(t)$. We obtain

$$\pi(r \sqcup_p s) = \pi(r \sqcup_s s).$$

Because π is injective, this yields the equality of $r \sqcup_p s$ and $r \sqcap_s s$. To obtain a proof for the infimum case, replace $\sqcup$ by $\sqcap$ in the formulas above. $\square$

Due to Lemma 3.3, we may just write $\sqcup$ for the suffix (resp. prefix) supremum in sets $\mathrm{Pre}(l) \cap \mathrm{Suf}(t)$. We say that x is an overlap of y, and write $x \leq_o y$, when $x \leq_p y$ and $x \leq_s y$. It is clear that $(\mathrm{M}(A,D), \leq_o)$ is a poset. As a corollary of Lemma 3.3 we obtain

Theorem 3.4 *Let $l, t \in M(A,D)$. Then*

$$(Pre(l) \cap Suf(t), \leq_p) = (Pre(l) \cap Suf(t), \leq_s) = (Pre(l) \cap Suf(t), \leq_o),$$

and the poset of overlaps $(Pre(l) \cap Suf(t), \leq_o)$ is a lattice.

It is now obvious that $\mathrm{Pre}(l) \cap \mathrm{Suf}(t) = \mathrm{Pre}(l) \cap \mathrm{Suf}(p) = \mathrm{Pre}(p) \cap \mathrm{Suf}(p)$ when $p = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(t))$. So each overlap of l and t is an overlap of p, and vice versa.

3.3 An Automaton Recognizing $\mathrm{M}(A,D) \cdot l$

We present now a minimal deterministic automaton recognizing the language $\mathrm{M}(A,D) \cdot l = \{t \in \mathrm{M}(A,D) \mid l \leq_s t\}$. Clearly, l is a suffix of t if, and only if, $l = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(t))$. Examine the extension of t by a letter a:

Lemma 3.5 *Let $a \in \Sigma$, $t, l \in M(A,D)$ and $p = \sqcup(Pre(l) \cap Suf(t))$. Then*

$$\sqcup(Pre(l) \cap Suf(ta)) \;=\; \sqcup(Pre(l) \cap Suf(pa)).$$

Proof. We show $\mathrm{Pre}(l) \cap \mathrm{Suf}(ta) = \mathrm{Pre}(l) \cap \mathrm{Suf}(pa)$. First let $r \in \mathrm{Pre}(l) \cap \mathrm{Suf}(ta)$. Either $r = r'a$ with $r' \in \mathrm{Pre}(l) \cap \mathrm{Suf}(t)$, or $r \in \mathrm{Pre}(l) \cap \mathrm{Suf}(t)$ and $\mathrm{alph}(r) \times \{a\} \subseteq I$. In the first case, if $r = r'a$, r' is an overlap of p, therefore $r = r'a \in \mathrm{Suf}(pa) \cap \mathrm{Pre}(l)$. In the second case, r is an overlap of p. Clearly, r is also suffix of pa as r is independent from a. This implies $r \in \mathrm{Suf}(pa) \cap \mathrm{Pre}(l)$. Hence, the inclusion $\mathrm{Pre}(l) \cap \mathrm{Suf}(ta) \subseteq \mathrm{Pre}(l) \cap \mathrm{Suf}(pa)$ results. The opposite inclusion $\mathrm{Pre}(l) \cap \mathrm{Suf}(ta) \supseteq \mathrm{Pre}(l) \cap \mathrm{Suf}(pa)$ is obvious, since $p \leq_s t$. $\square$

By induction we obtain:

Theorem 3.6 *Let $l \in M(A,D)$. Let $\Phi_l : Pre(l) \times A \to Pre(l)$ be defined by*

$$\Phi_l(p, a) \;=\; \sqcup(Pre(l) \cap Suf(pa)).$$

Then $\mathcal{A}_l = (Pre(l), A, \Phi_l, \lambda, \{l\})$ is a minimal $M(A,D)$-automaton recognizing $M(A,D) \cdot l$.

For $p, p' \in Pre(l)$ it holds $p \xrightarrow{t}_{\mathcal{A}_l} p'$ if, and only if, $p' = \sqcup(Pre(l) \cap Suf(pt))$.

Proof. We first show by induction on $|w|$ that $p \xrightarrow{w} p'$ implies $p' = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(pt))$, where t is the trace represented by w. For $w = \lambda$ this is obvious. Now let $w' = wa$, $a \in A$, $w \in A^*$ representing the trace $t \in \mathrm{M}(A,D)$, and $t' = ta$ be the trace represented by w'. We have $p \xrightarrow{w'} p'$ if, and only if, $p \xrightarrow{w} q \xrightarrow{a} p'$ for some $q \in \mathrm{Pre}(l)$. By the inductive hypothesis, $q = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(pt))$. By the definition of Φ_l, $p' = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(qa)$. Using Lemma 3.5 we obtain $p' = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(pta))$.

Because $\mathcal{A}_l$ is complete this implies that $\mathcal{A}_l$ is an $\mathrm{M}(A,D)$-automaton. Notice also that a trace t is accepted by $\mathcal{A}_l$ if, and only if, $l = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(\lambda t))$, which is the case if, and only if, l is a suffix of t. It remains to show that $\mathcal{A}_l$ is a minimal automaton. Let $p, q \in \mathrm{Pre}(l)$, we show that $p = q$ if $p \xrightarrow{r} l$ and $q \xrightarrow{r} l$ for all $r \in \mathrm{M}(A,D)$. Let $r, s \in \mathrm{M}(A,D)$ be such that $l = pr = qs$. It is clear that $p \xrightarrow{r} l$ and $q \xrightarrow{s} l$. Assume additionally $q \xrightarrow{r} l$ and $p \xrightarrow{s} l$. This is the case only if l is a suffix of qr and also a suffix of ps. Thus qs is a suffix of qr and pr is a suffix of ps which implies $r = s$

(to see this, let $\mathcal{C}$ be a clique-covering. Now for any clique $C \in \mathcal{C}$, the word $\pi_C(q)\pi_C(s)$ is a suffix of the word $\pi_C(q)\pi_C(r)$, and the word $\pi_C(p)\pi_C(r)$ is a suffix of the word $\pi_C(p)\pi_C(s)$ which implies $\pi_C(s) = \pi_C(r)$). We obtain $p = q$. $\square$

Let in the following Φ_l and $\mathcal{A}_l$ refer to the notions given in the theorem.

4 An Algorithm for the Suffix Problem

In this section we present an algorithm computing the transition function of $\mathcal{A}_l$. On input $p \in \mathrm{Pre}(l)$ and $a \in A$ the algorithm outputs $\Phi_l(p, a)$ in time linear to $|pa| - |\Phi_l(p, a)|$. This time complexity yields linear time complexity when simulating the automaton $\mathcal{A}_l$ on some input $x = a_1 \dots a_n$. Therefore one obtains a linear-time algorithm for the suffix problem in $\mathrm{M}(A,D)$ which is to decide on input of two words $v, x \in A^*$, whether the trace l represented by v is a suffix of the trace t represented by x. Implicitly, the computation of Φ_l was already given in [13, Algorithm 5.2]. We improve the time-complexity of this algorithm from $\mathcal{O}(|A|^3 \cdot |vx|)$ ([13, Theorem 5.1]) to $\mathcal{O}(|A| \cdot |vx|)$ in Theorem 5.8.

A state $p \in \mathrm{Pre}(l)$ will be uniquely represented by its tuple-representative. We give some preliminary results on this representation. Let $\mathcal{C}$ be a covering of (A,D) (not necessarily a clique-covering). Extending terminology of [4] a tuple $(u_C)_{C\in\mathcal{C}}$ is said to be quasi-reconstructible, when $|u_C|_a = |u_{C'}|_a$ for any $C, C' \in \mathcal{C}$, $a \in C \cap C'$. Where, for a subgraph $H = (A',D')$ of (A,D), we write $a \in H$ when $a \in A'$. Proposition 4.1 is a slight generalization of [10, Proposition 1.6 (ii)].

Proposition 4.1 *Let $\mathcal{C}$ be a covering of (A,D), $t \in M(A,D)$. A tuple $(u_C)_{C\in\mathcal{C}} \in \prod_{C\in\mathcal{C}} M(C)$ represents a prefix (suffix) of t if, and only if,*

(i) it is quasi-reconstructible and

(ii) u_C is a prefix (resp. suffix) of $\pi_C(t)$ for all $C \in \mathcal{C}$.

Proof. We show the statement for the prefix case. A proof for the suffix case is obtained by symmetry. The only-if part is clear, as $(\pi_C(p))_{C\in\mathcal{C}}$ is reconstructible for each trace p, and $\pi_C(p) \leq_p \pi_C(l)$ is implied by $p \leq_p l$. The proof for the if-part is by induction on $|t|$. For $t = \lambda$ the statement is clear. Let $t' = at$ with $a \in A$. Let $(u'_C)_{C\in\mathcal{C}}$ be a quasi-reconstructible tuple such that u'_C is a prefix of $\pi_C(t')$ for all $C \in \mathcal{C}$. Assume first $\mathrm{alph}(u'_C) \times \{a\} \subseteq I$ for some $C \in \mathcal{C}$ with $a \in C$. Because the tuple is quasi-reconstructible this implies $|u'_C|_a = 0$ for all $C \in \mathcal{C}$. From $u'_C \leq_p a\pi_C(t)$ for any $C \in \mathcal{C}$ with $a \in C$, we deduce that u'_C is independent of a in $M(C)$. Thus u_C is a prefix of $\pi_C(t)$ for each $C \in \mathcal{C}$ which by the inductive hypothesis implies that $(u'_C)_{C\in\mathcal{C}}$ represents a prefix p of t. Observe that p is independent of a: if a were dependent of p then, as $\mathcal{C}$ is a covering of (A,D), a were dependent of $\pi_C(p)$ in $M(C)$ for some $C \in \mathcal{C}$ with $a \in C$ which is a contradiction. Thus in this case the trace p represented by the tuple $(u'_C)_{C\in\mathcal{C}}$ is also a prefix of t. Assume now $\mathrm{alph}(u'_C) \times \{a\} \not\subseteq I$ for all $C \in \mathcal{C}$ with $a \in C$. This implies that a is a prefix of u'_C for all $C \in \mathcal{C}$ with $a \in C$. For $C \in \mathcal{C}$ define u_C such that $u'_C = au_C$ if $a \in C$ and $u_C = u'_C$ otherwise. Clearly, the tuple $(u_C)_{C\in\mathcal{C}}$ is also quasi-reconstructible and u_C is a prefix of $\pi_C(t)$ for each $C \in \mathcal{C}$. By the inductive hypothesis, $(u_C)_{C\in\mathcal{C}}$ represents a prefix p of t. Notice now that $(u'_C)_{C\in\mathcal{C}}$ represents the prefix $p' = ap$ of t'. $\square$

As a direct consequence one obtains a useful tool to prove a prefix (resp. suffix) relation between two traces.

Corollary 4.2 *Let $\mathcal{C}$ be a covering of (A,D), $p, t \in M(A,D)$. Then p is a prefix (suffix) of t if and only if $\pi_C(p)$ is a prefix (resp. suffix) of $\pi_C(t)$ for all $C \in \mathcal{C}$.*

In the following $\mathcal{C}$ denotes a clique-covering of (A,D) (we consider first a trivial one which can be computed in time $\mathcal{O}(|D|)$). A state $p \in \mathrm{Pre}(l)$ is represented by its tuple-representative $(\pi_C(p))_{C\in\mathcal{C}}$ (to be more precise, a prefix of the word $\pi_C(l)$ is represented by its length). To obtain this representation we need to obtain $(\pi_C(l))_{C\in\mathcal{C}}$ in a first phase. Also the failure functions $\phi_{\pi_C(l)}$ for $C \in \mathcal{C}$ have to be calculated. These initializations can be done in time linear to $\sum_{C\in\mathcal{C}} |\pi_C(l)| \leq |A| \cdot |l|$ (we assume here and in the following that the clique-covering has been chosen reasonably, i.e., each letter appears in at most $|A|$ cliques). The structure $(p_C)_{C\in\mathcal{C}}$ with $p_C \in \mathrm{Pre}(\pi_C(l))$ will be denoted $\mathcal{C}$-tuple. While computing Φ_l the $\mathcal{C}$-tuple p may not be reconstructible, but this will hold before and after any call to Φ_l. So we will in most cases identify a $\mathcal{C}$-tuple with the represented prefix. We give the algorithm here (remember that $\varphi_{\pi_C(l)}$ can be computed using $\phi_{\pi_C(l)}$, for each $C \in \mathcal{C}$):

```
function $\Phi_l(p : \mathcal{C}$–tuple, $a$ : letter$) : \mathcal{C}$–tuple      (* $p \in \mathrm{Pre}(l)$ *)
  for each $C \in \mathcal{C}$ with $a \in C$ do $p_C := \varphi_{\pi_C(l)}(p_C, a)$
  $\mathcal{I} := \{C \in \mathcal{C} \mid \exists C' \in \mathcal{C}, \exists b \in C \cap C'\ |p_C|_b > |p_{C'}|_b\}$
  while $\mathcal{I} \neq \emptyset$ do
    for some $C \in \mathcal{I}$ do $p_C := \phi_{\pi_C(l)}(p_C)$
    $\mathcal{I} := \{C \in \mathcal{C} \mid \exists C' \in \mathcal{C}, \exists b \in C \cap C'\ |p_C|_b > |p_{C'}|_b\}$
  endwhile
  return $p$.
```

Lemma 4.3 reflects the basic considerations for the correctness of the algorithm. We formulate the lemma for arbitrary coverings as this will be useful later. For traces p, q we write $p <_o q$, when $p \leq_o q$ and $p \neq q$.

Lemma 4.3 *Let $l, t \in M(A,D)$, $p = \sqcup(Pre(l) \cap Suf(t))$, and let $\mathcal{C}$ be a covering of (A,D). For each $C \in \mathcal{C}$ let p_C be a prefix of $\pi_C(l)$ and a suffix of $\pi_C(t)$ such that $\pi_C(p) \leq_o p_C$. Then (i) and (ii) holds.*

(i) $\pi_C(p) = p_C$ for all $C \in \mathcal{C}$ if, and only if, the tuple $(p_C)_{C\in\mathcal{C}}$ is quasi-reconstructible.

(ii) If $|p_{C'}|_b < |p_C|_b$ for some $b \in C \cap C'$, $C, C' \in \mathcal{C}$, then $\pi_C(p) <_o p_C$.

Proof. The only-if-part of the first statement is clear. For the if-part, we obtain by Proposition 4.1 that the tuple $(p_C)_{C\in\mathcal{C}}$ is reconstructible and represents a prefix q of l which is by Corollary 4.2 also a suffix of t. From $\pi_C(p) \leq_o p_C = \pi_C(q)$ for each $C \in \mathcal{C}$ one deduces $p \leq_o q$. Because p is the maximal element in $\mathrm{Pre}(l) \cap \mathrm{Suf}(t)$, we obtain $p = q$, $\pi_C(p) = \pi_C(q) = p_C$ for $C \in \mathcal{C}$.

Assume now $|p_{C'}|_b < |p_C|_b$ for some $b \in C \cap C'$, $C, C' \in \mathcal{C}$. We have $|p|_b \leq |p_{C'}|_b$, as $\pi_{C'}(p)$ is by assumption an overlap of $p_{C'}$. Clearly, $|p|_b = |\pi_C(p)|_b$ for any $C \in \mathcal{C}$ with $b \in C$. Thus $|p|_b < |p_C|_b$, $\pi_C(p) \neq p_C$. $\Box$

Let $p' = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(pa))$, the value which has to be computed. For the while-loop of the above algorithm we obtain the invariant

$$\pi_C(p') \ \leq_o \ p_C \text{ for all } C \in \mathcal{C}. \tag{1}$$

After the for-loop p_C is for each $C \in \mathcal{C}$ assigned to the value $\sqcup(\mathrm{Pre}(\pi_C(l)) \cap \mathrm{Suf}(\pi_C(pa)))$ (where here p denotes the original value of p). Thus, as $\pi_C(p')$ is a member of the set $\mathrm{Pre}(\pi_C(l)) \cap \mathrm{Suf}(\pi_C(pa))$, (1) holds before entering the while-loop. Now assume that (1) holds before executing the body of the while-loop which means that the conditions of Lemma 4.3 hold (it is easy to see that $p_C \in \mathrm{Pre}(\pi_C(l)) \cap \mathrm{Suf}(\pi_C(ta))$ is also an invariant of the while-loop). We deduce by case (ii) of Lemma 4.3 that $\pi_C(p') <_o p_C$ for each $C \in \mathcal{I}$. Now, $\pi_C(p') <_o p_C$ for a clique C if, and only if, $\pi_C(p') \leq_o \phi_{\pi_C(l)}(p_C)$. Thus (1) still holds after executing the body of the while-loop. The while-loop terminates if, and only if, the tuple $(p_C)_{C\in\mathcal{C}}$ is quasi-reconstructible which allows us to apply Lemma 4.3(i) to deduce that $\pi_C(p') = p_C$ for all $C \in \mathcal{C}$ which implies the correctness of the computation.

Because each application of some failure function $\phi_{\pi_C(l)}$ shortens the value of p_C, in the computation of $\Phi_l(p, a)$ all failure functions are applied at most $\sum_{C\in\mathcal{C}} |p_C\pi_C(a)| - |p'_C| \leq |A| \cdot (|pa| - |p'|)$ times altogether, which gives a time bound for the algorithm (see below how the set $\mathcal{I}$ can be maintained within the given complexity bounds).

Theorem 4.4 *After preprocessing (A,D) in time $\mathcal{O}(|D|)$, and a word v representing the trace $l \in M(A,D)$ in time $\mathcal{O}(|A|\cdot|l|)$, the computation of $\Phi_l(p,a)$ needs, for any prefix p of l, and any $a \in A$, at most $c \cdot (|pa| - |\Phi_l(p,a)| + 1)$ time for some $c \in \mathcal{O}(|A|)$.*

To maintain $\mathcal{I}$ within the given bound, we represent sets over a given universe by a structured data type which allows to perform the question whether a given element is in the set, the operations of inclusion and exclusion of some element from the universe in constant time. Further it should be possible to access some (say the first) element in a nonempty set in constant time. These requirements can be fulfilled by using a doubly linked list of the elements which are in the set together with an array which assigns to each element of the universe a pointer to its representation in the list (elements not in the set obtain the special value nil). Attached to a $\mathcal{C}$-tuple $p = (p_C)_{C\in\mathcal{C}}$ keep for each $C \in \mathcal{C}$ and each $a \in C$ counters recording the value $|p_C|_a$, for each $a \in A$ the integer value m_a, the set $S_a \subseteq \{C \in \mathcal{C} \mid a \in C\}$, and for each $C \in \mathcal{C}$ a set $R_C \subseteq C$. These structures are designated to obey (for $a \in A$ and $C \in \mathcal{C}$) the invariants

$$\begin{aligned} m_a &= \min\{|p_C|_a \mid C \in \mathcal{C}, a \in C\}, \\ S_a &= \{C \in \mathcal{C} \mid a \in C, |p_C|_a = m_a\}, \\ R_C &= \{a \in C \mid |p_C|_a > m_a\}, \\ \mathcal{I} &= \{C \in \mathcal{C} \mid |p_C|_a < m_a\}. \end{aligned} \tag{2}$$

Thus, outside the computation of Φ_l we have $|p|_a = m_a = |p_C|_a$ for $a \in C$, $S_a = \{C \in \mathcal{C} \mid a \in C\}$ for $a \in A$, $R_C = \emptyset$ for $C \in \mathcal{C}$, and $\mathcal{I} := \emptyset$ as then, the $\mathcal{C}$-tuple p is reconstructible. While computing Φ_l basically two operations affect these structures: the application of some failure function, and the concatenation of a to some p_C with $a \in C$. We show how the given structures can be updated after each operation such that the invariants still hold and the time needed for all updates while computing $\Phi_l(p,a)$ is linear in $\sum_{C\in\mathcal{C}} |\pi_C(pa)| - |\pi_C(\Phi_l(p,a))|$. First examine the application of some failure function $\phi_{\pi_C(l)}$ to $p_C \neq \lambda$. We will call this operation "shorten p_C".

```
shorten $p_C$:
    Let $q \in C^*$ be such that $\phi_{\pi_C(l)}(p_C)q = p_C$
    $p_C := \phi_{\pi_C(l)}(p_C)$
    **for each** $b \in \text{alph}(q)$ **do**
        determine $|p_C|_b$ to $|p_C|_b - |q|_b$
        **if** $|p_C|_b < m_b$ **then**
            $\mathcal{I} := (\mathcal{I} \cup S_b) \setminus \{C\}$
        for each $C' \in S_b \setminus \{C\}$ do $R_{C'} := R_{C'} \cup \{b\}$
        $S_b := \{C\}$; $R_C := R_C \setminus \{b\}$
        $m_b := |p_C|_b$
    else if $|p_C|_b = m_b$ then
        $S_b := S_b \cup \{C\}$; $R_C := R_C \setminus \{b\}$
        if $R_C = \emptyset$ then $\mathcal{I} := \mathcal{I} \setminus \{C\}$
```

Notice that an execution of "shorten p_C" preserves the invariants in (2). For time-complexity observe that it suffices to count the number of changes to some $R_{C'}$ plus the length of q (the time needed is linear in this value). During an execution there may be up to $|\mathrm{alph}(q)|\cdot|\mathcal{C}|$ inclusions to some $R_{C'}$, however, there are at most $|\mathrm{alph}(q)| \leq |q|$ exclusions (namely from R_C). As in the (intended) computation of Φ_l for any $C' \in \mathcal{C}$ the value of $R_{C'}$ equals $\emptyset$ before and after the computation, the total number of exclusions equals the total number of inclusions such that it suffices to count the exclusions. This yields that the total time needed for all executions of the above routine during the computation of $p' = \Phi_l(p, a)$ is bounded linearly by $\sum_{C\in\mathcal{C}} |\pi_C(pa)| - |\pi_C(p')| \leq \Sigma \cdot (|pa| - |p'|)$.

We give a more detailed formulation of the algorithm for the computation of Φ_l:

```
function $\Phi_l(p : \mathcal{C}$–tuple, $a$ : letter) : $\mathcal{C}$–tuple        (* $p \in \mathrm{Pre}(l)$ *)
  (* $m_a = |p|_a$ and $S_a = \{C \in \mathcal{C} \mid a \in C\}$ for $a \in A$ *)
  (* $R_C = \emptyset$ for $C \in \mathcal{C}$ and $\mathcal{I} := \emptyset$ *)
  for each $C \in \mathcal{C}$ with $a \in C$ do
    while $p_C \neq \lambda$ and $p_C a \notin \mathrm{Pre}(\pi_C(l))$ do shorten $p_C$
    if $p_C a \in \mathrm{Pre}(\pi_C(l))$ then $p_C := p_C a$
  endfor
  if $p_C = \lambda$ for some $C \in S_a$ then (* it holds $m_a = 0$ *)
    for each $C \in S_a$ do $p_C := \lambda$
  else $m_a := m_a + 1$
  while $\mathcal{I} \neq \emptyset$ do shorten $p_C$ for some $C \in \mathcal{I}$
  return $p$.
```

To examine the correctness of the modifications, first assume that the if-statement in the first for-loop would not be present. Then the for-loop would preserve the invariants in (2). When the if-statement is present the invariants may be violated. However we are able to reestablish them after the for-loop. There are two cases: First assume that for some $C \in \mathcal{C}$ with $a \in C$, a was not appended to p_C. Then we know $p_C = \lambda$ and we are, by Proposition 4.1, able to deduce that after a correct computation of $\Phi_l(p, a)$, $p_C = \lambda$ for any C with $a \in C$. Notice that $C \in S_a$ implies $p_C = \lambda$ or $p_C = a$. Only the $C \in S_a$ with $p_C = a$ violate the invariants. By setting $p_C = \lambda$

for those C we are able to reestablish (2). In the second case a had been appended to each p_C with $C \in \{C \in \mathcal{C} \mid a \in C\}$ which implies $p_C \neq \lambda$ for those C. In this case it suffices to add one to m_a to reestablish (2).

The total time needed for the processing of both if-statements is bounded linearly by $|\{C \in \mathcal{C} \mid a \in C\}| = \sum_{C\in\mathcal{C}} |\pi_C(a)| \leq |A|$. We obtain altogether that the time needed for the computation of $p' = \Phi_l(p, a)$ is bounded linearly by $\sum_{C\in\mathcal{C}} |\pi_C(paa)| - |\pi_C(p')| \leq |A| \cdot (|pa| - |p'| + 1)$.

Given the computation of Φ_l it is easy to deduce an algorithm for the suffix problem. One just has to simulate the automaton $\mathcal{A}_l$, where l is determined by the first input word v, on the second word x: In a first phase the structures depending from v and (A,D) have to be computed (see above). Let $x = a_1 \ldots a_n$ with $a_i \in A$ for $1 \leq i \leq n$. Set $p_0 = \lambda$, and compute successively the values $p_i = \Phi_l(p_{i-1}, a_i)$ for $i = 1$ to n. Test finally whether $p_n = l$.

The time used for the second phase is bounded above by $\sum_{i=0}^{n-1} c\cdot(|p_i a_{i+1}| - |p_{i+1}| + 1)$ for some $c \in \mathcal{O}(|A|)$. This equals $c\cdot(2n - |p_n|)$ which is in $\mathcal{O}(|A|\cdot|x|)$.

In [13, Algorithm 5.2] the loop is only repeated when $p_i \neq l$. Thus the algorithm decides whether l is a suffix of a trace represented by some prefix $a_1 \ldots a_i$ of x. The comparison $p_i = l$ can be done in constant time when a set $\{C \in \mathcal{C} \mid p_{i,C} \neq \pi_C(l)\}$ is maintained, which needs altogether the time $\mathcal{O}(|A| \cdot |vx|)$. This way we obtain the same complexity for this modified algorithm.

Theorem 4.5 *On input of (A,D), and $v, x \in A^*$ it is decidable in time linear to $|A| \cdot |vx| + |D|$ using space linear to $|A| \cdot |v| + |D|$, whether the trace $l \in M(A,D)$ represented by v is a suffix of a trace represented by x (resp. whether l is a suffix of a trace represented by some prefix of x).*

It is sometimes even better to compute a covering of (A,D) by maximal cliques, i.e., cliques which don't remain cliques when including some other letter, which can be done by an efficient greedy algorithm. In this case, after calculating the covering, one even obtains the time-complexity $\mathcal{O}(|vx|)$, when M(A,D) is a free monoid. Notice that the above observations about complexity hold for any reasonable clique-covering. So the time-complexity gets never worse than $\mathcal{O}(|A| \cdot |vx|)$ when using arbitrary reasonable coverings (not considering the time used for the calculation of the covering). A slight improvement is also obtained by considering independent components of the dependence graph independently: Let (A,D) be the union of several subgraphs $H_i = (A_i,D_i)$ such that $A_i \cap A_j = \emptyset$ for $1 \leq i < j \leq k$. Then every tuple $(u_i)_{1\leq i\leq k}$ with $u_i \in M(H_i)$ is quasi-reconstructible. So by Proposition 4.1, to determine whether l is suffix of t it suffices to determine independently whether $\pi_{A_i}(l)$ is a suffix of $\pi_{A_i}(t)$ for $1 \leq i \leq k$, which can be done in time $\sum_{i=1}^{k} |D_i| + |A_i| \cdot |vx|$. When $D = \emptyset$, i.e., when $\mathrm{M}(A,D)$ is a free commutative monoid, one obtains time-complexity $\mathcal{O}(|A| + |vx|)$ this way.

In a free monoid an automaton for the suffix-language can easily be transformed to an automaton for the factor-language (just stay in the final state, when it is reached once). However, in a free partially commutative monoid this is not so easy.

Example 1 Assume a monoid $\mathrm{M}(A,D)$ with $a, b, c \in A$ and $(a, b) \in D$, $(b, c) \in I$. Let $l = ac \in \mathrm{M}(A,D)$ and $x = abc \in A^*$. Because $x \sim_I acb$, l is a factor of the trace represented by x. However, l is not a suffix of a trace represented by some prefix $\{\lambda$, a, ab, $abc\}$ of x.

5 Solving the Factor Problem

We first construct (for $l \in \mathrm{M}(A,D)$) a finite $\mathrm{M}(A,D)$-automaton recognizing $\mathrm{M}(A,D) \cdot l \cdot \mathrm{M}(A,D)$ using a known result about the concatenation of recognizable trace-languages. Then we give an algorithm which simulates this automaton in linear time for fixed (A,D).

5.1 Recognizing $\mathbf{M}(A,D) \cdot l \cdot \mathbf{M}(A,D)$

The concatenation of recognizable trace languages is constructively recognizable by Theorem 5.1 (the construction was given in the proof of [7, Proposition 2.2.1]).

Theorem 5.1 *For $i \in \{1, 2\}$ let $\mathcal{A}_i = (Q_i, A, \delta_i, q_{0i}, F_i)$ be a finite M(A,D)-automaton recognizing $L_i \in$ M(A,D). Then the trace-language $L_1 \cdot L_2$ is recognized by the nondeterministic finite M(A,D)-automaton*

$$\mathcal{A} \;=\; (Q_1 \times \mathcal{P}(A) \times Q_2,\ A,\ \delta,\ (q_{01}, \emptyset, q_{02}),\ F_1 \times \mathcal{P}(A) \times F_2),$$

where $((p, B, q), a, (p', B', q')) \in \delta$ (for $B, B' \subseteq A$) if, and only if,

(i) $p' = p \in Q_1,\quad B' = \ B \cup \{a\},\qquad (q, a, q') \in \delta_2,\quad$ *or*

(ii) $q' = q \in Q_2,\quad B' = \ B,\ a \notin D(B),\quad (p, a, p') \in \delta_1.$

The constructed automaton is a product automaton of $\mathcal{A}_1$ and $\mathcal{A}_2$. On each input letter it is nondeterministically chosen, whether $\mathcal{A}_1$ or $\mathcal{A}_2$ consumes it. In the alphabetic component the letters already read by $\mathcal{A}_2$ are remembered. $\mathcal{A}_1$ may only consume letters independent of this set. It holds

$$(p, B, q) \quad \xrightarrow{t}_{\mathcal{A}} \quad (p', B', q')$$

if, and only if, for some $r, s \in \mathrm{M}(A,D)$ such that $t = rs$, and $\mathrm{alph}(r) \times B \subseteq I$:

$$p \xrightarrow[\mathcal{A}_1]{r} p', \quad q \xrightarrow[\mathcal{A}_2]{s} q', \quad \text{and } B' = B \cup \mathrm{alph}(s).$$

We already know the automaton $\mathcal{A}_l$ recognizing $\mathrm{M}(A,D) \cdot l$. $\mathrm{M}(A,D)$ itself, is recognized by the trivial automaton $(\{q\}, A, \{(q,a,q) \mid a \in A\}, q, \{q\})$. Using the construction in Theorem 5.1 we obtain the state set $\mathrm{Pre}(l) \times \mathcal{P}(A) \times \{q\}$. The third component can be omitted, as it is unique. The set of final states is then $\{l\} \times \mathcal{P}(A)$. Notice now that a final state (l, B') is reachable from a state $(p, B) \in \mathrm{Pre}(l) \times \mathcal{P}(A)$ by a trace t only if $\pi_{D(B)}(p) = \pi_{D(B)}(l)$. This yields

Theorem 5.2 *The trace language $M(A,D) \cdot l \cdot M(A,D)$ is recognized by the nondeterministic finite $M(A,D)$-automaton $N_l = (S_l,\ A,\ \delta,\ (\lambda, \emptyset),\ \{l\} \times \mathcal{P}(A))$, where*

$$S_l \;=\; \{(p, B) \in Pre(l) \times \mathcal{P}(A) \mid \pi_{D(B)}(p) = \pi_{D(B)}(l)\},$$

and $((p,B), a, (p',B')) \in \delta$ if, and only if, $(p,B), (p',B') \in S_l$, and either

(i) $p' = p, \qquad B' = B \cup \{a\}, \qquad$ *or*

(ii) $p' = \Phi_l(p,a), \quad B' = B, \quad a \notin D(B)$.

In the following we denote by S_l and N_l the notions given above. For a trace t let $S_l(t) = \{q \in S_l \mid (\lambda, \emptyset) \xrightarrow[N_l]{t} q\}$. From the construction of N_l we obtain

Lemma 5.3 *Let $l, t \in M(A,D)$. Then $(p,B) \in S_l(t)$ if, and only if, $(p,B) \in S_l$ and for some traces r, s: $t = rs$, $p = \sqcup(Pre(l) \cap Suf(r))$, and $B = alph(s)$.*

5.2 An Algorithm for the Factor Problem

We give some preliminary results.

Lemma 5.4 *Let $l, t \in M(A,D)$, $p = \sqcup(Pre(l) \cap Suf(t))$, and let $\{H, G\}$ be a covering of (A,D). If $\pi_H(p) = \pi_H(l)$ then $\pi_G(p) = \sqcup(Pre(\pi_G(l)) \cap Suf(\pi_G(t)))$.*

Proof. Let $p_G = \sqcup(\mathrm{Pre}(\pi_G(l)) \cap \mathrm{Suf}(\pi_G(t)))$. Observe that $\pi_G(p) \leq_o p_G$ and that $(p_G, \pi_H(p))$ is quasi-reconstructible, because $|l|_b = |p_b| \leq |p_G|_b \leq |l|_b$ for any $b \in H \cap G$. By Lemma 4.3(i) $\pi_G(p) = p_G$. $\square$

The following lemma shows that if $(p,B) \in S_l(t)$ then p is uniquely determined by $D(B)$ and some suffix of $\pi_{D(A-D(B))}(t)$. As a consequence there is at most one p such that $(p,B) \in S_l(t)$ for a given $B \subseteq A$. Thus, if (A,D) is fixed, the set $S_l(t)$ of states reachable in N_l by t has constant size for any $l, t \in \mathrm{M}(A,D)$.

Lemma 5.5 *Let $l, t \in M(A,D)$. Let $(p, B) \in S_l(t)$, and $\Gamma = D(A - D(B))$, then $\pi_{D(B)}(p) = \pi_{D(B)}(l)$ and $\pi_\Gamma(p) = \sqcup(Pre(\pi_\Gamma(l)) \cap Suf(\pi_\Gamma(t)))$. Hence, for any $B \subseteq \Sigma$ there is at most one p such that $(p, B) \in S_l(t)$.*

Proof. By Lemma 5.3 there are some traces $r, s \in \mathrm{M}(A,D)$ such that $t = rs$, $p = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(p))$, and $B = \mathrm{alph}(s)$. Because $(p, B) \in S_l$, we have $\pi_{D(B)}(p) = \pi_{D(B)}(l)$. Observe that $\{\Gamma, D(B)\}$ is a covering of (A,D). By Lemma 5.4 $\pi_\Gamma(p)$ is the greatest overlap of $\pi_\Gamma(l)$ and $\pi_\Gamma(r)$. We need to show $\pi_\Gamma(s) = \lambda$ which implies $\pi_\Gamma(r) = \pi_\Gamma(t)$. But this is clear, because $B \cap D(A - D(B)) = \emptyset$ for symmetric $D \subseteq A \times A$. Notice finally, that for $(p_1, B), (p_2, B) \in S_l(t)$, $p_1 = p_2$ is implied by Proposition 2.2 using the equalities $\pi_{D(B)}(p_1) = \pi_{D(B)}(l) = \pi_{D(B)}(p_2)$ and $\pi_\Gamma(p_1) = \sqcup(\mathrm{Pre}(\pi_\Gamma(l)) \cap \mathrm{Suf}(\pi_\Gamma(t)) = \pi_\Gamma(p_2)$. $\square$

We define a notation for the corresponding suffixes and give some other notations which will be useful for the construction of the algorithm:

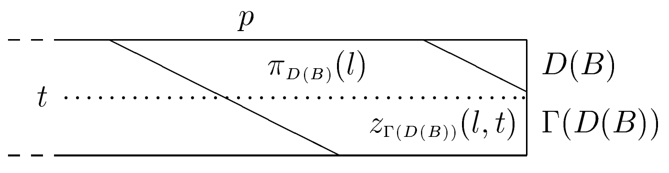
Definition 5.6 *For $l, t \in M(A,D)$, $B \subseteq A$, let*

$$z_B(l, t) = \sqcup(Pre(\pi_B(l)) \cap Suf(\pi_B(t))),$$
$$\mathcal{B}(l, t) = \{B \subseteq A \mid (p, B) \in S_l(t)\},$$
$$\Gamma(B) = D(A - B).$$

Let further, for a family $\mathcal{B} \subseteq \mathcal{P}(A)$, $D(\mathcal{B}) = \{D(B) \mid B \in \mathcal{B}\}$.

By Lemma 5.5, the tuple $(z_{\Gamma(D(B))}(l, t), \pi_{D(B)}(l))$ is a tuple-representative of p, when $(p, B) \in S_l(t)$. The result is visualized in Figure 1. In the picture, the alphabet ranges on the vertical axis, the dotted line is drawn where $D(B)$ and $\Gamma(D(B))$ intersect.

Figure 1: Location of p in t when $(p, B) \in S_l(t)$.



To decide whether a trace t is accepted by N_l, i.e., whether l is a factor of t, it suffices to examine if there is a $B \in \mathcal{B}(l, t)$ such that $z_{\Gamma(D(B))}(l, t) = \pi_{\Gamma(D(B))}(l)$. Lemma 5.7 shows how the set $D(\mathcal{B}(l, ta))$ can be computed from $D(\mathcal{B}(l, t))$, $a \in A$, and some $z_{\Gamma(D(B))}(l, t)$ for $B \subseteq A$.

Lemma 5.7 *Let $l, t \in M(A,D)$, $a \in A$, $\Delta' \in \mathcal{P}(A)$. Then $\Delta' \in D(\mathcal{B}(l, ta))$ if, and only if, there is a $\Delta \in D(\mathcal{B}(l, t))$ such that either*

(i) $\Delta' = \Delta \cup D(a)$ and $|z_{\Gamma(\Delta)}(l, t)|_b = |l|_b$ for all $b \in D(a) - \Delta$, or

(ii) $a \notin \Delta' = \Delta$, and the tuple $(\pi_\Delta(l), z_{\Gamma(\Delta)}(l, ta))$ is quasi-reconstructible.

Proof. If $\Delta' \in D(\mathcal{B}(l,ta))$ then there is a pair $(p',B') \in S_l(ta)$ such that $\Delta' = D(B')$. Then, by definition of $S_l(ta)$, there is a state $(p,B) \in S_l(t)$ with $(p,B) \xrightarrow[N_l]{a} (p',B')$. Either $B' = B \cup \{a\}$ and $p = p'$ or $B = B'$ and $a \notin D(B')$. With $\Delta = D(B)$ this means either $\Delta' = \Delta \cup D(a)$ or $a \notin \Delta' = \Delta$. As by Lemma 5.5 the tuple $(\pi_{\Delta'}(l), z_{\Gamma(\Delta')}(l,ta))$ represents p' it is reconstructible. Thus in the case $a \notin \Delta' = \Delta$ (ii) is necessary. In the case $\Delta' = \Delta \cup D(a)$ with $p = p'$ we have, because $(p',B') \in S_l$, $|p|_b = |p'|_b = |l|_b$ for $b \in \Delta'$. Thus, in this case, $|z_{\Gamma(\Delta)}(l,t)|_b = |p|_b = |l|_b$ for $b \in \Delta' - \Delta = D(a) - \Delta \subseteq \Gamma(\Delta)$.

Let us now show sufficiency. Let $\Delta \in D(\mathcal{B}(l,t))$, let $(p,B) \in S_l(t)$ be such that $\Delta = D(B)$. By Lemma 5.5 $\pi_{\Gamma(\Delta)}(p) = z_{\Gamma(\Delta)}(l,t)$. Thus $|z_{\Gamma(\Delta)}(l,t)|_b = |l|_b$ for $b \in D(a) - \Delta$ implies $|p|_b = |l|_b$ for all $b \in \Delta \cup D(a)$ (remember $\pi_\Delta(p) = \pi_\Delta(l)$, as $(p,B) \in S_l$). This implies $(p, B \cup \{a\}) \in S_l$ and, by the definition of N_l, $(p, B \cup \{a\}) \in S_l(ta)$. Thus, if (i) holds, we have $\Delta' = \Delta \cup D(a) \in D(\mathcal{B}(l,ta))$. Assume now that (ii) holds. Let $p' = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(pa))$. We show $(p',B) \in S_l$ which implies $\Delta \in D(\mathcal{B}(l,ta))$. By Lemma 5.5 $z_{\Gamma(\Delta)}(l,t) = \pi_{\Gamma(\Delta)}(p)$, thus by Lemma 3.5, $z_{\Gamma(\Delta)}(l,ta)$ is the greatest overlap of $\pi_{\Gamma(\Delta)}(l)$ and $\pi_{\Gamma(\Delta)}(pa)$ which implies that $\pi_{\Gamma(\Delta)}(p')$ is an overlap of $z_{\Gamma(\Delta)}(l,ta)$. Notice also $\pi_\Delta(pa) = \pi_\Delta(p) = \pi_\Delta(l)$, because $a \notin \Delta$ and $(p,B) \in S_l$, thus $\pi_\Delta(p') \leq_o \pi_\Delta(l)$. As the tuple $(\pi_\Delta(l), z_{\Gamma(\Delta)}(l,ta))$ is required to be quasi-reconstructible, we can apply Lemma 4.3(i) on the covering $\{\Delta, \Gamma(\Delta)\}$ to deduce that the tuple is, in fact, a tuple-representative of p', i.e., $\pi_{\Gamma(\Delta)}(p') = z_{\Gamma(\Delta)}(l,ta)$ and $\pi_\Delta(p') = \pi_\Delta(l)$. Note that the latter implies $(p',B) \in S_l$. $\square$

Theorem 5.8 *On input $v, x \in A^*$ it is decidable in time linear to $|vx|$ using space linear in $|v|$, whether the trace $l \in M(A,D)$ represented by v is a factor of the trace $t \in M(A,D)$ represented by x.*

Proof. First preprocess v like in the proof of Theorem 4.4. Let $x = a_1 \dots a_n$ with $a_i \in A$ for $1 \leq i \leq n$. Let $\mathcal{B}_0 = \{\emptyset\}$ $(= D(\mathcal{B}(l,\lambda)))$. Now proceed in n stages: for $0 \leq i \leq n-1$ let $\mathcal{B}_{i+1}$ be the union of the two sets

(i) $\{\Delta \cup D(a_i) \mid \Delta \in \mathcal{B}_i, \text{ and } |z_{\Gamma(\Delta),i}|_b = |l|_b \text{ for all } b \in D(a) - \Delta\}$

(ii) $\{\Delta \in \mathcal{B}_i \mid a \notin \Delta, \text{ and } (\pi_\Delta(l), z_{\Gamma(\Delta),i+1}) \text{ is quasi-reconstructible}\}$,

where the values $z_{\Gamma(\Delta),i}$ are for $\Delta \in D(\mathcal{P}(A))$ obtained by $z_{\Gamma(\Delta),0} = \lambda$, and

$$z_{\Gamma(\Delta),i+1} = \begin{cases} \Phi_{\pi_{\Gamma(\Delta)}(l)}(z_{\Gamma(\Delta),i}, a_{i+1}) & \text{if } a \in \Gamma(\Delta), \\ z_{\Gamma(\Delta),i} & \text{else.} \end{cases}$$

Finally test, whether there exists a $\Delta \in \mathcal{B}_n$ such that $z_{\Gamma(\Delta),n} = \pi_{\Gamma(\Delta)}(l)$.

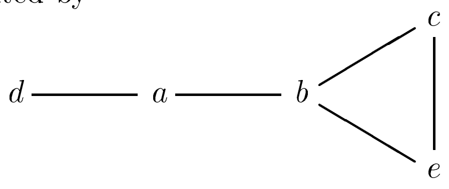
Using Theorem 3.6 one deduces $z_{\Gamma(\Delta),i} = z_{\Gamma(\Delta)}(l, a_1 \dots a_i)$. The correctness of the algorithm is thus due to Lemma 5.7 which implies by induction on i that $\mathcal{B}_i = D(\mathcal{B}(l, a_1 \dots a_i))$ for $0 \le i \le n$.

Examine now complexity. Notice that a set $\mathcal{B}_i \subseteq D(\mathcal{P}(A))$ has constant size, as (A,D) is constant. Thus $\mathcal{B}_{i+1}$ can be computed from $\mathcal{B}_i$ in constant time using $z_{\Gamma(\Delta),i}$ and $z_{\Gamma(\Delta),i+1}$ for $\Delta \in \mathcal{B}_i$. The successive computation of $z_{\Gamma(\Delta),i}$, when done for $1 \le i \le n$, takes time linear to $|x|$ for each $\Delta \in D(\mathcal{P}(A))$, as can be seen by the considerations in the proof of Theorem 4.5. Together with the preprocessing of v this yields the time-complexity $\mathcal{O}(|vx|)$. For space-complexity notice that between stage i and $i+1$ only the values of $\mathcal{B}_i$, and $z_{\Gamma(\Delta),i}$ for $\Delta \in D(\mathcal{P}(A))$ have to be remembered. This needs constant space in addition to the $\mathcal{O}(|v|)$-size structures depending from v. $\square$

The set $D(\mathcal{B}(l,t))$ may equal $D(\mathcal{P}(A))$, so that the time-complexity of the given algorithm is, in general, exponential in $|A|$ when (A,D) is considered as a part of the input. However, if we have an upper bound k for the shortest path between any two vertices in (A,D) (consider only connected (A,D)), the algorithm remains efficient. Notice, in this case $|D(\mathcal{P}(A))| \le 2 + \sum_{i=1}^{k-1} \binom{|A|}{i} \le 2 + |A|^{k-1}$ which yields the time bound $\mathcal{O}(|A|^k \cdot |vx|)$ when using a trivial clique-covering. In the free monoid, for example, we have $D(\mathcal{P}(A)) = \{\emptyset, A\}$ and the time bound $\mathcal{O}(|A| \cdot |vx|)$. Similar to what was said to the algorithm for the suffix problem it is better to consider independent (i.e., not connected) components of (A,D) independently. This way one obtains the time-complexity $\mathcal{O}(|vx|)$ when $D = \emptyset$, i.e., when (A,D) is a free commutative monoid.

If one adjusts [13, Algorithm 6.2] to our framework, one could roughly say that in that algorithm, between stage i and stage $i+1$, there is only one $B \subseteq A$ remembered (which should be maximal in $\mathcal{B}(l, a_1 \dots a_i)$). However, we are able to show that this information does not suffice to determine the next state correctly:

Example 2 Let $A = \{a, b, c, d, e\}$, and let the dependence relation D be the reflexive and symmetric closure of $\{(d,a), (a,b), (b,c), (c,e), (e,b)\}$. (A,D) is graphically represented by



Let $l = adce$, $t = acebcec^{n-1}$, and $t' = acebec^n$ for some $n \ge 2$. Then $\mathcal{B}(l,t) = \{\emptyset, \{c\}, \{b,c,e\}\}$, and $\mathcal{B}(l,t') = \{\emptyset, \{b,c,e\}\}$ thus in both sets $B =$

$\{b, c, e\}$ is maximal (notice also $z_B(l,t) = z_B(l,t')$ for $B \subseteq A$ which is due to the fact that $z_C(l,t) = z_C(l,t')$ for all trivial cliques C of (A,D)). But $\mathcal{B}(l, ta) = \{\emptyset, \{c\}\}$, and $\{c\} \notin \mathcal{B}(l, t'a) = \{\emptyset\}$.

6 Extensible Trace Pairs

Extensible pairs were introduced in [13] to investigate the factor problem. They allow us to study the automaton N_l and the algorithm for the factor problem in a less technical way.

Definition 6.1 *Let $l, t \in M(A,D)$. An extensible trace pair (short, extensible pair) of (l,t) is a pair $(p,s) \in Pre(l) \times Suf(t)$ with $ps \leq_s t$, and $alph(p^{-1}l) \times alph(s) \subseteq I$, where $p^{-1}l$ denotes the unique suffix of l with $p(p^{-1}l) = l$.*

Clearly, $\text{alph}(p^{-1}l) \times \text{alph}(s) \subseteq I$ if, and only if, $\pi_\Delta(p) = \pi_\Delta(l)$ for $\Delta = D(\text{alph}(s))$. Therefore we obtain the following alternative definition of extensible pairs:

> The pair (p, s) is an extensible pair of (l, t) if, and only if, $ps \leq_s t$ and $(p, \text{alph}(s)) \in S_l$.

Surprisingly the extensible pairs of (l,t) form a sublattice of the direct product of the lattices $(\text{Pre}(l), \leq_p)$ and $(\text{Suf}(t), \leq_s)$.

Theorem 6.2 *Let (p_1, s_1) and (p_2, s_2) be both extensible pairs of (l,t). Then*

$$(p_1 \sqcup_p p_2, s_1 \sqcup_s s_2) \quad \text{and} \quad (p_1 \sqcap_p p_2, s_1 \sqcap_s s_2)$$

are extensible pairs of (l,t), too. Further it holds

$$\begin{aligned} p_1 s_1 \sqcup_s p_2 s_2 &= (p_1 \sqcup_p p_2)(s_1 \sqcup_s s_2), \text{ and} \\ p_1 s_1 \sqcap_s p_2 s_2 &= (p_1 \sqcap_p p_2)(s_1 \sqcap_s s_2). \end{aligned}$$

Proof. Let (p_1, s_1) and (p_2, s_2) be both extensible pairs of (l,t). Let $p = p_1 \sqcup_p p_2$ and $s = s_1 \sqcup_s s_2$. Because projections are lattice morphisms, $|p|_b = \max(|p_1|_b, |p_2|_b) = |l|_b$, for each $b \in D(\text{alph}(s)) = D(\text{alph}(s_1) \cup \text{alph}(s_2))$. This implies $(p, \text{alph}(s)) \in S_l$. We now show $ps = p_1 s_1 \sqcup_s p_2 s_2$ which implies $ps \leq_s t$, thus (p,s) is an extensible pair of (l,t). Let $\mathcal{C}$ be a clique-covering of (A,D). By Proposition 2.2 it suffices to show for each $C \in \mathcal{C}$ that the words $\pi_C(ps)$ and $\pi_C(p_1 s_1 \sqcup_s p_2 s_2)$ are equal. To do this, we heavily use the fact that projections are monoid and lattice morphisms (see Lemma 3.2). Let $C \in \mathcal{C}$. Assume by symmetry that the word $\pi_C(p_2 s_2)$ is not longer than $\pi_C(p_1 s_1)$, i.e.,

$$\pi_C(p_2 s_2) \leq_s \pi_C(p_1 s_1), \tag{3}$$

thus $\pi_C(p_1s_1 \sqcup_s p_2s_2) = \pi_C(p_1s_1)$. We now show that (3) implies $\pi_C(p_2) \leq_p \pi_C(p_1)$ and $\pi_C(s_2) \leq_s \pi_C(s_1)$ from which one deduces $\pi_C(ps) = \pi_C(p_1s_1)$. If $\pi_C(s_2) \neq \lambda$ then $\pi_C(p_2) = \pi_C(l)$, because $C \subseteq D(\mathrm{alph}(s_2))$ (remember C is a clique in (A,D)). Therefore, using the assumption (3), $\pi_C(s_1) \neq \lambda$ and $\pi_C(p_1) = \pi_C(l)$ too, which implies $\pi_C(s_2) \leq_s \pi_C(s_1)$. Let now $\pi_C(s_2) = \lambda$. If $\pi_C(s_1) \neq \lambda$ then $\pi_C(p_2) \leq_p \pi_C(p_1) = \pi_C(l)$. If $\pi_C(s_1) = \lambda$ then by (3) $\pi_C(p_2) \leq_s \pi_C(p_1)$ which implies $\pi_C(p_2) \leq_p \pi_C(p_1)$.

A proof for the infimum is obtained in similar way: Let $p = p_1 \sqcap_p p_2$, $s = s_1 \sqcap_s s_2$. First observe $(p, \mathrm{alph}(s)) \in S_l$ because $|l|_b = |p_1|_b = |p_2|_b = |p|_b$ for $b \in D(\mathrm{alph}(s)) \subseteq D(\mathrm{alph}(s_1)) \cap D(\mathrm{alph}(s_2))$. Now we show $ps = p_1s_1 \sqcap_s p_2s_2$ using projections to free monoids C^*: we show that for an arbitrary clique C of (A,D), $\pi_C(ps) = \pi_C(p_1s_1 \sqcap_s p_2s_2)$. By symmetry we assume $\pi_C(p_2s_2) \leq_s \pi_C(p_1s_1)$, which implies $\pi_C(p_2) \leq_p \pi_C(p_1)$ and $\pi_C(s_2) \leq \pi_C(s_1)$ as we saw above. Therefore $\pi_C(p) = \pi_C(p_2)$, and $\pi_C(s) = \pi_C(s_2)$ which implies $\pi_C(ps) = \pi_C(p_2s_2) = \pi_C(p_1s_1 \sqcap_s p_2s_2)$. □

Theorem 6.2 was inspired by [13, Theorem 4.1 (1)] which erroneously states that for two extensible pairs (p_1, s_1), (p_2, s_2) there is an extensible pair (p, s) with $s_1, s_2 \leq_s s$ and $p_1, p_2 \leq_o p$. Let $s = s_1 \sqcup_s s_2$ and let $r \in \mathrm{M}(A,D)$ be such that $t = rs$. In the proof of [13, Theorem 4.1 (1)] it was misleadingly assumed that $\pi_C(p_i)$ is a suffix of $\pi_C(r)$ for any trivial clique C and $i \in \{1, 2\}$. However, this need not to be true for $C \subseteq D(\mathrm{alph}(s_1)$ and $C \not\subseteq D(\mathrm{alph}(s_2))$. The only thing one could say in this case is that $\pi_C(p_1) = \pi_C(l)$ is a suffix of $\pi_C(r)$, and that $\pi_C(p_2)$ is a prefix of $\pi_C(l)$ and a suffix of $\pi_C(t)$. However, this allows not to deduce that $\pi_C(p_2)$ is a suffix of $\pi_C(l)$. We exploit this error in the following counterexample which can be generalized easily to any trace monoid which is not a free commutative monoid (and also to cases where p_1 and p_2 are incomparable via $\leq_p$).

Example 3 Let $\mathrm{M}(A,D)$ be the free monoid $\{a, b\}^*$. Let $l = ab$, $t = aba$. The extensible pairs of (l, t) are (ab, a), (a, λ), and (λ, λ). Let now $(p_1, s_1) = (ab, a)$, and $(p_2, s_2) = (a, \lambda)$. Observe that there is no extensible pair (p, s) of (l, t) such that p_1 and p_2 is a suffix of p.

A pair $(p, B) \in \mathrm{Pre}(l) \times \mathcal{P}(A)$ is called extensible trace-alphabet-pair of (l, t) if (p, s) is an extensible pair of (l, t) for some $s \leq_s t$ with $B = \mathrm{alph}(s)$. It is easy to see how Theorem 6.2 is transfered to extensible trace-alphabet-pairs to show that they form a sublattice of the direct product of $(\mathrm{Pre}(l), \leq_p)$ and $(\mathcal{P}(A), \subseteq)$. Lemma 6.3 gives the relationship between extensible trace-alphabet-pairs of (l, t) and the states reachable in the automaton N_l by t: The elements of $S_l(t)$ are those extensible trace-alphabet-pairs (p, B) of (l, t) whose first component p is maximal with respect to the extensible trace-alphabet-pairs having the same second component B.

Lemma 6.3 *Let $l, t \in M(A,D)$. It holds (i) and (ii).*

(i) The pair (p, B) is an extensible trace-alphabet-pair of (l, t) if, and only if, $(p, B) \in S_l$, and $p \leq_o q$ for a trace q such that $(q, B) \in S_l(t)$

(ii) $(q, B) \in S_l(t)$ if, and only if, (q, B) is an extensible trace-alphabet-pair of (l, t), and, for any p such that (p, B) is an extensible trace-alphabet-pair of (l, t), $p \leq_o q$.

Proof. Let (p, s) be an extensible pair of (l, t), and $B = \mathrm{alph}(s)$. It is clear that $(p, B) \in S_l$ and $p \leq_s r$ for a trace r such that $t = rs$. Let $q = \sqcup(\mathrm{Pre}(l) \cap \mathrm{Suf}(r))$, thus $p \leq_o q$, and further $(q, B) \in S_l$. This allows N_l the transitions $(\lambda, \emptyset) \xrightarrow{r}_{N_l} (q, \emptyset) \xrightarrow{s}_{N_l} (q, B)$, thus $(q, B) \in S_l(t)$ which proves the only-if-part of (i). Let now $(q, B) \in S_l(t)$. By Lemma 5.3 there are traces r, s such that $t = rs$, $q \leq_s r$, and $\mathrm{alph}(s) = B$. Thus qs is a suffix of t. Clearly, $p \leq_o q$ implies that ps is a suffix of t and p is a prefix of l. This proves the if-part of the first statement.

The second statement is corollary of the first one. Let first $(q, B) \in S_l(t)$. By the if-part of (i) (q, B) is an extensible trace-alphabet-pair of (l, t). By the only-if-part of (i), for any p such that (p, B) is an extensible trace-alphabet-pair of (l, t) there is a q' such that $p \leq_o q'$ and $(q', B) \in S_l(t)$. However by Lemma 5.5 all those q' are equal to q. Let now (q, B) be an extensible trace-alphabet-pair of (l, t) such that for any p such that (p, B) is an extensible trace-alphabet-pair of (l, t), $p \leq_o q$. By the only-if-part of (i) there is a q' such that $(q', B) \in S_l(t)$ and $q \leq_o q'$. However by the if-part of (i), (q', B) is an extensible trace-alphabet-pair of (l, t) which by the assumption implies $q' \leq_o q$, thus $q = q'$. □

This observation yields Theorem 6.4. It is obtained immediately from the following Lemma 6.5 which implies that $\mathcal{B}(l, t)$ is a lattice. Because $\mathcal{B}(l, t) \subseteq \mathcal{P}(A)$ it is clear that the lattice is distributive.

Theorem 6.4 *Let $l, t \in M(A,D)$. Then $(\mathcal{B}(l, t), \cap, \cup)$ is a distributive lattice.*

Lemma 6.5 *Let $l, t \in M(A,D)$. Let $(p_1, B_1), (p_2, B_2) \in S_l(t)$. Then*

$$(p_1 \sqcap_p p_2, B_1 \cap B_2), \text{ and } (p, B_1 \cup B_2) \text{ for some } p \geq_p p_1 \sqcup_p p_2$$

are elements of $S_l(t)$, too.

Proof. By Lemma 6.3 and Theorem 6.2 we deduce that if (p_1, B_1), and (p_2, B_2) are both elements of $S_l(t)$ then there are extensible trace-alphabet-pairs $(p_1 \sqcup_p p_2, B_1 \cup B_2)$ and $(p_1 \sqcap_p p_2, B_1 \cap B_2)$ of (l, t). Again by Lemma 6.3

there are $p, q \in \mathrm{Pre}(l)$ such that $p_1 \sqcup_p p_2 \leq_p p$, $p_1 \sqcap_p p_2 \leq_p q$, and $(p, B_1 \cup B_2), (q, B_1 \cap B_2) \in S_l(t)$. Clearly $(q, B_1 \cap B_2)$ is an extensible trace-alphabet-pair of (l, t). By Theorem 6.2 $(q \sqcup_p p_1, B_1)$ is an extensible trace-alphabet-pair of (l, t), too. By Lemma 6.3 $q \sqcup_p p_1 \leq_o p_1$ which implies $q \leq_p p_1$. Similarly one obtains $q \leq_o p_2$. Therefore $q = p_1 \sqcap_p p_2$ $\square$

It is an open question whether $(p_1, B_1), (p_2, B_2) \in S_l(t)$ implies $(p_1 \sqcup_p p_2, B_1 \cup B_2) \in S_l(t)$. Although the author does not believe that this is likely, he was not able to find a counterexample.

In the following we show that we can represent a set $S_l(t)$ by not more than $|A|$ of its elements. This greatly reduces the number of states reachable in the subset automaton of N_l. The result is obtained using Lemma 6.5 and the observations in [2], chapter III, § 3. We recall the definition of meet-irreducible elements in [2].

Definition 6.6 *Let $(L, \sqcap, \sqcup)$ be lattice. An element x of L, which is not a greatest element in L, is called meet-irreducible if $x = y \sqcap z$ implies $y = x$ or $z = x$ for all elements y and z of L.*

Let $l, t \in M(A,D)$. Let $\mathcal{M}(l, t)$ denote the set of meet-irreducible elements of $\mathcal{B}(l, t)$ together with the greatest element, i.e.,

$$\mathcal{M}(l, t) \;=\; \{B \in \mathcal{B}(l, t) \mid B_1 \cap B_2 \neq B \text{ for all } B_1, B_2 \in \mathcal{B}(l, t) \setminus \{B\}\}.$$

Let $\mathcal{S}_l(t)$ denote the corresponding elements in $S_l(t)$, i.e.,

$$\mathcal{S}_l(t) \;=\; \{(p, B) \in S_l(t) \mid B \in \mathcal{M}(l, t)\}.$$

One now obtains

Lemma 6.7 *Let $l, t \in M(A,D)$. Each element $(p, B) \in S_l(t)$ is the meet of the greater elements in $\mathcal{S}_l(t)$. I.e., let $\mathcal{S}_l(t)_{\geq B} = \{(q, E) \in \mathcal{S}_l(t) \mid B \subseteq E\}$, then*

$$(p, B) \;=\; \sqcap \mathcal{S}_l(t)_{\geq B},$$

where $(q_1, E_1) \sqcap (q_2, E_2)$ is defined by $(q_1 \sqcap_p q_2, E_1 \cap E_2)$.

Proof. Let $(p, B) \in S_l(t)$, let $\mathcal{M}(l, t)_{\geq B} = \{E \in \mathcal{M}(l, t) \mid B \subseteq E\}$. We first observe that $B = \cap \mathcal{M}(l, t)_{\geq B}$. Clearly, $B \subseteq \cap \mathcal{M}(l, t)_{\geq B}$, we show the opposite inclusion by induction. If $B \in \mathcal{M}(l, t)$ the statement is clear. If $B \notin \mathcal{M}(l, t)$ there are $B_1, B_2 \in \mathcal{B}(l, t)$ such that $B = B_1 \cap B_2$ and $B_2 \neq B \neq B_1$, i.e., $B_1, B_2 \supset B$. Assume $B_i = \cap \mathcal{M}(l, t)_{\geq B_i}$ for $i \in \{1, 2\}$. Then $B = (\cap \mathcal{M}(l, t)_{\geq B_1}) \cap (\cap \mathcal{M}(l, t)_{\geq B_2}) = \cap(\mathcal{M}(l, t)_{\geq B_1} \cup \mathcal{M}(l, t)_{\geq B_2}) \supseteq \cap \mathcal{M}(l, t)_{\geq B}$. This implies that the statement holds for the second component, i.e., there is a $q \in \mathrm{Pre}(l)$ such that $(q, B) = \sqcap \mathcal{S}_l(t)_{\geq B}$. By Lemma 6.5 $(q, B) \in S_l(t)$, by Lemma 5.5 $q = p$. $\square$

Lemma 6.7 implies that the set $S_l(t)$ is fully determined by the elements in $\mathcal{S}_l(t)$. From Lemma 2 in [2], chapter III, § 3 one deduces that a sublattice of $\mathcal{P}(A)$ has at most $|A| - 1$ meet-irreducible elements. Therefore there are at most $|A|$ elements in $\mathcal{M}(l, t)$ and thus in $\mathcal{S}_l(t)$.

Clearly, the elements in $\mathcal{S}_l(ta)$ can be computed (inefficiently) from the elements in $\mathcal{S}_l(t)$, (A,D), $a \in A$, and l by reconstructing all elements in $S_l(t)$ computing $S_l(ta)$ and finally determining the meet-irreducible elements in $S_l(ta)$. If one does not keep all elements in $S_l(t)$ simultaneously in the memory, and computes them only when needed, (and similar for the elements in $S_l(ta)$) this computation can be performed using only $\mathcal{O}(|A|^k|v|)$ space for some $k \geq 1$. Clearly, time complexity is not improved by this approach.

It is an open question whether $\mathcal{S}_l(ta)$ can be computed efficiently from $\mathcal{S}_l(t)$, (A,D), $a \in A$, and $l \in \mathrm{M}(A,D)$. A positive answer immediately yields an $\mathcal{O}(|A|^k|v|^{k'}|w|)$-time algorithm for the factor problem for some k, k'. To obtain an $\mathcal{O}(|A|^k|vw|)$-time algorithm for the factor problem, the computation should only use amortized constant time in the length of l (i.e., time $\mathcal{O}(|A|^k)$).

7 Conclusion

We have shown that the pattern matching problem in trace monoids is solvable in linear time using an approach where a finite automaton which is determined by the searched pattern is simulated on the trace where the pattern is searched in. This approach has the advantage that the search-space, i.e. the trace where the pattern is searched in, can be read as a stream of symbols, and has not to be stored in memory. One step of this finite automaton can be simulated in amortized constant time, where space linear in the size of the pattern is needed to represent the actual state. However, this is only true when considering a fixed trace monoid. If the dependence alphabet (A,D) is a part of the input, the time and the space complexity of the presented algorithm is exponential in the size of the dependence alphabet (A,D). This complexity is clearly not desirable when considering parallel systems with many different actions. As we discussed, a polynomial space complexity of the algorithm can be achieved; each state of the automaton can be represented by a structure of size polynomial in the size of the dependence alphabet and the pattern. It is not clear how this observation can be used to obtain an efficient simulation of the automaton. However, we conjecture that this is possible. An even better result, which one could expect, would be an efficient simulation of the automaton using amortized time depending only on the size of the alphabet and not on the size of the pattern. Clearly, it is an open question whether this is possible.

8 Acknowledgments

The author wishes to thank V. Diekert, A. Muscholl, and J. Torán for their advice when preparing the paper, and K. Hashiguchi for a discussion about the problem.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading, Mass., 1974.

[2] Garrett Birkhof. *Lattice Theory*, volume XXV of *Colloquium Publications.* American Mathematical Society, Providence, Rhode Island, 3. edition, 1967.

[3] Pierre Cartier and Dominique Foata. *Problèmes combinatoires de commutation et réarrangements.* Number 85 in Lecture Notes in Mathematics. Springer, Berlin-Heidelberg-New York, 1969.

[4] Robert Cori and Yves Métivier. Recognizable subsets of some partially abelian monoids. *Theoretical Computer Science*, 35:179–189, 1985.

[5] Robert Cori, Yves Métivier, and Wiesław Zielonka. Asynchronous mappings and asynchronous cellular automata. *Information and Computation*, 106:159–202, 1993.

[6] Robert Cori and Dominique Perrin. Automates et commutations partielles. *R.A.I.R.O. — Informatique Théorique et Applications*, 19:21–32, 1985.

[7] Volker Diekert. *Combinatorics on Traces.* Number 454 in Lecture Notes in Computer Science. Springer, Berlin-Heidelberg-New York, 1990.

[8] Volker Diekert and Yves Métivier. Partial commutation and traces. In G. Rozenberg and A. Salomaa, editors, *Handbook on Formal Languages*, volume III. Springer, Berlin-Heidelberg-New York. To appear.

[9] Volker Diekert and Grzegorz Rozenberg, editors. *The Book of Traces.* World Scientific, Singapore, 1995.

[10] Christine Duboc. On some equations in free partially commutative monoids. *Theoretical Computer Science*, 46:159–174, 1986.

[11] Samuel Eilenberg. *Automata, Languages, and Machines*, volume A. Academic Press, New York and London, 1974.

[12] Paul Gastin and Brigitte Rozoy. The poset of infinitary traces. *Theoretical Computer Science*, 120:101–121, 1993.

[13] Kosaburo Hashiguchi and Kazuya Yamada. String matching problems over free partially commutative monoids. *Information and Computation*, 101:131–149, 1992.

[14] Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–350, 1977.

[15] Hai-Ning Liu, Celia Wrathall, and Kenneth Zeger. Efficient solution of some problems in free partially commutative monoids. *Information and Computation*, 89:180–198, 1990.

[16] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Rep. PB 78, Aarhus University, Aarhus, 1977.

[17] Jochen Messner. Pattern matching in trace monoids. In R. Reischuk, editor, *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science 1997*, Lecture Notes in Computer Science, Berlin-Heidelberg-New York, 1997. Springer. To appear.

[18] Friedrich Otto and Celia Wrathall. Overlaps in free partially commutative monoids. *Journal of Computer and System Sciences*, 42:186–198, 1991.

[19] Karl Rüdiger Reischuk. *Einführung in die Komplexitätstheorie.* Leitfäden und Monographien der Informatik. Teubner, Stuttgart, 1990.