Vectorial Languages and Linear Temporal Logic

Olivier Serre

VECTORIAL LANGUAGES AND LINEAR TEMPORAL LOGIC

Olivier Serre
LIAFA, Université Paris VII
2, place Jussieu, case 7014
F-75251 Paris Cedex 05
serre@liafa.jussieu.fr

Abstract

Determining for a given deterministic complete automaton the sequence of visited states while reading a given word is the core of important problems with automata-based solutions, such as approximate string matching. The main difficulty is to do this computation efficiently, especially when dealing with very large texts. Considering words as vectors and working on them using vectorial (parallel) operations allows to solve the problem faster than in linear time using sequential computations.

In this paper, we show first that the set of vectorial operations needed by an algorithm representing a given automaton depends only on the language accepted by the automaton. We give precise characterizations of vectorial algorithms for star-free, solvable and regular languages in terms of the vectorial operations allowed. We also consider classes of languages associated with restricted sets of vectorial operations and relate them with languages defined by fragments of linear temporal logic.

Finally, we consider the converse problem of constructing an automaton from a given vectorial algorithm. As a byproduct, we show that the satisfiability problem for some extensions of linear-time temporal logic characterizing solvable and regular languages is PSPACE-complete.

Keywords: Parallel automata simulation, linear temporal logic and extensions

Introduction

Given a deterministic complete automaton and an input word, a classical question is to decide whether or not the automaton accepts the word. A more detailed information is the sequence of visited states while processing the word. Computing this sequence is the core of important problems such as approximate string matching [8]. An elegant way to solve this problem consists in simulating a dynamic programming approach by a finite, deterministic automaton [2] on

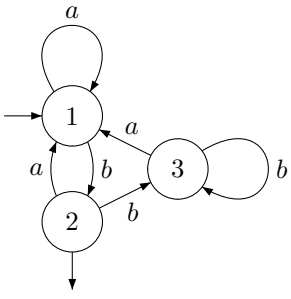
the input sequence. However, approximate string matching is generally used on very long sequences (as genomic ones) and the natural algorithm, which is linear in the length of the input word, is not performing enough. A natural solution to accelerate the computation is to consider words as vectors and therefore to compute the sequence of visited states using vectorial operations, that can be efficiently achieved using parallelism [8].

In this paper, we are interested in vectorial algorithms, that were introduced and investigated by A. Bergeron and S. Hamel in [2, 3]. Such an algorithm computes the sequence of visited states while reading a word using a *constant* number (independent of the length of the word) of vectorial operations. The existence of an algorithm for a given automaton depends on the automaton and on the kind of vectorial operations we allow. The problem can also be studied from the language point of view: can we find a deterministic complete automaton recognizing a given language and an associated vectorial algorithm? We first show that the existence of a vectorial algorithm depends on the languages only. Then we exhibit a very tight connection between temporal logic operators and vectorial operations. This relation will motivate the notion of PTL-vectorial algorithm, where PTL stands for Past Temporal Logic. Moreover, we obtain an alternative proof of the equivalence between star-free languages and vectorial algorithms (Note that the inclusion of star-free languages in the class of PTL-vectorial languages was established in [3]. Hence, we show here that the converse also holds). Then, we describe extensions of vectorial algorithms, first to capture a larger subclass of regular languages, the solvable ones, and finally for the whole class of regular languages.

Finally, we consider the converse problem, that is we want to check for a given vectorial algorithm whether there exists an automaton associated with it. To solve this problem, we show how to decide the satisfiability of formulas belonging to extensions of linear temporal logic introduced in [1]. Our constructions are based on alternating automata. Proofs omitted in the paper can be found in [13].

1. Notation and Definitions

1.1. Vectorial Algorithms



Throughout the paper, vectors are noted in bold characters (e.g. $\mathbf{u}$) and are considered as words. Conversely, vectorial operations can be applied to words, considering them as vectors. Therefore, a word u is associated with a canonical vectorial representation $\mathbf{u}$ and a vector $\mathbf{v}$ is associated with a canonical word representation v.

Let $\mathcal{A} = (Q, A, \cdot, q_0, F)$ be a deterministic complete automaton, where A is a finite alphabet, Q the finite set of states of $\mathcal{A}$, q_0 the initial state, F the set of finite states and $\cdot$ the transition function $Q \times A \to Q$ of $\mathcal{A}$.

With each input vector $\mathbf{u} = a_1 a_2 \cdots a_m \in A^*$ we associate the output vector $\mathbf{r} = r_1 r_2 \cdots r_m \in Q^*$ representing the sequence of states reached reading $\mathbf{u}$ (we omit the leading initial state). Therefore, $\mathbf{u}$ and $\mathbf{r}$ have the same length. For instance, consider the automaton given in figure below. With the input vector $\mathbf{u} = bbaabbbababab$ we associate the output vector $\mathbf{r} = 2311233121212$. A *vectorial algorithm* for $\mathcal{A}$ consists of a sequence of vectorial operations of constant length (i.e., a straight-line expression of length which is independent on $\mathbf{u}$) computing $\mathbf{r}$ from $\mathbf{u}$.

Given a word $\mathbf{u} = a_1 a_2 \cdots a_m$, we consider for every letter $a \in A$, the boolean vector $(\mathbf{u} = \mathbf{a}) = b_1 \cdots b_m$ where, for each i, $b_i = 1$ if $a_i = a$ an $b_i = 0$ otherwise. Hence, $(\mathbf{u} = \mathbf{a})$ is the characteristic boolean vector of the letter a in the word $\mathbf{u}$. Just as for words, for any state $q \in Q$, with an output vector $\mathbf{r} = r_1 r_2 \cdots r_m$ we associate the boolean vector $(\mathbf{r} = \mathbf{q}) = (r_1 = q) \cdots (r_m = q)$ that is, the characteristic vector of state q. For example, $(bbaabbbababab = \mathbf{a}) = 0011000101010$ and $(2311233121212 = \mathbf{1}) = 0011000101010$.

The sequence $(\mathbf{u} = \mathbf{a})_{a \in A}$ (respectively, the sequence $(\mathbf{r} = \mathbf{q})_{q \in Q}$) is an equivalent boolean representation for the input word $\mathbf{u}$ (respectively for the output vector $\mathbf{r}$). In order to work only with boolean vectors, the vectorial algorithms presented in this paper compute the sequence of characteristic vectors $(\mathbf{r} = \mathbf{q})_{q \in Q}$ from the sequence of characteristic vectors $(\mathbf{u} = \mathbf{a})_{a \in A}$.

Let Ω be a class of vectorial operations. Vector algorithms based on operations from Ω and on the bit-wise logical operations (combinations of $\vee$, $\wedge$ and $\neg$) are called Ω-*vectorial algorithms*. A deterministic complete automaton is called Ω-*vectorial* if there is an Ω-vectorial algorithm computing for every $\mathbf{u} \in A^*$ the sequence $(\mathbf{r} = \mathbf{q})_{q \in Q}$ from the sequence $(\mathbf{u} = \mathbf{a})_{a \in A}$. Finally, a language is Ω-*vectorial* if it is recognized by a deterministic complete Ω-vectorial automaton.

Given a class Ω of vectorial operations, we write $\mathrm{VA}[\Omega]$ for the set of Ω-vectorial algorithms and $\mathrm{V}\mathcal{L}[\Omega]$ for the set of Ω-vectorial languages.

Proposition 1 below shows that minimization preserves the property of being an Ω-vectorial automaton. Therefore a language is Ω-vectorial if and only if its minimal automaton is Ω-vectorial (by minimal automaton we always mean the minimal *complete* automaton). This property is very useful, because to decide whether or not a language is Ω-vectorial, it suffices to know how to decide whether or not a given automaton (the minimal one) is Ω-vectorial.

Proposition 1 *The property of being Ω-vectorial is preserve by automaton homomorphisms. Therefore if a deterministic complete automaton $\mathcal{A}$ is Ω-vectorial, then its minimal automaton $\mathcal{A}_{min}$ is also Ω-vectorial.*

1.2. Past Temporal Logic

Given a class Ω of vectorial operators our aim is to give a characterization of Ω-vectorial languages that allows us to decide whether or not a given language is Ω-vectorial. For this, we use characterizations in terms of past temporal logic PTL [6]. We first recall the syntax of PTL:

PTL-formulas are constructed inductively according to the following rules:

(1) For every $a \in A$, p_a is a PTL-formula.

(2) If φ_1 and φ_2 are PTL-formulas, so are $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, $\mathbf{Y}\varphi_1$ and $\varphi_1 \,\mathbf{S}\, \varphi_2$.

Semantics is defined by induction on the rules. Given a word $w \in A^+$ and an integer $n \in \{1, 2, \ldots, |w|\}$, we define that "$w$ satisfies φ at position n", denoted $(w, n) \models \varphi$, as follows:

(1) $(w, n) \models p_a$ if the nth letter of w is a.

(2) $(w, n) \models \varphi_1 \vee \varphi_2$ if $(w, n) \models \varphi_1$ or $(w, n) \models \varphi_2$.

(3) $(w, n) \models \neg\varphi_1$ if $(w, n) \not\models \varphi_1$.

(4) $(w, n) \models \mathbf{Y}\varphi_1$ if $n > 1$ and $(w, n-1) \models \varphi_1$.

(5) $(w, n) \models \varphi_1 \,\mathbf{S}\, \varphi_2$ if there exists $m \leq n$ such that $(w, m) \models \varphi_2$ and, for every k such that $m < k \leq n$, $(w, k) \models \varphi_1$.

With each PTL-formula φ, we associate the language L_φ of finite words satisfying φ as $L_\varphi = \{u \in A^+ \mid (u, |u|) \models \varphi\}$. Given a class Λ of logical operators we write TL[Λ] for the set of temporal formulas in which modalities other than ones from Λ do not occur. A language $L \subseteq A^*$ is Λ-definable if there exists a formula $\varphi \in \mathrm{TL}[\Lambda]$ such that $L = L_\varphi$. The set of Λ-definable languages will be denoted by $\mathcal{L}[\Lambda]$.

1.3. Vectorial Languages and Temporal Logic

We will show that there exists a tight link between languages defined by logical conditions and languages defined by "equivalent" vectorial conditions. But, whereas logical satisfiability depends exclusively on the language, vectorial characterizations seem to be closely related with a specific automaton. Vectorial characterizations are stronger than logical characterizations because in order to have a vectorial algorithm for a given automaton one must be able to characterize *any* state, hence any language recognized by the automaton obtained by setting a given state as unique final state. For a logical formula one just needs to exhibit the set of final states needed for the given language.

But under some assumptions, logical fragments and vectorial fragments define the same class of languages. Let us be more explicit. Given a set Ω of vectorial operations and a set Λ of logical operators, we will say that Ω and Λ are *equivalent* if they verify the following conditions:

(1) To any vectorial algorithm Φ using only operations in Ω one can associate a formula φ using only operators in Λ such that for any word u and any positive integer i smaller than $|u|$, the i-th entry of the vector obtained by applying Φ to u is 1 if and only if $(u, i) \models \varphi$.

(2) To any formula φ using only operators in Λ, one can associate a vectorial algorithm Φ using only operators in Ω such that for any word $u = u_1 \cdots u_m$, the computation of the binary vector $\mathbf{v}_\varphi = v_1 \cdots v_m$, where $v_i = 1 \Leftrightarrow (u_1 \cdots u_i, i) \models \varphi$, is performed by the algorithm Φ.

Several fragments [4, 19, 20] and extensions [1] of temporal logic have been studied. Therefore, in order to characterize for a given class Ω of vectorial operations the class of Ω-vectorial languages, a solution consists in finding an equivalent fragment in temporal logic. We have to find a condition on two equivalent sets Ω and Λ to have $\mathrm{V}\mathcal{L}[\Omega] = \mathcal{L}[\Lambda]$.

A class Λ of logical operators will be called *finally stable* if for every language L that belongs to $\mathcal{L}(\Lambda)$, any language recognized by an automaton obtained from the minimal automaton of L by letting some arbitrary state to be the unique final state, belongs to $\mathcal{L}(\Lambda)$.

For example, any set Λ such that $\mathcal{L}(\Lambda)$ is a variety of languages (see [9, 10] for the definition of variety) of languages is finally stable. Formally, if L is a language in $\mathcal{L}(\Lambda)$ and $\mathcal{A}$ its minimal automaton, any automaton $\mathcal{A}'$ obtained by modifying the final states of $\mathcal{A}$ recognizes a language L' of $\mathcal{L}(\Lambda)$ because the syntactic monoid of $\mathcal{A}'$ divides the syntactic monoid of $\mathcal{A}$.

The notion of final stability gives us the following lemma:

Lemma 1 *Let Ω be a class of vectorial operations and let Λ be an equivalent class of logical operators. Then Λ is finally stable if and only if $\mathrm{V}\mathcal{L}(\Omega) = \mathcal{L}(\Lambda)$.*

2. A Vectorial Characterization of Star-Free Languages

To make our algorithms precise we have to state which vectorial operations are allowed. As in [3], we first consider a basic class of vectorial operations:

- Bit-wise logical operations such as $\vee$, $\wedge$, $\neg$ and the atomic formulas $(\mathbf{u} = \mathbf{a}) = (u_1 = a) \cdots (u_m = a)$ for each $a \in A$.
- Right shift: $\uparrow_i u_1 \cdots u_m = i u_1 \cdots u_{m-1}$, $i \in \{0, 1\}$.
- Binary addition between two vectors of same length: we perform the usual binary addition from left to right but we do not keep the highest bit (carry) if the length of the result exceeds the initial vectors' ones. For example $\uparrow_0 110101 + 101011 = 110100$.

Vectorial algorithms using only these operations are called in this paper *PTL-vectorial algorithms*. Recall that PTL stands for Past Temporal Logic, as we show that PTL-vectorial operations and PTL operators are equivalent:

Theorem 1 *The class $\{\uparrow_0, +\}$ of vectorial operation is equivalent to the class $\{\mathbf{Y}, \mathbf{S}\}$ of logical operators.*

The main point of the proof is to use the operator $\mathbf{S}$ in order to express the carry bit, and thus addition of vectors.

Corollary 1 *A regular language is PTL-vectorial if and only if it is star-free.*

3. Beyond Star-Freeness

3.1. Solvable Languages

An alternative proof for PTL-vectorial languages being star-free, is based on the following result:

Proposition 2 *Let* $\mathbf{u}$ *be a vector and let* Φ *be a PTL-vectorial algorithm. If* $\mathbf{u}$ *has period* $p \in \mathbb{N}$ *then the result* $\Phi(\mathbf{u})$ *of* Φ *applied on* $\mathbf{u}$ *has period* p.

Hence, to provide characterizations for families of regular languages that strictly contain the star-free languages we need to introduce vectorial operations, that do not preserve the period. For any integers k, l such that $0 \leq l < k$, we define the *modular operation* $S_{l,k}$ by

$$S_{l,k}(x_1 x_2 \cdots x_m) = (s_1 \cdots s_m) \text{ where } s_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{i} x_j = l \pmod{k}, \\ 0 & \text{otherwise.} \end{cases}$$

Vectorial algorithms using only the PTL-vectorial operations plus the modular operations $S_{l,k}$ will be called *MTL-vectorial algorithms*. A language is an *MTL-vectorial language* if there is an MTL-vectorial algorithm for its minimal automaton.

Consider now the following extension of temporal logic, called modular temporal logic (MTL) [1]: by modular temporal logic we mean past temporal logic augmented with the unary operators $\text{Mod}_{l,k}$ for integers $0 \leq l < k$. The new modular operators have the following natural semantics: given an MTL formula φ, we have $(u, i) \models \text{Mod}_{l,k}\varphi$ if, there are l positions j, $1 \leq j \leq i$ (modulo k) such that $(u, j) \models \varphi$.

It was shown in [1], that MTL characterizes the variety of solvable languages. A regular language is called *solvable* if its syntactic monoid contains no non solvable group [15].

Theorem 2 ([1]) *A regular language is MTL-definable if and only if it is solvable.*

Therefore we have the following result:

Theorem 3 *The class* $\{\uparrow_0, +\} \cup \{S_{l,k} |\ 0 \leq l \leq k,\ k \in \mathbb{N}\}$ *of vectorial operations is equivalent to the finally stable class* $\{\mathbf{Y}, \mathbf{S}\} \cup \{\text{Mod}_{l,k},\ 0 \leq l \leq k\}$ *of logical operators.*

We easily obtain now a vectorial characterization of solvable languages:

Corollary 2 *A regular language is MTL-vectorial if and only if it is solvable.*

3.2. Regular Languages

MTL-vectorial algorithms do not characterize all regular languages. Therefore, we propose an extension to MTL-vectorial algorithms, which we denote as *GTL-vectorial algorithms*, which captures all regular languages.

For this we consider an extension of modular temporal logic introduced in [1] and construct an equivalent class of vectorial operations. This extension of temporal logic is obtained by augmenting modular temporal logic with group temporal operators $\Gamma_{g,G}$ for any finite group G and any element $g \in G$. The operator $\Gamma_{g,G}$ always binds $|G| - 1$ formulas.

Let us now explain the semantics of $\Gamma_{g,G}$ for a given finite group G and an element $g \in G$. We first have to order the elements of the group G (this order will not be modified afterwards), say as $g_1, g_2, \ldots, g_q = id$. Let u be an element of A^+ and let $\varphi_1, \varphi_2, \ldots, \varphi_{q-1}$ be GTL-formulas. With each j, $1 \leq j \leq |u|$ we associate an element of G, denoted $\langle \varphi_1, \varphi_2 \ldots \varphi_{q-1} \rangle \langle u, j \rangle$, defined by:

$$\langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g_k$$

where $k = \min\{l \mid (u, j) \models \varphi_l\}$ with the convention that $\min \emptyset = q$.

Finally, we define $(u, i) \models \Gamma_{g,G} \langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle$ to mean that:

$$\prod_{j=1}^{i} \langle \varphi_1, \varphi_2, \ldots, \varphi_{q-1} \rangle \langle u, j \rangle = g$$

It was shown in [1] that a language is expressible in group temporal logic if and only if it is regular:

Theorem 4 ([1]) *A language is GTL-definable if and only if it is regular.*

Thus, to have a vectorial characterization of all regular languages it suffices to find vectorial operations equivalent to $\Gamma_{g,G}$ for all finite groups G. Let $G = \{g_1, g_2, \ldots, g_q = id\}$ be a finite group of cardinality q and let $\mathbf{v}_{g_1}, \mathbf{v}_{g_2}, \ldots, \mathbf{v}_{g_{q-1}}$ be $q-1$ vectors of same length m. With each position j, $1 \leq j \leq m$ we associate an element of G, denoted $\mathcal{G}(\langle \mathbf{v}_{g_1}, \mathbf{v}_{g_2}, \ldots, \mathbf{v}_{g_{q-1}} \rangle, G, j)$ and defined by:

$$\mathcal{G}(\langle \mathbf{v}_{g_1}, \mathbf{v}_{g_2}, \ldots, \mathbf{v}_{g_{q-1}} \rangle, G, j) = g_k$$

where $k = \min\{l \mid \mathbf{v}^j_{g_l} = 1\}$ ($\mathbf{v}^j_{g_l}$ representing the j-th bit of vector $\mathbf{v}_{g_l}$) with the convention that $\min \emptyset = q$.

Finally, we introduce the vectorial operator $P_{g,G}$ defined by:

$$P_{g,G}(\mathbf{v}_{g_1}, \mathbf{v}_{g_2}, \ldots, \mathbf{v}_{g_{q-1}}) = (s_1 \ldots s_m)$$

$$\text{with } s_i = \begin{cases} 1 & \text{if } \displaystyle\prod_{j=1}^{i} \mathcal{G}(\langle \mathbf{v}_{g_1}, \mathbf{v}_{g_2}, \ldots, \mathbf{v}_{g_{q-1}} \rangle, G, j) = g, \\ 0 & \text{otherwise.} \end{cases}$$

Remark 1 One can note that the modular operations are special cases of group operations that only use cyclic groups $(\mathbb{Z}/k\mathbb{Z}, +)$, as we have $S_{l,k}(\mathbf{x}) = \Gamma_{l,(\mathbb{Z}/k\mathbb{Z},+)}(\mathbf{x}, \mathbf{0}, \ldots, \mathbf{0})$, where the elements of $(\mathbb{Z}/k\mathbb{Z}, +)$ are ordered as $(1, 2, \ldots, k)$.

By construction we have the following result:

Theorem 5 *The class* $\{\uparrow_0, +\} \cup \{P_{g,G} \mid g \in G\}$ *of vectorial operations is equivalent to the finally stable class* $\{\mathbf{Y}, \mathbf{S}\} \cup \{\Gamma_{g,G} \mid g \in G\}$ *of logical operators.*

Therefore, we obtain a vectorial characterization for regular languages:

Corollary 3 *A language is GTL-vectorial if and only if it is regular.*

4. Reconstructing an Automaton From a Vectorial Algorithm

In the preceding sections we determined a vectorial algorithm from a given automaton. We now consider the converse problem, that is we want to check for a given vectorial algorithm whether there exists a deterministic complete automaton associated with it (and determine an automaton, if this is the case). This question becomes interesting for instance when we modify a given vectorial algorithm (associated with a deterministic automaton) and we want to check afterwards that the new algorithm is equivalent to the old one. We will show that the complexity of this test is actually the same as testing the satisfiability of a GTL-formula i.e. PSPACE-complete.

Vectorial algorithms are associated with deterministic complete automata and therefore depend on the initial state, and not only on the underlying labeled graph structure of the given automaton. We will thus suppose that the initial state is part of the input.

To begin with, let us call a *valid* vectorial algorithm an algorithm for which there exists a corresponding deterministic complete automaton. Let us explain how to construct such an associated automaton. Let $A = \{a_1, \ldots, a_k\}$ be the alphabet of the automaton and let n be the number of states. We will identify the states with the integers $1 \ldots n$. To compute an associated automaton $\mathcal{A}_\Phi$ from a given vectorial algorithm Φ we perform a depth-first search of $\mathcal{A}_\Phi$, that is we start from the initial state q_0 and compute the states that can be reached by reading a letter from q_0 and then we repeat this step with the new states found so far. We are done when we have explored all reachable states. With this method we explore all the transitions of the accessible part of the automaton. We just have to explain how to compute the reachable states from a given state. In our algorithm we maintain a vector, *state_direction*, giving for any encountered state q a word u leading from the initial state to q. Therefore, when considering a state q, and a letter a to compute the transition from q reading a we have to apply Φ to the word ua and consider the $|u| + 1$ component of the result, denoted $\Phi^{|u|+1}(ua)$.

The algorithm for constructing $\mathcal{A}_\Phi$ is the following one:

Variables and initialization:
δ: $(n \times k)$-vector.
$new\_states = [1]$: last-in-first-out structure.
$known\_states = \{1\}$: Set structure.
$state\_direction = [\varepsilon, \varepsilon, \ldots, \varepsilon]$: n-vector.

Main loop:
While $new\_states \neq \emptyset$ **Do**
Let q =Delete element from new_states.
Let $u = state\_direction.(q)$.
Let $h = |u|$.
For $i = 1$ to k **Do**
Let $q' = \Phi^{h+1}(ua_i)$.
Let $\delta(q, i) = q'$.
If $q' \notin known\_states$ **Then**
Add q' to new_states and to $known\_states$.
Set $state\_direction.(q') = ua$.
End If.
End For
End While

Return δ

To test the validity of a given algorithm Φ we will first use the preceding algorithm to compute the automaton $\mathcal{A}_\Phi$ associated with Φ, if it is valid. If the algorithm does not work (that is if $\Phi^{h+1}(ua_i)$ is not defined for a given step of the algorithm) this implies that Φ is not valid. Otherwise we test the validity as stated in the theorem below. For any state q, let $L(q)$ denote the regular language defined by the logical formula obtained by translating the algorithm computing $(\mathbf{r} = \mathbf{q})$.

Theorem 6 *Let Φ be a PTL-vectorial algorithm and let $\mathcal{A}_\Phi$ be the deterministic complete automaton constructed by the algorithm above. Then Φ is a valid algorithm associated with $\mathcal{A}_\Phi$ if and only if:*

(1) *For any non reachable state q of $\mathcal{A}_\Phi$, we have $L(q) = \emptyset$.*
(2) *For any reachable state q, we have that $L(q) \neq \emptyset$. In addition, the following assertions are equivalent:*
   (i) *$L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$, where $E_q = \{\varepsilon\}$ if q is the initial state and $E_q = \emptyset$ otherwise. Moreover, a_j is a letter and each q_j is a reachable state.*
   (ii) *$\{(q_1, a_1), \ldots, (q_i, a_i)\}$ is exactly the set of the pairs (q_j, a_j) such that $q_j.a_j = q$ in $\mathcal{A}_\Phi$.*

We can now give a method to test the validity of a vectorial algorithm Φ:

(1) We apply the depth-first search algorithm described above to Φ. If the algorithm does not yield a deterministic automaton $\mathcal{A}_\Phi$, then Φ is not a valid algorithm and we can stop. Otherwise we go to the next step.

(2) We determine the reachable states and the non reachable states of the automaton $\mathcal{A}_\Phi$ constructed in the preceding step.

(3) For every non reachable state q we translate the associated component in Φ into a formula φ_q and test whether or not it can be satisfied. If φ_q is satisfiable for a non reachable-state q then Φ is not valid and we stop. Otherwise we go to the next step.

(4) For every reachable state q we determine the set $\{(q_1, a_1), \ldots, (q_i, a_i)\}$ of the pairs (q_j, a_j) such that $q_j \cdot a_j = q$ in $\mathcal{A}_\Phi$ and we verify that $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$. To achieve this efficiently we can determine for every j a formula associated with $L(q_j)a_j$. It suffices to consider the formula $p_{a_j} \wedge \mathbf{Y}\varphi_{q_j}$ where φ_{q_j} is the translation of the component of Φ associated with q_j. Then, we can construct a PTL-formula for the language $L(q)\Delta[L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q]$, where Δ holds for the symmetric difference, and verify that it cannot be satisfied, what is equivalent to the equality $L(q) = L(q_1)a_1 \cup \cdots \cup L(q_i)a_i \cup E_q$. If the test does not fail, then Φ is valid and associated with $\mathcal{A}_\Phi$, otherwise Φ is not valid.

Let us now give the complexity of this algorithm. The most costly part of the algorithm is the satisfiability problems for GTL-formulas that appear in step (3) and (4) of the algorithm. This problem is known to be PSPACE-complete [18] for PTL-formulas. In fact this result can be extended to GTL:

Theorem 7 *Deciding whether an GTL-formula is satisfiable is a PSPACE-complete problem.*

Using this result we conclude that our algorithm works in polynomial space. In fact we can give a more precise result:

Theorem 8 *Deciding whether or not a GTL-vectorial algorithm is valid is a PSPACE-complete problem.*

Note that the same problem applied to some fragments of PTL-vectorial algorithms become simpler. For instance we have the following result for algorithms in $\mathrm{VA}(\uparrow_0)$:

Theorem 9 *Deciding whether or not an algorithm in* $\mathrm{VA}(\uparrow_0)$ *is valid is an NP-complete problem.*

5. Conclusion

Using vectorial algorithms we have given new characterizations of star-free languages (as the class of PTL-vectorial languages), of solvable languages (as the class of MTL-vectorial languages) and of regular languages (as the class of GTL-vectorial languages). However, even in the easiest case, that is for star-free languages, there is no general efficient method to compute an algorithm associated with a given language. Nevertheless, since vectorial languages are closely related with temporal logic this is not that surprising at all, as the computation of an algorithm associated with an automaton is at least as difficult as finding a temporal logic formula associated with a given language, which might be of exponential size with regard to the automaton.

We have characterized subsets of vectorial operations by equivalent sets of temporal logic operators.

It is interesting to note that vectorial algorithms provide a more detailed information about an automaton than logical formulas without any loss in computational complexity and in the complexity of the operators used in both models.

Finally, we have shown that deciding the validity of a GTL-vectorial algorithm is PSPACE-complete. As a byproduct we have obtained that the extension of LTL with group operators does not change the complexity of the satisfiability problem, which is still PSPACE-complete, and we have given an effective algorithm deciding this question.

Another interesting investigation is to study fragments of algorithms based on the set of PTL-vectorial operations. Here, we want to know which subset of star-free languages can be characterized by forbidding certain vector operations. One can show that these fragments are closely related with the fragments of past temporal logic as defined and characterized in [4, 19, 20].

Acknowledgments

I gratefully acknowledge the many helpful suggestions of Anca Muscholl during the preparation of the paper. I also wish to express my thanks to Jean-Eric Pin for suggesting many stimulating ideas, and to the anonymous referees for their remarks.

References

[1] A. Baziramwabo, P. McKenzie, and D. Therien. Modular temporal logic. In *14th Symposium on Logic in Computer Science (LICS'99)*, pages 344–351. IEEE, 1999.

[2] A. Bergeron and S. Hamel. Cascade decomposition are bit-vector algorithms. In *Proceedings of the Conference CIAA'01*, Lecture Notes in Computer Science. Springer Verlag, 2001, to appear. http://www.lacim.uqam.ca/~anne.

[3] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. *International Journal of Foundations of Computer Science*, 13(1):53–66, 2002.

[4] J. Cohen, D. Perrin, and J.-E. Pin. On the expressive power of temporal logic for finite words. *Journal of Computer and System Sciences*, 46:271–294, 1993.

[5] J.A. Kamp. *Tense Logic and the Theory of Linear Order*. Ph.d. thesis, University of California, Los Angeles, 1968.

[6] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Proceedings of the Conference on Logics of Programs (LICS'85)*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218. Springer Verlag, 1985.

[7] R. McNaughton and S. Papert. *Counter-free Automata*. MIT Press, 1971.

[8] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the Association of Computing Machinery*, 46-3:395–415, 1999.

[9] J.-E. Pin. *Varieties of formal languages*. North Oxford, LondonPlenum, New-York, 1986. (Translation of Variétés de langages formels).

[10] J.-E. Pin. Syntactic semigroups. In G. Rozenberg and A. Salomaa, editors, *Handbook of formal languages*, volume 1, chapter 10, pages 679–746. Springer Verlag, 1997.

[11] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium Foundations of Computer Science (FOCS 1977)*, pages 46–57, 1977.

[12] M.P. Schützenberger and D. Perrin. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, 1965.

[13] O. Serre. Vectorial languages an linear temporal logic: Version with proofs. http://www.liafa.jussieu.fr/~serre.

[14] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.

[15] H. Straubing. Families or recognizable sets corresponding to certain varieties of finite monoids. *Journal of Pure and Applied Algebra*, 15:305–318, 1979.

[16] H. Straubing. *Finite automata, formal logic, and circuit complexity*. Birkhäuser, 1994.

[17] D. Thérien. Classification of finite monoids: the language approach. *Theoretical Computer Science*, 14:195–208, 1981.

[18] M. Y. Vardi. An automata-theoretic approach to linear-temporal logic. In F. Moller and G. Birtwistle, editors, *Logics for concurrency*, number 1043 in Lecture Notes in Computer Science, pages 238–266. Springer, 1996.

[19] Th. Wilke. *Classifying discrete temporal properties.* Habilitation thesis, Kiel, Germany, 1998.

[20] Th. Wilke. Classifying discrete temporal properties. In *STACS 99*, number 1563 in Lecture Notes in Computer Science, pages 32–46, Berlin, 1999. Springer.