Efficient Online Abelian Pattern Matching in Strings by Simulating Reactive Multi-Automata

Domenico Cantone and Simone Faro

Università di Catania, Dipartimento di Matematica e Informatica
Viale Andrea Doria 6, I-95125 Catania, Italy
{cantone,faro}@dmi.unict.it

Abstract. The *abelian pattern matching problem* consists in finding all substrings of a text which are permutations of a given pattern. This problem finds application in many areas and can be solved in linear time by a naïve sliding window approach. In this paper we introduce a new approach to the problem which makes use of a reactive multi-automaton modeled after the pattern, and provides an efficient nonstandard simulation of the automaton based on bit-parallelism.

Keywords: string permutations, nonstandard pattern matching, combinatorial algorithms on words, bit-parallelism, reactive multi-automata

1 Introduction

Given a pattern p and a text t, the *abelian pattern matching* problem [11] (also known as *jumbled matching* [8,7]) consists in finding all substrings of the text t, whose characters have the same multiplicities as in p, so that they could be converted into the input pattern just by permuting their characters.

It is a special case of the *approximate string matching* problem and naturally finds applications in many areas, such as string alignment [4], SNP discovery [5], and also in the interpretation of mass spectrometry data [6].

In the field of text processing and in computational biology, algorithms for abelian pattern matching are used as a filtering technique [3], usually referred to as *counting filter*, to speed up complex combinatorial searching problems. For instance, the counting filter technique has been used in the solution to the k-mismatches [15] and k-differences [17] problems. More recently, it has also been used in a solution to the approximate string matching problem allowing for inversions [9] and translocations [14]. A detailed analysis of the abelian pattern matching problem and of its solutions is presented in [11].

In this paper we are interested in the *online* version of the problem, whose worst-case time complexity is well known to be $\mathcal{O}(n)$, which assumes that the input pattern and text are given together for a single instant query, so that no preprocessing is possible.

Specifically, after introducing in Section 2 the relevant notations and describing in Section 3 the related literature, we present in Section 4 a new solution of the online abelian pattern matching problem in strings, based on a generalization of reactive automata [10,13] in which multiple links are allowed. In addition, we propose a nonstandard simulation of the automaton based on bit-parallelism. Despite its quadratic worst-case complexity, the resulting algorithm performs very well in practice, better than existing solutions in most practical cases (especially when the alphabet is large), as can be inferred from the experimental results reported in Section 5. Finally, the paper is closed with some concluding remarks in Section 6.

2 Notations and Definitions

We represent a string p of length $|\mathsf{p}| = m > 0$ as a finite array $\mathsf{p}[0\,..\,m-1]$ of characters from a finite alphabet Σ of size σ. Thus, $\mathsf{p}[i]$ will denote the $(i+1)$-st character of p, for $0 \leq i < m$, whereas $\mathsf{p}[i\,..\,j]$ will denote the substring of p contained between the $(i+1)$-st and the $(j+1)$-st characters of p.

For a character $c \in \Sigma$, we denote by $\rho_{\mathsf{p}}(c)$ the rightmost position in p of the character c, if present, -1 otherwise. Likewise, $\lambda_{\mathsf{p}}(c)$ will denote the leftmost position in p of the character c, if present, m otherwise. More formally, for $c \in \Sigma$, we have

$$\begin{aligned}\rho_{\mathsf{p}}(c) &:= \max\big(\{i \mid 0 \leq i < m \text{ and } \mathsf{p}[i] = c\} \cup \{-1\}\big)\\ \lambda_{\mathsf{p}}(c) &:= \min\big(\{i \mid 0 \leq i < m \text{ and } \mathsf{p}[i] = c\} \cup \{m\}\big)\,.\end{aligned}$$

For any index $0 \leq i < m$, we let $\nu_{\mathsf{p}}(i)$ denote the smallest index $i < j < m$ such that $\mathsf{p}[j] = \mathsf{p}[i]$, if such an index exists, m otherwise. In addition, we extend the definition of ν_{p} to m by putting $\nu_{\mathsf{p}}(m) := -1$. In the rest of the paper, when the pattern p is understood, we will simply write λ, ρ, and ν in place of λ_{p}, ρ_{p}, and ν_{p}, respectively. For a function f, we use the notation f^j, with $j \geq 0$, for the j-th *iterate* of f.[1] Thus, for instance, $f^3(i) = f(f(f(i)))$.

It is easy to see that, for any index $0 \leq i < m$, the sequence of indices

$$\big\langle \lambda(\mathsf{p}[i]), \nu(\lambda(\mathsf{p}[i])), \nu^2(\lambda(\mathsf{p}[i])), \ldots, \nu^r(\lambda(\mathsf{p}[i]))\big\rangle,$$

where $r+1$ is the multiplicity of $\mathsf{p}[i]$ in p (so that $\nu^r(\lambda(\mathsf{p}[i])) = \rho(\mathsf{p}[i])$), is the sequence of the positions of the character $\mathsf{p}[i]$ in p, in increasing order.

Example 1. Let $\mathsf{p} = gactaagtac$ be a pattern of length $m = 10$ over the alphabet $\Sigma = \{a, c, g, t\}$. Then we have $\lambda(a) = 1$ and $\rho(a) = 8$. Moreover, $\nu(1) = 4$, $\nu^2(1) = \nu(4) = 5$, and $\nu^3(1) = \nu(5) = 8 = \rho(a)$. Thus, $\langle 1, 4, 5, 8\rangle$ is the increasing sequence of the positions of the character a in p.

The *Parikh vector* [1,18] of p (denoted by pv_{p} and also known as *compomer* [6], *permutation pattern* [12], and *abelian pattern* [11]) is the vector of the multiplicities of the characters in p. More precisely, for each $c \in \Sigma$, we have

$$pv_{\mathsf{p}}[c] := |\{i : 0 \leq i < m \text{ and } \mathsf{p}[i] = c\}|\,.$$

In the following, the Parikh vector of the substring $\mathsf{p}[i\,..\,i+h-1]$ of p, of length h and starting at position i, will be denoted by $pv_{\mathsf{p}(i,h)}$.

In terms of Parikh vectors, the abelian pattern matching problem can be formally expressed as the problem of finding the set $\Gamma_{\mathsf{p},\mathsf{t}}$ of positions in t, defined as

$$\Gamma_{\mathsf{p},\mathsf{t}} := \{s : \ 0 \leq s \leq n-m \text{ and } pv_{\mathsf{t}(s,m)} = pv_{\mathsf{p}}\}.$$

We close the section by recalling that a *finite automaton* is a 5-tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, $F \subseteq Q$ is the collection of *final states*, Σ is an *alphabet*, and $\delta \subseteq (Q \times \Sigma \times Q)$ is the *transition relation* of $\mathcal{A}$. We also recall the notation of some bitwise infix operators on computer words, namely the bitwise `and` "&", the bitwise `or` "|", and the `left shift` "$\ll$" operator (which shifts its first argument to the left by a number of bits equal to its second argument): in this context, we will say that a bit *is set* to indicate that its value is equal to 1.

[1] Formally, we put $f^0(x) := x$ and, recursively, $f^{j+1}(x) := f(f^j(x))$, provided that f is defined on $f^j(x)$.

3 Previous Results

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *online abelian pattern matching problem* can be solved in $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma)$ space by using a naïve *prefix based approach* [11], which slides a window of size m over the text while updating in constant time the corresponding Parikh vector. Indeed, for each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathsf{t}(s+1,m)}[c] = pv_{\mathsf{t}(s,m)}[c] - \big|\{c\} \cap \{\mathsf{t}[s]\}\big| + \big|\{c\} \cap \{\mathsf{t}[s+m]\}\big|\,,$$

so that the vector $pv_{\mathsf{t}(s+1,m)}$ can be computed from $pv_{\mathsf{t}(s,m)}$ by incrementing the value of $pv_{\mathsf{t}(s,m)}[\mathsf{t}[s+m]]$ and by decrementing the value of $pv_{\mathsf{t}(s,m)}[\mathsf{t}[s]]$. Thus, the test "$pv_{\mathsf{t}(s+1,m)} = pv_{\mathsf{t}(s,m)}$" can be easily performed in constant time.

A more efficient prefix-based approach, which uses less branch conditions, has been recently proposed in [14]. Specifically, for each position $0 \leq s \leq n-m$, a function $G_s : \Sigma \to \mathbb{Z}$ is defined by putting $G_s(c) := pv_{\mathsf{p}}[c] - pv_{\mathsf{t}(s,m)}[c]$, for $c \in \Sigma$. Also, a distance value δ_s can be defined as $\delta_s := \sum_{c\in\Sigma} \big|G_s(c)\big|$. Then the set $\Gamma_{\mathsf{p},\mathsf{t}}$ takes on the form $\Gamma_{\mathsf{p},\mathsf{t}} = \{s \;:\; 0 \leq s \leq n-m \text{ and } \delta_s = 0\}$. Observe that $G_{s+1}(c)$ and δ_{s+1} can be computed in constant time from $G_s(c)$ and from δ_s, respectively. Hence, it follows that all values δ_s, for $s = 0, \ldots, n-m$, can be computed in $\mathcal{O}(n)$ time.

A *suffix-based approach* to the problem has been presented in [11], as an adaptation of the Horspool strategy [16]. Rather than reading the characters of the window from left to right, characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $\mathcal{O}(nm)$ worst-case time complexity but performs well in practical cases.

Experimental results show that the prefix based algorithm outperforms the suffix based algorithm only for abelian patterns over small alphabets (as, for instance, in the case of binary data or DNA sequences) and for patterns whose characters have a frequency distribution similar to that of the input text. In all other cases, the suffix based approach achieves better results than the prefix based approach. The gap becomes more significant in the case of very large alphabets as is the case, for instance, in natural language texts.

In [11], a *parameterized suffix based approach* has been presented in which the current frequency vector is reset only if the number of the characters read before an overflow does not exceed εm, where ε is a user defined parameter. The worst-case time complexity of the resulting algorithm is $\mathcal{O}(\frac{n}{1-\varepsilon})$. However, experimental results show that the algorithm never outperforms the prefix- and the suffix-based algorithms.

For the sake of completeness, we notice that recently the problem has also been solved in its *offline* form, where one has to search for several patterns in the same text, so that it makes sense to perform in advance a suitable preprocessing of the text. We mention also a solution presented in [7,8], in which a useful data structure over the input text is constructed beforehand in $\mathcal{O}(n)$ time and space. As a result, each query can be answered in $\mathcal{O}(n)$ worst-case time complexity, though with a sublinear expected time complexity.

4 A New Algorithm Based on Reactive Multi-Automata

Reactive automata, introduced in [10,13], are ordinary automata (deterministic or nondeterministic) augmented with a switching mechanism to turn links on or off during computation. Thanks to the switching mechanism, the number of states in an ordinary automaton can be dramatically reduced.

For our purposes, we will need to slight generalize the notion of reactive automata as given in [10,13], by also allowing multiple links labeled by a same character between any two states.[2] We will therefore provide in Section 4.1 a formal definition of *reactive multi-automata* and of the related acceptance notion. Then, in Section 4.2, we show how to construct a compact reactive multi-automaton which recognizes all abelian occurrences of a given input pattern, and prove its correctness in Section 4.3. Subsequently, in Section 4.4, we present an algorithm for the online abelian pattern matching problem which makes use of such an automaton and, finally, in Section 4.5 we describe how to efficiently simulate it using bit-parallelism.

4.1 Reactive Multi-Automata

A reactive automaton is an ordinary automaton extended with *reactive links* between its (ordinary) links. These can be of two types, namely *activation* and *deactivation* reactive links. At any step of the computation of a reactive automaton on a given input string S, states and links are distinguished as active and non-active. At start (step 0), the initial state is the only active state and all links of a given *initial transition relation* are active.[3] Active states at step $h+1$ are all states which are reachable by a direct *active* link (at step h), labeled by the character $S[h]$, from any *active* state (at step h). Active links at step $h+1$ are all links which are active at step h and are not deactivated in the transition from step h to step $h+1$, plus all links which are activated in the transition from step h to step $h+1$. A link is activated in the transition from step h to step $h+1$, if it is the endpoint of an activation reactive link from an active (ordinary) link at step h labeled by the character $S[h]$. A link is deactivated in the transition from step h to step $h+1$, if it is the endpoint of a deactivation reactive link from an active (ordinary) link at step h labeled by the character $S[h]$ *and* it does not get activated at the same time (in other words, we stipulate that when a link is both activated and deactived, activation prevails).

Reactive multi-automata extend reactive automata in that they allow the presence of multiple links labeled by a same character between any two states. We choose to represent multiplicity by means of *multiplicity labels* drawn from a finite set of labels L. Thus, a link in a multi-automata is a quadruple (q, c, ℓ, q'), where q, q' are states, c is an alphabet character, and ℓ is a multiplicity label. From an operational point of view, two links differing only on their multiplicity label are regarded just the same.

Let us be more formal. Let Q, Σ, L be finite sets of states, of characters, and of labels, respectively, and let $\mathcal{D} := Q \times \Sigma \times L \times Q$ denote the collection of all possible labeled links on Q, Σ, and L. Also, let $T^+, T^- \subseteq \mathcal{D} \times \mathcal{D}$ be two collections of activation and deactivation reactive links, respectively. Given a set $\psi \subseteq \mathcal{D}$ of links (which are supposed to be the active links at a certain step h) and a subset $\varphi \subseteq \psi$ (of the links in ψ from active states and labeled by the input word character which is being read at step h), then the set of active links (at the subsequent step $h+1$) relative to φ and to the collections T^+, T^- of reactive links, denoted by $\psi^{(\varphi, T^+, T^-)}$, is

$$\begin{aligned}\psi^{(\varphi,T^+,T^-)} := \big(\psi \setminus \{\gamma \mid \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^-\}\big) \\ \cup\ \{\gamma \mid \exists\, \tau \in \varphi \text{ such that } (\tau,\gamma) \in T^+\}\,.\end{aligned}$$

[2] In fact, we will only need multiple self-loops.

[3] As we will see, the initial transition is a subset of the transition relation of the underlying automaton.

The map $\psi \mapsto \psi^{(\varphi,T^+,T^-)}$ just defined is the *switch reactive transformation relative to* T^+, T^-.

We are now ready to give a precise definition of reactive multi-automata and of their nondeterministic runs.

Definition 2 (Reactive multi-automata). *Let Q, Σ, L be finite sets of states, of characters, and of labels, respectively.*

A reactive multi-automaton *is a nonuple $\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$, where*

- $(Q, \Sigma, L, q_0, \delta, F)$ *is a* multi-automaton *(called the* multi-automaton underlying *$\mathcal{R}$), with $q_0 \in Q$ (initial state), $F \subseteq Q$ (set of final states), and $\delta \subseteq Q \times \Sigma \times L \times Q$ (transition relation);*
- $T^+, T^- \subseteq \delta \times \delta$ *are the sets of activation and deactivation reactive links;*
- $\overline{\delta} \subseteq \delta$ *is the set of initially active links (initial transition relation).*

Definition 3 (Nondeterministic runs). *Let $\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$ be a reactive multi-automaton and let $S = s_0 s_1 \cdots s_{n-1}$ be a word on the alphabet Σ.*

The nondeterministic run of $\mathcal{R}$ over S is a sequence of pairs (Q_h, δ_h), for $h = 0, \ldots, n$, where $Q_h \subseteq Q$ and $\delta_h \subseteq \delta$ are respectively the set of active states *and the set of* active transitions *at step h, where, for $h = 0$,*

$$(Q_0, \delta_0) := \big(\{q_0\}, \overline{\delta}\big)$$

and, recursively, for $0 < k \leq n$,

$$\begin{aligned} Q_h &:= \big\{q \mid (r, s_{h-1}, \ell, q) \in \delta_{h-1}, \textit{ for some } r \in Q_{h-1}, \ell \in L\big\} \\ \delta_h &:= \delta_{h-1}^{(\varphi_{h-1},T^+,T^-)}, \end{aligned}$$

where $\varphi_{h-1} := \{(r, s_{h-1}, \ell, q) \mid (r, s_{h-1}, \ell, q) \in \delta_{h-1}$ and $r \in Q_{h-1}\}$ and $\delta_{h-1}^{(\varphi_{h-1},T^+,T^-)}$ is the result of a switch reactive transformation applied to δ_{h-1}, relative to φ_{h-1}, T^+, T^-.

We say that the word S is accepted *by $\mathcal{R}$ provided that the nondeterministic run $\big\langle (Q_0, \delta_0), (Q_1, \delta_1), \ldots, (Q_n, \delta_n)\big\rangle$ of $\mathcal{R}$ over S is such that $Q_n \cap F \neq \emptyset$.*

Remark 4. The above definitions of switch reactive transformation, reactive multi-automaton, and nondeterministic run can be easily extended to the case in which ε-transitions are present, at least when no reactive link is allowed to have an ε-transition as its first component, which is what we will assume in the rest of the paper. In the context of multi-automata, ε-transitions take the form $(q, \varepsilon, \ell, q')$, where q, q' are states and ℓ is a label. In the nondeterministic run over a word S, if at a certain step h the ε-transitions

$$(q, \varepsilon, \ell, q'),\ (q', \varepsilon, \ell', q''),\ \ldots,\ (q^{(r-1)}, \varepsilon, \ell^{(r-1)}, q^{(r)})$$

are active and the states $q, q', \ldots, q^{(r-1)}$ are also active, the state $q^{(r)}$ will become active at step $h+1$, independently of the $(h+1)$-st character of S.

In view of the above observation, it is not hard to extend formally Definitions 2 and 3 to the case in which ε-transition are allowed.

4.2 The Abelian Reactive Multi-Automaton

Next we define the *abelian reactive multi-automaton* for a given pattern p of length m over an alphabet Σ, which accepts all and only the $\frac{m!}{\prod_{c\in\Sigma}(pv_{\mathsf{p}}[c])!}$ distinct permutations of p, where pv_{p} is the Parikh vector of p.

Definition 5 (Abelian Reactive Multi-Automaton). *Let* p *be a pattern of length* m *over an alphabet* Σ *and let* $\langle b_0, b_1, \ldots, b_{k-1}\rangle$ *be the sequence of the distinct characters occurring in* p*, ordered by their first occurrence. The* abelian reactive multi-automaton *(ARMA) for* p *is the reactive multi-automaton with* ε*-transitions*

$$\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$$

such that

- $Q = \{q_0, q_1, \ldots, q_k, \omega\}$ *is the set of states, where* q_0 *is the initial state and* ω *is a special state called the* overflow state*;*
- $F = \{q_k\}$ *is the set of final states;*
- $L = \{\ell_0, \ell_1, \ldots, \ell_{m-1}\}$ *is a set of labels of size* m*;*
- *the transition relation* δ *of* $\mathcal{R}$ *and its subset* $\overline{\delta} \subseteq \delta$ *of the links initially active (initial transition relation) are defined as follows*

$$\begin{aligned}
\delta \;&:= \{(q_i, \varepsilon, \ell_0, q_{i+1}) \mid 0 \le i < k\} && \textit{(ε-transitions)}\\
&\cup \{(q_0, p[i], \ell_i, q_0) \mid 0 \le i < m\} && \textit{(self-loops)}\\
&\cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma\} && \textit{(overflow transitions)}\\
&\cup \{(\omega, c, \ell_0, \omega) \mid c \in \Sigma\} && \textit{(overflow self-loops)}\\
\overline{\delta} \;&:= \{(q_0, c, \ell_{\lambda(c)}, q_0) \mid c \in \Sigma_{\mathsf{p}}\}\\
&\cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma \setminus \Sigma_{\mathsf{p}}\}\\
&\cup \{(\omega, c, \ell_0, \omega) \mid c \in \Sigma\}
\end{aligned}$$

- *the sets* T^+ *and* T^- *of activation and deactivation reactive links are defined as follows*

$$\begin{aligned}
T^+ \;&:= \{((q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_i, \varepsilon, \ell_0, q_{i+1})) \mid 0 \le i < k\}\\
&\cup \{((q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_0, \mathsf{p}[\rho(b_i)], \ell_{\rho(b_i)}, \omega)) \mid 0 \le i < k\}\\
&\cup \{((q_0, \mathsf{p}[i], \ell_i, q_0), (q_0, \mathsf{p}[\nu(i)], \ell_{\nu(i)}, q_0)) \mid 0 \le i < m \textit{ and } i \ne \rho(p_i)\}\\
T^- \;&:= \{((q_0, p[i], \ell_i, q_0), (q_0, p[i], \ell_i, q_0)) \mid 0 \le i < m\}\,.
\end{aligned}$$

Fig. 1 shows the general structure of a portion of an abelian reactive automaton, whereas Fig. 2 shows the complete abelian reactive automaton for the pattern $P = \mathsf{acca}$, up to deactivation reactive links, which are not shown.

The following property states that the size of the abelian reactive automaton for a given pattern p of length m is linear in the size of p and of the underlying alphabet. This contrasts with the $\mathcal{O}(2^m)$ size of the minimal standard automaton accepting the same language.

Property 6. The abelian reactive automaton for a pattern p of length m, with $k \le m$ distinct characters, over an alphabet of size σ has size $\mathcal{O}(m+\sigma)$. Specifically it has $k+2$ states, $k+m+2\sigma$ transitions, and $2m+k$ reactive links. In addition, it can be constructed and initialized in $\mathcal{O}(m+\sigma)$ time and space.

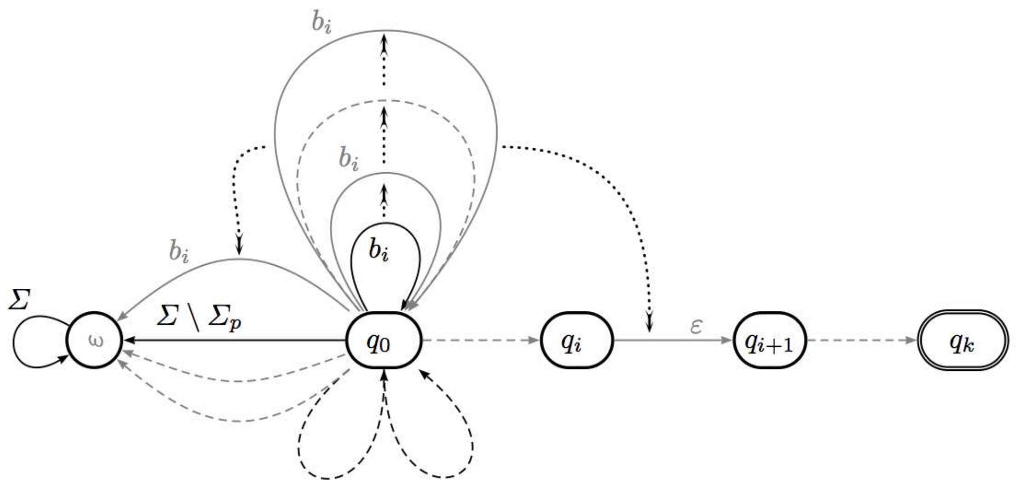

Figure 1. A portion of the general structure of an abelian reactive automaton. Standard transitions in δ are represented with solid and dashed lines, reactive links in T^+ are represented with dotted lines while reactive links in T^- are not represented. Non active links are represented in gray color.

Given a pattern p of length m with k distinct characters $b_0, b_1, \dots, b_{k-1}$ (ordered by their first occurrence in p), the abelian reactive multi-automaton for p contains $k+1$ 'ordinary' states $q_0, q_1, \dots, q_k$ and a path of k consecutive ε-transitions $(q_i, \varepsilon, \ell_0, q_{i+1})$, for $i = 0, 1, \dots, k-1$, starting from the initial state q_0 and ending on its final state q_k. For $i = 0, 1, \dots, k-1$, we will refer to $(q_i, \varepsilon, \ell_0, q_{i+1})$ as the *ε-transition of the automaton for the character b_i*. Initially, all such transitions are non-active.[4]

An additional state ω, named *overflow state*, is used to detect when the number of occurrences of a character in the current text window exceeds its multiplicity in the pattern. For each character c in the alphabet, the automaton contains a transition labeled by c from the initial state to the overflow state, called the *overflow transition for c*, and from the overflow state to itself, called the *overflow self-loop for c*. Initially, all overflow self-loops and all overflow transitions for the characters *not* occurring in the pattern are active,[5] whereas the overflow transitions for the characters occurring in the pattern are non-active.

For each character c occurring in the pattern p with multiplicity m_c (and, specifically, at positions $0 \leq h_0 < h_1 < \cdots < h_{m_c-1} < m$), the automaton contains also a set $\mathfrak{M}_c := \{(q_0, c, \ell_{h_i}, q_0) | i = 0, 1, \dots, m_c - 1\}$ (called the *monad of c*) of m_c self-loops labeled by c, from state q_0 into itself. Initially, only the first self-loop in $\mathfrak{M}_c$, corresponding to the leftmost occurrence of c in the pattern, is active, whereas the remaining ones are all non-active. Each self-loop in $\mathfrak{M}_c$ has a deactivation reactive link pointing to itself. In addition, each of the first $m_c - 1$ self-loops in $\mathfrak{M}_c$ has an activation reactive link pointing to the next self-loop in the monad, whereas the last self-loop in $\mathfrak{M}_c$ has two activation reactive links, one pointing to the overflow transition for c and one pointing to the ε-transition relative to c.

[4] As we will see, the final state becomes reachable only when all such ε-transitions have been activated during the recognition process.

[5] In fact, all overflow self-loops and all overflow transitions for the characters *not* occurring in the pattern remain active during the whole recognition process.

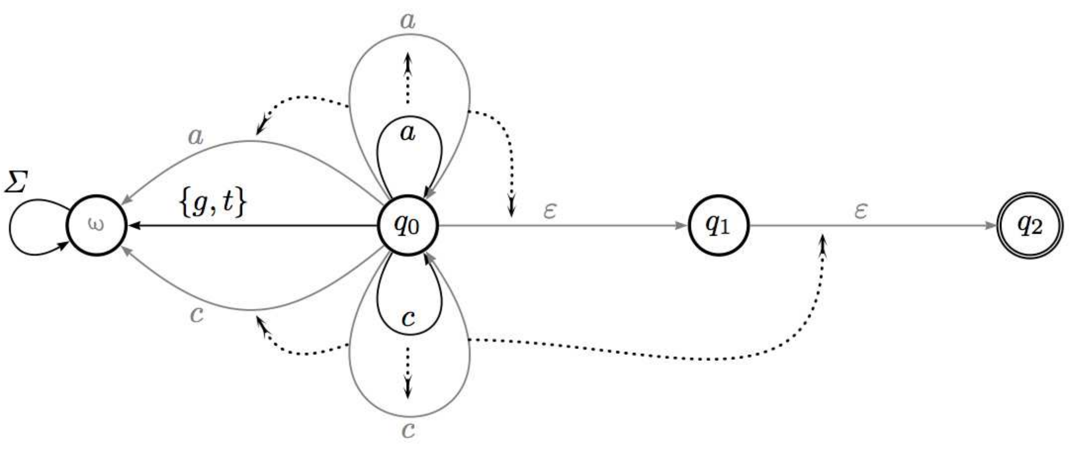

Figure 2. The complete abelian reactive automaton for the pattern $P =$ acca over the DNA alphabet $\Sigma = \{a, c, g, t\}$. Standard transitions are represented with solid lines while reactive links in T^+ are represented with dashed lines. Reactive links in T^- are not represented. Non active transitions are represented in gray color.

4.3 Correctness

Next we show that the language accepted by the abelian reactive multi-automaton for a pattern p is exactly the set of all the permutations of p.

As in the previous section, let p be a pattern of length m with k distinct characters $b_0, b_1, \ldots, b_{k-1}$ (ordered by their first occurrence in p) and let

$$\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, \{q_k\}\big)$$

be the ARMA for p, with $Q = \{q_0, q_1, \ldots, q_k, \omega\}$ and $L = \{\ell_0, \ell_1, \ldots, \ell_m\}$. In addition, let s be an input string to be recognized by $\mathcal{R}$.

To begin with, we observe that as soon as an overflow transition is followed (in which case we say that an *overflow condition* has occurred), the computation gets trapped in the overflow state ω, so that q_0 is no longer active and the final state q_k cannot be reached anymore. As we will soon see, this happens when it is detected that s contains some character whose multiplicity in s exceeds that in the pattern p.

As long as the initial state q_0 is active, the transitions in the monad $\mathfrak{M}_b$ of b, for each character b in p, allow one to count the number of occurrences of b which have been read so far from the string s, when this number does not exceed the multiplicity m_b of b in p. Specifically, as a result of the interplay of the deactivation and activation reactive links on each of the first $m_b - 1$ transitions of the monad, when exactly $0 \leq i < m_b$ occurrences of b have been read from the string s, it turns out that $(q_0, b, \ell_{h_i}, q_0)$ is the only active transition in the monad $\mathfrak{M}_b$, where $0 \leq h_0 < h_1 < \cdots < h_{m_b-1} < m$ are the positions of the occurrences of b in p in increasing order. In addition, just after the m_b-th occurrence of b is read from the string s, all transitions in $\mathfrak{M}_b$ are non-active (and remain so for the rest of the recognition process), whereas the overflow transition and the ε-transition for b (which initially were non-active) become active and stay active until the end. Thus, if a further occurrence of b is read, the overflow transition for b is followed, leading to the overflow state ω, where the computation gets trapped.

Since the overflow transition for any character not occurring in p is active for the whole recognition process, as soon as an occurrence of a character not in p is found in s, the overflow transition associated to it is followed, leading again to the overflow state ω. This corresponds to having an empty monad for each character not occurring in p.

The above considerations allow us to conclude that if the string s contains any character whose multiplicity in s exceeds that in p, then the recognition process gets trapped in the overflow state ω, so that s is correctly rejected by the automaton $\mathcal{R}$.

On the other hand, if the multiplicity of no character in s exceeds that in p, then the state q_0 remains active until the end of the recognition process by $\mathcal{R}$. However, at termination, the accepting state q_k is active only if all the ε-transitions from q_0 to q_k (initially non-active) have been activated. As seen above, since q_0 is always active, this happens only if the string s and the pattern p contain the same characters and each of them occurs in s and in p with the same multiplicity; in other words, only if s is a permutation of p.

In conclusion, the language accepted by the ARMA $\mathcal{R}$ for p is the set of all the permutations of p.

4.4 The Algorithm

The algorithm that we present in this section makes use of the abelian reactive multi-automaton defined above for locating all occurrences of the permutations of a given pattern p of length m in a text t of length n.

In the preprocessing phase, the algorithm computes in $\mathcal{O}(m+\sigma)$ time and space the Parikh vector pv_{p} of the pattern and constructs the corresponding reactive multi-automaton.[6]

The algorithm works by sliding a window of size m over the text. At start, the left ends of the window and of the text are aligned. An *attempt* consists in checking whether the current window is a permutation of the pattern. This is done by executing the ARMA for the pattern over the window text. When the whole window has been read (or as soon as an overflow condition occurs) the window is shifted to the right of the last character examined. The attempts take place in sequence, until the right end of the window goes past the right end of the text.

Let us consider a generic attempt at position s of the text t, so that the current window is the substring $\mathsf{t}[s\,..\,s+m-1]$. At the beginning of the attempt, the automaton is initialized in time $\mathcal{O}(m)$. Then, during the attempt, the algorithm scans the window from right to left, while executing the corresponding automaton transitions.

If the whole text window has been scanned and no overflow condition has occurred, an occurrence of a permutation of the pattern is reported at position s. In this case the window is advanced by one position to the right.

On the other hand, if an overflow condition occurs while reading the character at position j in the text, with $s \leq j < s+m$, then the substring $\mathsf{t}[j\,..\,s+m-1]$ cannot be a permutation of the pattern, as it contains too many occurrences of the character $\mathsf{t}[j]$. Thus, it is safe to shift the window by $j-s+1$ positions to the right.

Each attempt takes $\mathcal{O}(m)$ worst-case time. Since the minimum advancement performed at the end of each attempt is by one position, the worst-case time complexity of the whole algorithm is $\mathcal{O}(nm)$.

[6] The construction of the reactive multi-automaton is straightforward and details have been omitted.

```
BAM($p, m, t, n, \Sigma$)
1. for each $c \in \Sigma$ do $M[c] \leftarrow pv_{\mathsf{p}}[c] \leftarrow 0$
2. $I \leftarrow F \leftarrow sh \leftarrow 0$
3. for $i \leftarrow 0$ to $m-1$ do $pv_{\mathsf{p}}[\mathsf{p}[i]] \leftarrow pv_{\mathsf{p}}[\mathsf{p}[i]] + 1$
4. for each $c \in \Sigma$ do
5.     if $pv_{\mathsf{p}}[c] > 0$ then
6.         $M[c] \leftarrow M[c] \mid (1 \ll sh)$
7.         $I \leftarrow I \mid (((1 \ll \log m) - pv_{\mathsf{p}}[c] - 1) \ll sh)$
8.         $F \leftarrow F \mid (1 \ll (sh + \log m))$
9.         $sh \leftarrow sh + \log m + 1$
10. $F \leftarrow F \mid (1 \ll sh)$
11. for each $c \in \Sigma$ do
12.     if $pv_{\mathsf{p}}[c] = 0$ then $M[c] \leftarrow M[c] \mid (1 \ll sh)$
13. $s \leftarrow 0$
14. while $s \leq n - m$ do
15.     $D \leftarrow I$; $j \leftarrow s + m - 1$
16.     while $j \geq s$ do
17.         $D \leftarrow D + M[\mathsf{t}[j]]$
18.         if $(D \,\&\, F)$ then break
19.         $j \leftarrow j - 1$
20.     if $j < s$ then
21.         OUTPUT($s$)
22.         $s \leftarrow s + 1$
23.     else $s \leftarrow j + 1$
```

Figure 3. The Bit-Parallel Abelian Matcher for the abelian pattern matching problem (BAM).

4.5 An Efficient Bit-Parallel Simulation

In this section we show how to simulate efficiently the abelian reactive multi-automaton for an input pattern p (cf. Definition 5), by using the bit-parallelism technique [2].

Let again $b_0, b_1, \ldots, b_{k-1}$ be the distinct characters in p.

The underlying idea is to associate a counter to each distinct character in p, plus a single 1-bit counter for the remaining characters of the alphabet which do not occur in p, maintaining them in the same computer word. In particular, the counter associated to the character b_i in p, for $i = 0, 1, \ldots, k-1$, will be represented by a group of l_i bits, where $l_i := \lceil \log(pv_{\mathsf{p}}[b_i]) \rceil + 1$. These are just enough to allocate the multiplicity $pv_{\mathsf{p}}[b_i]$ of b_i in p, plus an extra bit called the *i-th overflow bit*. Whenever an occurrence of the character b_i is read in the current text window (which, as before, is scanned backwards), its counter is incremented. Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathsf{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathsf{p}}[b_i]$ increments. Hence, the overflow bit gets set only when the $(pv_{\mathsf{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window, if it exists, at which point it becomes clear that the text window cannot be a permutation of the pattern p. Likewise, the 1-bit counter reserved for all the characters not occurring in p is initially null and it gets set as soon as any character not in p is encountered in the text window, at which point, again, it becomes clear that the text window cannot be a permutation of the pattern p. By suitably masking the computer word allocating all the counters, it is possible to check in a single pass whether the character of the text window that has just been read has caused any of the $k+1$ overflow bits to be set. If this is the case, the window text is advanced just past the last character read. Otherwise, when the current text window has been scanned completely and no overflow bit has been set, a matching is reported and the window text is advanced one position to the right.

The resulting algorithm, named Bit-Parallel Abelian Matcher (BAM) is shown in Fig. 3. It works in a similar way as the ARMA algorithm.

During the preprocessing phase (lines 4-12), for each distinct character b_i occurring in p, a bit mask $M[b_i]$ of $l+1$ bits is computed, where

$$l := \sum_{i=0}^{k-1} l_i \qquad \text{and} \qquad M[b_i] := 1 \ll \Big(\sum_{j=0}^{i-1} l_j\Big).$$

The bit mask $M[b_i]$ is then used in line 17 to increment the counter in D associated to the character b_i.

Two additional bit masks of $l+1$ bits are used: the bit mask I, which contains the initial values for each counter, and the bit mask F, whose bits set are exactly the overflow bits. These are defined by

$$I := \sum_{i=0}^{k-1} \Big[\Big(2^{l_i} - pv_{\mathsf{p}}[b_i] - 1\Big) \ll \sum_{j=0}^{i-1} l_j\Big] \quad \text{and} \quad F := \sum_{i=0}^{k-1} \Big[1 \ll \Big(\sum_{j=0}^{i} l_j - 1\Big)\Big].$$

Let us consider a generic attempt at position s of the text (lines 14-23), so that the current text window is the substring $\mathsf{t}[s \mathinner{..} s+m-1]$. At the beginning of each attempt, a bit mask D of $l+1$ bits (intended to represent the Parikh vector of the text window) is initialized to I (line 15). Then, during the attempt, the window is read character by character, proceeding from right to left (lines 16-19). When reading the character $\mathsf{t}[j]$ of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathsf{t}[j]]$ (line 17).

The attempt stops when the left end of the window is reached or when an overflow bit in D is set. In the first case, an occurrence is reported at position s and the window is advanced to the right by one position (lines 20-22). In the second case, i.e., when the counter update for a character $\mathsf{t}[j]$ has set an overflow bit in D (and therefore $D\&F \neq 0$ holds), the substring $\mathsf{t}[j \mathinner{..} s+m-1]$ cannot be involved in any match, as it contains too many occurrences of the character $\mathsf{t}[j]$, and therefore it is safe to shift the window to the right by $j-s+1$ positions (line 23).

As in the case of the ARMA algorithm, each attempt takes $\mathcal{O}(m)$-worst case time and at most n attempts take place during the whole execution. Thus the worst-case time complexity of the BAM algorithm is $\mathcal{O}(nm)$, whereas the space requirement for maintaining a bit mask for each character of the alphabet is $\mathcal{O}(\sigma)$.

So far we have implicitly assumed that $l+1 \leq w$, where w is the size of a computer word, so that each of the vectors D, I, F, and $M[b_i]$, for b_i in p, fits in a single computer word.

When $l+1 > w$, we must content ourselves to maintain the counters only for a proper selection Σ'_{p} of the set of characters occurring in p. In this case, when a match relative to the characters in Σ'_{p} is reported, an additional verification phase must be run, in order to discard possible *false positives*.

5 Experimental Results

In this section we evaluate the performance of the bit-parallel simulation BAM described in the previous section and compare it with some standard solutions known in literature. In particular we compare the performances of the following three algorithms: the prefix based algorithm due to Grabowsky *et al.* (GFG) [14], the algorithm using the suffix based approach (SBA) [11], and the Bit-parallel Abelian Matcher (BAM) described in Section 4.5.

m	GFG	SBA	BAM
2	**23.56**	39.20	27.03
4	**23.56**	33.27	23.17
8	23.54	27.54	**19.01**
16	23.49	24.05	**16.21**
32	23.52	23.78	**15.63**
64	23.50	25.33	**16.12**
128	23.57	28.74	**17.69**
256	23.53	33.14	**19.63**

m	GFG	SBA	BAM
2	23.08	**18.07**	12.51
4	23.00	**15.39**	10.36
8	22.96	13.67	**9.40**
16	23.03	11.91	**8.44**
32	23.04	9.58	**7.16**
64	23.01	8.46	**6.64**
128	22.97	7.82	**6.49***
256	22.96	7.84	**7.69***

Table 1. Experimental results on a genome sequence (on the left) and a on a protein sequence (on the right). An asterisk symbol (*) indicates those runs where false positives have been detected. All best results have been boldfaced.

All algorithms have been implemented in `C` and compiled with the `GNU C Compiler 4.2.1`, using the optimization option `-O3`. The experiments have been executed locally on a MacBook Pro with 4 Cores, a 2 GHz Intel Core i7 processor, 4 GB RAM 1333 MHz DDR3. The three algorithms have been compared in terms of their running times, including any preprocessing time, measured with a hardware cycle counter, available on modern CPUs.

In our tests we used a genome sequence, with an alphabet of size $\sigma = 4$ (Table 1, on the left) and a protein sequence, with an alphabet of size $\sigma = 20$ (Table 1, on the right), both of 4 MB length.[7] For each input file, we have generated sets of 500 patterns of fixed length m randomly extracted from the text, where $m \in \{2, 4, 8, 16, 32, 64, 128, 256\}$, and reported the mean time over the 500 runs, expressed in milliseconds.

From the experimental results it turns out that the GFG algorithm has a linear behavior in practice and is almost insensitive to the size of the pattern, whereas the algorithms based on a backward approach, such as SBA and BAM, show a sublinear behavior although their theoretical worst-case time complexity is quadratic.

In all cases, when the pattern is longer than 4 characters, the BAM algorithm outperforms the other two algorithms.

The SBA and the BAM algorithms improve their performances in the case of larger alphabets and turn out to be the best solutions when searching for a protein sequence. In this case their performances are up to 3 times faster than the GFG algorithm.

In the case of the protein sequence, we observed some cases where false positive occurrences were detected by an additional verification. Such events are indicated in Table 1 with an asterisk (*). However, in all cases the average number of additional verification runs, for each text position, turned out to be less than 10^{-4}.

6 Conclusions

We have presented a new approach to solve the abelian pattern matching problem for strings which is based on a reactive multi-automaton with only $\mathcal{O}(k)$ states and $\mathcal{O}(m)$ transitions. We have also proposed an efficient simulation of such automaton using bit-parallelism. Our solution is based on a backward approach and, despite its quadratic worst-case time complexity, shows a sublinear behavior in practical cases.

[7] The text buffers are described and available for download at the SMART web page (`http://www.dmi.unict.it/~faro/smart/`).

References

1. A. Amir, A. Apostolico, G. M. Landau, G. Satta: Efficient Text Fingerprinting Via Parikh Mapping. Journal of Discrete Algorithms, 1(5–6) 2003, pp. 409–421.
2. R. A. Baeza-Yates, G. H. Gonnet: A new approach to text searching. Commun. ACM, 35 (10) 1992, pp. 74–82.
3. R. A. Baeza-Yates, G. Navarro: New and faster filters for multiple approximate string matching. Random Struct. Algorithms 20 (1) 2002, pp. 23–49.
4. G. Benson: Composition alignment. In: WABI 2003, pp. 447–461.
5. S. Böcker: Simulating multiplexed SNP discovery rates using base-specific cleavage and mass spectrometry. Bioinformatics 23 (2) 2007, pp. 5–12. `http://dx.doi.org/10.1093/bioinformatics/btl291`
6. S. Böcker: Sequencing from compomers: Using mass spectrometry for DNA de novo sequencing of 200+ nt. Journal of Computational Biology 11 (6) 2004, pp. 1110–1134.
7. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: Algorithms for jumbled pattern matching in strings. Int. J. Found. Comput. Sci. 23 (2) 2012, pp. 357–374.
8. P. Burcsi, F. Cicalese, G. Fici, Zs. Lipták: On approximate jumbled pattern matching in strings. Theory Comput. Syst. 50 (1) 2012, pp. 35–51.
9. D. Cantone, S. Cristofaro, S. Faro: Efficient matching of biological sequences allowing for non-overlapping inversions. In: CPM 2011, pp. 364–375.
10. M. Crochemore, D. M. Gabbay: Reactive automata. Inf. Comput., 209(4) 2011, pp. 692–704.
11. E. Ejaz: Abelian pattern matching in strings. Ph.D. Thesis, Dortmund University of Technology (2010), `http://d-nb.info/1007019956`.
12. R. Eres, G. M. Landau, L. Parida: Permutation Pattern Discovery in Biosequences. Journal of Computational Biology, 11(6) 2004, pp. 1050–1060.
13. D. M. Gabbay: Pillars of computer science. Springer-Verlag 2008. Ch. Introducing reactive Kripke semantics and arc accessibility, pp. 292–341.
14. S. Grabowski, S. Faro, E. Giaquinta: String matching with inversions and translocations in linear average time (most of the time). Inf. Process. Lett. 111 (11) 2011, pp. 516–520.
15. R. Grossi, F. Luccio: Simple and efficient string matching with k mismatches. Inf. Process. Lett. 33 (3) 1989, pp. 113–120.
16. R. N. Horspool: Practical fast searching in strings. Software – Practice & Experience 10 (6) 1980, pp. 501–506.
17. P. Jokinen, J. Tarhio, E. Ukkonen: A comparison of approximate string matching algorithms. Softw. Pract. Exp. 26 (12) 1996, pp. 1439–1458.
18. A. Salomaa: Counting (scattered) subwords. Bulletin of the EATCS 81 (2003), pp. 165–179.