Extracting approximate patterns
(Extended abstract)

Johann Pelfrêne[1,3]*, Saïd Abdeddaïm[2], and Joël Alexandre[3]

[1] ExonHit Therapeutics - 65, Boulevard Masséna - 75013 Paris
[2] ABISS,LIFAR - Université de Rouen - 76821 Mont Saint Aignan
[3] ABISS, UMR CNRS 6037 - Université de Rouen - 76821 Mont Saint Aignan
{johann.pelfrene,said.abdeddaim,joel.alexandre}@univ-rouen.fr

Abstract. In a sequence, approximate patterns are exponential in number. In this paper, we present a new notion of basis for the patterns with don't cares occurring in a given text (sequence). The primitive patterns are of interest since their number is lower than previous known definitions (and in a case, sub-linear in the size of the text), and these patterns can be used to extract all the patterns of a text.
We present an incremental algorithm that computes the primitive patterns occurring at least q times in a text of length n, given the N primitive patterns occurring at least $q-1$ times, in time $O(|\Sigma|Nn^2 \log^2 n \log\log n)$. In the particular case where $q = 2$, the complexity in time is only $O(|\Sigma|n^2 \log^2 n \log\log n)$. We also give an algorithm that decides if a given pattern is primitive in a given text.

1 Introduction

Pattern discovery plays a great role in 'knowledge' extraction [16]. Patterns are repeated structured objects in a text (sequence) or in a set of texts. Extraction algorithms are widely used in bioinformatics for protein motif discovery [6], gene prediction (promotor consensus identification [8], detection of splice sites), repeated patterns [4], sequence alignment [17, 15, 7, 9]. Patterns may be used in other fields such as data compression [2].

We consider a pattern with don't cares: a new symbol is considered to be a joker, matching every other letter of the alphabet. A problem of extracting patterns with don't cares (or simply patterns in the following) from a given text is that they are potentially exponential in number in the size of the sequence from which they are extracted. This is the main problem which arises for an interesting application of patterns: sequence indexing. Since indexing texts has been widely covered in the case of exact patterns [3], due to the exponential number of patterns, indexing approximate patterns had never been considered.

* Supported by the Ministry of Research (France) with the "Bio-ingénierie 2000" program, and the CIFRE convention, in a collaboration between ExonHit Therapeutics and ABISS - http://johann.jalix.org/research

A recent work [10] present an idea that tends to reduce the number of patterns, by introducing the maximal non redundant patterns that form a basis for all the patterns appearing at least q times in a sequence (q-patterns).

In [11, 12], we have proposed a new notion of primitive q-patterns. The primitive q-patterns form a basis for the patterns that occur at least q times, which is smaller than the maximal non redundant q-patterns basis. The number of primitive 2-patterns is bounded by n [12]. We also proposed an algorithm that computes the primitive 2-patterns in time $O(n^3)$. In this paper, we show the bound for the primitive 2-patterns, and we give a new algorithm, which is both an improvement of the one in [12], and a generalization to every q-pattern. The algorithm computes incrementally the primitive q-patterns given the N primitive (q-1)-patterns in time $O(|\Sigma|Nn^2 \log^2 n \log\log n)$. As there is only one primitive 1-pattern (the text itself), the complexity in time in the case where $q = 2$ is simply $O(|\Sigma|n^2 \log^2 n \log\log n)$. The number of primitive q-patterns is lower than the number of maximal non redundant q-patterns defined by [10]. We also give an algorithm, that decides if a pattern is primitive given the text where it occurs and the considered quorum.

Very recently, Pisanti et al. have independently presented the same basis they call 'tilling motifs', and the algorithm for the case $q = 2$ [13].

First, we will present some general definitions, followed by the definition of a primitive pattern, the new algorithm for extracting these primitive patterns and a new algorithm for deciding whether a given pattern is primitive or not.

2 Definitions

We denote by Σ the alphabet (set of elements called letters) on which are written texts (or sequences or words) $w = w[1]w[2]\cdots w[n] : w[i] \in \Sigma$. The length of w is n, denoted by $|w|$. In the following, we will work on a text t of length $|t| = n$.

In $t = uwv$, the integer $i = |u| + 1$ is a position of the word w in the word t. w is a factor of t at the position i, also denoted by $t[i, |w|]$. If $v = \varepsilon$ (the empty word), w is a suffix of t, denoted $suff(i, t)$.

A pattern is a word written on the alphabet $\Sigma \cup \{\_\}$: the letter $\_$ is to be thought as a don't care (we will name it 'joker' sometimes). Let $\Sigma = \{A, B, C, D\}$, a pattern $m = A\_C$ corresponds to the language $L_m = \{AAC, ABC, ACC, ADC\}$.

The set $pos(m, t)$ is the set of all the positions where the pattern m occurs in the text t. The shift δ of a position list P is a position list such that if $P = \{p_1, p_2, \cdots, p_k\}, P + \delta = \{p + \delta / p \in P, 1 \leq p + \delta \leq |t|\}$.

A q-pattern is a pattern that occurs at least q times (such that $|pos(m, t)| \geq q$). In the sequence `AAAAAXAAAAA`, `AA_AA_AA` is a 2-pattern.

2.1 Maximal patterns

The idea is to take patterns as long as possible, using the least amount of don't cares as possible. We first create groups of motifs that occur on the same positions, with eventually a shift : $m_1 \equiv_t m_2 \Leftrightarrow \exists \delta \in \mathbb{Z} : pos(m_1, t) = pos(m_2, t) + \delta$ This is clearly an equivalence relation.

For example, in the text
t=GACATGATTATTGATGCCTAACCGGCTTATGGATAGC, the following patterns are in the same equivalence class : TG_TA, ATG_T, AT__TA et T__TA, ATG_TA.

Next, we define the weight of a pattern m as $w(m)$, the number of solid letters in the pattern. For example $w(A\_C) = 2$.

Definition 1 (Maximal Pattern) *A maximal pattern m is the shortest pattern (in length) that has the highest weight among the patterns m_i such that $m_i \equiv_t m$.*

In the previous example, ATG_TA is a maximal pattern.

By this definition, a pattern belongs to the language $\Sigma(\Sigma \cup \{\_\})^*\Sigma \cup \Sigma$.

Lemma 1 (Unicity) *There is a unique maximal pattern in each equivalence class of $\equiv_t$.*

2.2 Non-redundancy in a set of patterns

The notion of redundancy will be used in the negative sense in the following:

Definition 2 (Non-Redundant pattern) *Let M be a set of patterns occurring in a text t. The pattern $m \in M$ is non-redundant in M if there do not exist patterns $m_i \in M$, $m_i \neq m$, such that $pos(m,t) = \cup_{i>1} pos(m_i,t)$.*

2.3 Non-redundant maximal q-patterns

Parida et al. [10] consider the patterns that are non-redundant in the set of all the maximal q-patterns.

Definition 3 (Irredundant maximal q-pattern [10]) *A maximal q-pattern m is redundant if there exist maximal q-patterns m_i, $m_i \neq m$, such that $pos(m,t) = \cup_{i>1} pos(m_i,t)$.*

In the text t = AAAAAXAAAAA, the maximal non-redundant 2-patterns are: AA, AAA, AAAA, AAAAA, A_AAA_A, AA_AA_AA, AAA_A_AAA, and AAAA__AAAA. The maximal pattern AAA__AAA is redundant because $pos(AAA\_\_AAA,t) = \{1,2,3\}$ and $pos(AAAA\_\_AAAA,t) \cup pos(AAA\_A\_AAA,t) = \{1,2\} \cup \{1,3\}$.

2.4 Non-redundant maximal patterns

As Parida et al. define the non-redundant maximal q-patterns, Apostolico [1] defines the non-redundant maximal patterns. In that case, it can be remarked that the definition corresponds to that of Parida in the case $q = 1$.

3 Primitive Patterns

Using the idea of redundancy, we give a new definition, the primitive patterns, which is more restrictive than the Parida et al. maximal non-redundant definition. The primitive patterns are the maximal non-redundant q-patterns.

3.1 Definition

Definition 4 (Primitive q-pattern [12]) *A maximal q-pattern m is primitive if and only if it is non-redundant in the set of <u>all</u> the q-patterns occurring in the text t.*

In other words, m is primitive if and only if there is no patterns m_i, $m_i \neq m$ and m_i not necessarily maximal, such that $pos(m,t) = \cup_{i>1} pos(m_i,t)$.

In the previous example, there are only five primitive 2-patterns : `AAAAA`, `A_AAA_A`, `AA_AA_AA`, `AAA_A_AAA`, and `AAAA__AAAA`. Using that definition, we may see that the pattern `AAA___AAA` that is redundant using Parida's definition is, also, not primitive (as we kept the same patterns that make this pattern redundant).

Lemma 2 *Let m be a q-pattern in a text t. If m is primitive, then m is maximal non redundant.*

The proof is straightforward : let m be a primitive q-pattern, then there are no m_i such that $pos(m,t) = \cup_{i>1} pos(m_i,t)$, which means, of course, that there are no maximal q-patterns m'_i such that $pos(m,t) = \cup_{i>1} pos(m'_i,t)$.

We denote the set of all the primitive q-patterns in the text t by $\mathcal{P}(t,q)$.

3.2 Pattern fusion

The pattern fusion ($\oplus$) is an operation on two patterns that watches the patterns letter by letter, keeping only common letters, and replacing a difference by a don't care. For example, $ABB\_CD \oplus AC\_CC = A\_\_C\_$.

Definition 5 *Let m_1, m_2 be two patterns. The pattern $m = m_1 \oplus m_2$ is such that for all $i, 1 \leq i \leq min\{|m_1|, |m_2|\}$, $m[i] = m_1[i]$ if $m_1[i] = m_2[i]$, and $m[i] = \_$ otherwise.*

That operation is trivially commutative and associative. The obtained pattern 'tends' to be maximal : all the jokers are present because there is a conflict between two letters on a same position. Some jokers may be present at the beginning of the pattern, or at the end. The operation $\oplus^{max}$ is an extension of the operation $\oplus$, but returns a pattern with no heading or tailing jokers.

Definition 6 *Let m_1, m_2 be two patterns. The pattern $m = m_1 \oplus^{max} m_2$ is such that $m[1], m[|m|] \in \Sigma$, and there exist $u, v \in \{\_\}^*$ such that $umv = m_1 \oplus m_2$.*

These operations are on 'abstract' patterns. We now focus on the text, from which the patterns are extracted, by considering positions of the text, and the corresponding suffix.

Given a set of positions P, we denote by $P_{norm} = P - min\{p_i \in P\} + 1$ the 'normalized' of P (remark that $|P| = |P_{norm}|$, and $1 \in P_{norm}$).

Definition 7 *Let $P = \{p_1, p_2, \cdots p_k\}$ be a set of positions from the text. The pattern $m = \bigoplus_t P$ is such that $m = \oplus_{p_i \in P_{norm}} suff(p_i, t)$.*

And as for the $\oplus$ operation that is extended by $\oplus^{max}$, we define $\bigoplus_t^{max}$ from $\bigoplus_t$:

Definition 8 *Let $P = \{p_1, p_2, \cdots p_k\}$ be a set of positions from the text. The pattern $m = \bigoplus_t^{max} P$ is such that $m[1], m[|m|] \in \Sigma$, and there exist $u, v \in \{\_\}^*$ such that $umv = \bigoplus_t P$*

Theorem 1 *Let t be a text. m is a maximal pattern if and only if there exists a set of positions P, such that $m = \bigoplus_t^{max} P$.*

Proof 1 *Let m be a maximal pattern in t, and $P = pos(m, t)$. Let $m' = \bigoplus_t P$. As $P = P_{norm} + \delta$, m' can be written as $m' = umv$, with $u, v \in \{\_\}^*$. Then is it clear that there exists P such that $m = \bigoplus_t^{max} P$.*

Let P be a set of positions in t, and $m' = \bigoplus_t P$. As m is build using the normalized set of positions, there is no pattern m'', $m'' \equiv_t m'$, such that $w(m'') > w(m')$. Once we remove from m' the tailing and heading jokers, there is still no pattern m'', $m'' \equiv_t m'$, such that $w(m'') > w(m')$. But in that case, the pattern is such that its first and last characters are solid letters, i.e. it is the shortest with the highest weight in its equivalence class : the pattern $m = \bigoplus_t^{max} P$ is maximal.

□

3.3 Characterization

In the following, we give two definitions in order to characterize the primitive patterns.

Covering pattern

Definition 9 (Covering pattern) *Given a text t and two maximal patterns m and m', m' is said to cover m if and only if there exists $\delta \geq 0$ such that $pos(m', t) + \delta \subseteq pos(m, t)$.*

We will say that a pattern m' covers m on a position p if $p \in pos(m, t)$ and p belongs to the subset of positions covered by m'.

It is clear that the covering relation is reflexive, transitive and antisymmetric (except if $m = m'$), but is not defined for every maximal pattern m and m'.

Hearts

A pattern may have a lot of positions, but only some of them may be sufficient to retrieve the letters composing the pattern. We call heart a subset of the position list of a given pattern such that this subset allows to build the pattern.

Definition 10 (Heart) *Let m be a pattern, and $P \subseteq pos(m, t)$. P is a heart of m, $heart(m, t)$, if and only if $m = \oplus_t^{max} P$.*

A natural consequence is that m must be a maximal pattern, since there exists a set of positions P such that $m = \bigoplus_t^{max} P$ (see theorem 1).

Lemma 3 *Let m be a pattern, and $P \subset pos(m,t)$. If P is a heart of m, then for all $P' \subset pos(m,t)$, $P \cap P' = \emptyset$, we have $P \cup P'$ is a heart of m.*

Characterization

Theorem 2 (Primitive pattern characterization) *Let t be a text, and $q \geq 1$. Let M be the set of all maximal q-patterns of the text t. The five following propositions are equivalent:*

1. *$m \in M$ is a primitive q-pattern;*
2. *there exists a position p^*, $p^* \in pos(m,t)$, such that no pattern $m' \in M, m \neq m'$, covers m on p^*;*
3. *there exists a position p^*, $p^* \in pos(m,t)$, such that $\forall P$, $P \subseteq pos(m,t)$ with $p^* \in P$ and $|P| = q$, we have $m = \bigoplus_t^{max} P$;*
4. *there exists a position p^*, $p^* \in pos(m,t)$, such that $\forall P$, $P \subseteq pos(m,t)$, $p^* \in P$ and $|pos(\bigoplus_t^{max} P, t))| \geq q$, we have $m = \bigoplus_t^{max} P$;*
5. *there do not exist patterns $m_i \in M$, $m_i \neq m$, and associated shifts $\delta_i \geq 0$ such that $pos(m,t) = \cup_{i>1}(pos(m_i,t) + \delta_i)$.*

We call such a position p^* a characteristic position for the primitive pattern m.

Proof 2 $1 \Rightarrow 2$ *Let m be a primitive q-pattern. Then there do not exist patterns m_i (even not maximal) such that $pos(m,t) = \cup_{i>1} pos(m_i,t)$. It means that there exists at least one position not covered by any other pattern.*

$2 \Rightarrow 3$ *Let $p^* \in pos(m,t)$ such that no pattern $m' \in M, m \neq m'$, covers m on p^*. Assume that there exists $P \subseteq pos(m,t)$, $|P| = q$, and $p^* \in P$: it is clear that $m' = \bigoplus_t^{max} P$ is a maximal pattern, with $|pos(m',t)| \geq q$, and m' covers m on p, which is a contradiction.*

$3 \Rightarrow 4$ *Let $p^* \in pos(m,t)$ such that $\forall P \subseteq pos(m,t)$, $p^* \in P$, $|P| = q$, we have $m = \bigoplus_t^{max} P$. Then it is clear that $\forall P \subseteq pos(m,t)$, $p^* \in P$ and $|P| > q$, we have $\bigoplus_t^{max} P = m$ (all the set containing p^*, and having exactly q positions are hearts of m, then adding positions of m cannot form another pattern than m (cf. lemma 3)). If $|P| < q$ and $|pos(\bigoplus_t^{max} P, t)| \geq q$, then as every $P' \subseteq pos(m',t)$, $|P'| = q$, is such that $m = \bigoplus_t^{max} P'$, we have clearly $m' = m$ (cf. 3). Consequently, there exists $p^* \in pos(m,t)$ such that $\forall P \subseteq pos(m,t)$, $p^* \in P$ and $|pos(\bigoplus_t^{max} P, t))| \geq q$, we have $m = \bigoplus_t^{max} P$.*

$4 \Rightarrow 5$ *Assume there exists $p^* \in pos(m,t)$ such that $\forall P \subseteq pos(m,t)$, $p^* \in P$ and $|pos(\bigoplus_t^{max} P, t))| \geq q$, we have $m = \bigoplus_t^{max} P$. It means that no pattern $m' \in M$ is such that $pos(m',t) + \delta \subset pos(m,t)$, with $p - \delta \in pos(m',t)$. Then, clearly, we cannot find maximal patterns m_i and $\delta_i \geq 0$ such that $pos(m,t) = \cup_{i>1}(pos(m_i,t) + \delta_i)$.*

$5 \Rightarrow 1$ *Assume there do not exist patterns* $m_i \in M$ *and associated shifts* $\delta_i \geq 0$ *such that* $pos(m,t) = \cup_{i>1}(pos(m_i,t) + \delta_i)$. *Then, there do not exist non-maximal patterns* m'_i *such that* $pos(m,t) = \cup_{i>1} pos(m'_i,t)$*: Assume we can find such patterns* m'_i*, then for each non-maximal pattern* m'_i*, there exists a maximal pattern* m_i*,* $m_i \equiv_t m'_i$*, and* $\delta \geq 0$*, which is a contradiction with the hypothesis.*

□

It can be remarked that there can exist more than one characteristic position for a given pattern m.

The example below shows a 2-pattern, `G__TA`, that is not primitive since it is covered by two 2-patterns, `ATG__TA` and `GC_TA`, because we have $pos(G\_\_TA,t) = (pos(ATG\_\_TA,t)+2) \cup pos(GC\_TA,t)$.

```
GACATGATTATTGATGCCTAACCGGCTTATGGATAGC
   ATG__TA   ATG__TA        ATG__TA
               GC_TA    GC_TA
     G__TA     G__TA    G__TA G__TA
```

3.4 Patterns and quorum

As the primitive patterns are now defined, it is important to discuss the role of the quorum. Looking for a pattern in a set of sequence leeds naturally to define a thresold that fixes the minimal number of sequence in which the pattern shall be found to be taken into account. However, in a single sequence, the interest of the quorum may not be clear.

In fact, depending on the value of the quorum q, a maximal pattern can be considered as primitive or not.

Lemma 4 (Pattern's life) *Let* t *be a text, and* m *a maximal pattern that occurs in* q_m *positions. There exists an integer* $q_p \leq q_m$ *such that :*

- *for all quorum* $q < q_p$*, the q-pattern* m *is non-primitive,*
- *for all quorum* q*,* $q_p \leq q < q_m$*, the q-pattern* m *is primitive,*
- *and for all quorum* q*,* $q \geq q_m$ *the pattern* m *is not a q-pattern.*

Proof 3 *When we consider a quorum* q *greater than a position list size, the pattern is clearly not a q-pattern. A maximal pattern is either primitive or not, for a given quorum. It is clear that for* $q_p = |pos(m,t)|$*, the pattern is primitive. It is also clear that* q_p *may be lower than* $|pos(m,t)|$*. We denote by* q_p *the lowest quorum such that* m *is primitive. What we may show is that once a pattern becomes primitive for a quorum* q_p*, it may not be non-primitive for any greater quorum* q*,* $q_p \leq q \leq q_m$*.*

The proof is simple: assume there exists $q > q_p$ *such that* m *is not primitive. By theorem 2, all the positions of* m *are covered by patterns of* $\mathcal{P}(t,q)$*. As* m *is primitive for* q_p*, then by the same proposition, it means there exists a position*

$p^* \in pos(m,t)$ such that no other pattern covers m on p^*. Let $m' \in \mathcal{P}(t,q)$ cover m on p^* for quorum q and have a look at quorum q_p. As m is primitive, m' may not be primitive (otherwise, m would not be primitive). But if m' is not primitive for quorum q_p, it means that there exist patterns such that every position is covered, and then a pattern that covers m' should cover m too. This is a contradiction.

□

This lemma could apply to the non-redundant patterns with minor modifications. It shows in particular the interest of the quorum, since the patterns occuring a small number of times hide the other patterns. As an example, the pattern t (the text itself), hides every other pattern (we recall that $\mathcal{P}(t,1) = \{t\}$ [12]). That is why the first interesting quorum is 2.

3.5 Pattern basis

Every q-pattern can be generated by the primitive q-patterns. Interesting patterns are, in fact, the ones which are maximal in composition : patterns that are factors of maximal patterns (patterns such that no don't care can be replaced by just one letter). As their number is huge, we cannot keep them in memory, but we want to be able to generate them from the primitive patterns.

A pattern m is generated by $\oplus$ operation.

For a given integer $q > 1$ and a text t, every pattern m, that is maximal in composition, can be written as $m = \oplus_{i>0} m_i[k_i, l_i]$, where m_i are primitive q-patterns. The primitive q-patterns are free, that is to say no primitive q-pattern can be generated using other ones. Then, the primitive q-patterns form a basis for the maximal in composition patterns that occur at least q times in t.

3.6 Number of primitive patterns

Given $\delta \geq 1$, $t \oplus^{max} t[\delta]$ is the unique maximal pattern that occurs at least in two positions p and $p + \delta$.

This leads to a bound of the primitive 2-pattern number :

Lemma 5 *The set $\cup_{\delta=2,\dots|t|} t \oplus^{max} t[\delta]$ contains all the primitive 2-patterns.*

Proof 4 *Let E be the list of the patterns $t \oplus^{max} t[\delta]$. By theorem 1, every pattern in E is maximal.*

Assume now that there exists a pattern m, primitive, but not in E. Let us denote by p and p' two positions of m, and $\delta = |p - p'|$, then there exists $m' = t \oplus^{max} t[\delta]$, $m' \in E$. Then as m' is maximal, there exists $s \geq 0$ such that $p - s$ and $p' - s$ belong to $pos(m', t)$. That means that m' covers m on p and p'.

Then, either $|pos(m,t)| = 2$ and then m is not maximal, or for all $p, p' \in pos(m,t)$, there exists a pattern m_i such that m_i covers m on p and p', which means that m is not primitive.

□

Corollary 1 (Bound on the primitive 2-patterns) *The number of primitive 2-patterns is bounded by* $n-1$.

In [10], Parida et al. give a linear bound for the non-redundant q-patterns in a text. Since the number of primitive patterns is lower than the number of non-redundant patterns, we could think that the number of primitive patterns are bounded by $3n$. However the following example shows that this is not the case. In the text $A^k X A^k$, with $k = 9$ and $q = 5$, we have 112 primitive 5-patterns, and as k grows, the number of primitive patterns grows too. In section 5, we show there is a strong relation between the number of primitive patterns and C_{n-1}^{q-1}.

Lemma 6 (Upper bound) *Given a text* t *and a quorum* q*, there exist at most* $C_{|t|-1}^{q-1}$ *primitive q-patterns in* t.

Proof 5 *The idea is a consequence of our theorem 2, and particularly the third proposition: a pattern* m *is q-primitive if and only if there exists a set* $P \subseteq pos(m,t)$ *such that* $|P| = q$ *and* P *contains a characteristic position. Thus, if all the normalized sets containing q positions are build, we have all the q-primitive patterns. There are exactly* $C_{|t|-1}^{q-1}$ *distinct sets (the normalized sets contain the position* 1*, we have to choose* $q-1$ *among* $|t|-1$*).*

□

An other example is given by Pisanti et al. [13] and shows that the $3n$ bound is not correct. In this paper, Pisanti et al. show that there exists an infinite family of strings for which the number of primitive patterns is at least $C_{\frac{n-1}{2}-1}^{q-1}$, and give the same upper bound.

4 Algorithm for primitive pattern extraction

The primitive q-patterns may be obtained as follow :

Lemma 7 (From $\mathcal{P}(t,q-1)$ **to** $\mathcal{P}(t,q)$**)** *Let* m *be a primitive pattern for* q*,* $q \geq 2$ *(*$m \in \mathcal{P}(t,q)$*). If* $m \notin \mathcal{P}(t,q-1)$*, then there exists a maximal pattern* m'*,* $|pos(m',t)| = q-1$ *and a position* $p \notin pos(m',t)$ *such that* $m = m' \oplus^{max} suff(p,t)$.

Proof 6 *If* $m \notin \mathcal{P}(t,q-1)$*, then* m *is not primitive and there exist patterns* $m_i \in \mathcal{P}(t,q-1)$ *such that* m *is covered by* m_i*. As* $m \in \mathcal{P}(t,q)$*, it means that one position of* m *that was covered for* $\mathcal{P}(t,q-1)$ *is not anymore : there exists a pattern* m' *from* $\mathcal{P}(t,q-1)$ *that is not taken into account (*$|pos(m',t)| = q-1$*).* $m' \notin \mathcal{P}(t,q)$*, then* $|pos(m',t)| < |pos(m,t)|$*: there exists a position* p *such that* m *is not covered by* m' *on* p*. Let* $\delta \geq 0$ *such that* $pos(m',t) + \delta \subset pos(m,t)$*. Let* P *be the set of positions of* m *such that* m *is not covered by* m' *on* $p \in P$.

Let us have a look at the patterns $m_p = m' \oplus^{max} suff(p,t)$, $\forall p \in P - \delta$. *Either there exists* $p \in P - \delta$ *such that* $m_p = m$, *or* $\forall p \in P - \delta, m_p \neq m$ *(that is to say, adding just one position was not enough to build* m*). Assume no* $p \in P - \delta$ *is such that* $m_p = m$. *In this case, there is more than one position in* P, *and* $\forall p \in P - \delta, m_p$ *belongs to the basis* $\mathcal{P}(t,q)$. *It means that* m *is non-primitive since* $pos(m,t) = \cup_{p \in P-\delta}(pos(m_p,t))$.

□

Note that a pattern m can be obtained from a pattern m', $m' \in \mathcal{P}(t, q-1)$ and $m' \in \mathcal{P}(t,q)$. But in the case where m is primitive, it can be easily shown that there exists a pattern m'', $|pos(m'',t)| = q-1$, such that m is also obtained by adding a position.

The lemma justifies the incremental approach for computing the basis for a given quorum. Informally speaking, the idea is:

1. generate hearts of possible primitive q-patterns, by taking the primitive (q-1)-patterns that have exactly $q-1$ positions, and by adding a new position that is not already in the position list of the given pattern.
2. all the primitive (q-1)-patterns, that have more than $q-1$ positions, are immediately primitive;
3. for each generated pattern, compute its position list : if the size is exactly q, the pattern is primitive, otherwise test the primitivity.

The algorithm ExtractPrimitiveQPatterns $(t, q, \mathcal{P}(t, q-1))$ computes incrementaly the primitive patterns for a quorum q, given the primitive patterns for the quorum $q-1$:

```
ExtractPrimitiveQPatterns(text, q, P(t, q-1))
 1  ▷ Input :
    ▷    the text, the quorum q, the primitive (q-1)-patterns
 2  ▷ Output : result, a trie containing all the primitive q-patterns
 3  result ← NewTrie()
 4  for each pattern m ∈ P(t, q-1), |pos(m, t)| = q-1 do
 5          for shift ← 1 to |text| do
 6                  if shift ∉ pos(m, t) then
 7                          pattern ← m ⊕^max suff(shift, text)
 8                          Push(stack, pattern)
 9                          Insert(result, pattern)
10  while stack ≠ ∅ do
11          pattern ← Pop(stack)
12          pos ← ComputePatternPositions(text, pattern)
13          if TestPrimitivity(pattern, t, q) then
14                  ▷ The pattern is primitive
15                  Affect pos to the final state
16          else    Remove(pattern, result)
17                  ▷ The pattern is not primitive
18  return result
```

In order to store the created patterns, we use a trie, in which the insertion/removal of a word of size n costs $O(n)$. Once the position list is computed, it is associated to the corresponding final state.

Note that for quorum $q = 2$, it is as if we computed all the $t \oplus^{max} suff(\delta, t)$ patterns, for all δ between 2 and $|t|$;

In order to compute the position list of pattern build by a given heart, we need to obtain the pattern itself (the letters composing the pattern): the operation $\oplus^{max}$ is done in time $O(n)$.

Two approaches are possible for testing the primitivity: First, we may use the way patterns are extracted to decide if they are primitive or not (valid for the case $q = 2$, this does not generalize easily). That idea is described in the next section. Second, we may use the way position lists can be encoded as integers, and then use multiplications to discard covered positions. The section 6 describes the algorithm that allows to check the primitivity of a pattern in time $O(|\Sigma| n \log^2 n \log\log n)$.

5 Using the hearts

Removing non-primitive patterns is done using the hearts that have been studied during the extraction.

We will use a simple counting process for the removal step : a counter $c(p, m)$ is associated to each position for each pattern. Using the theorem 2, and particularly the idea of the third proposition, we have the following accounting scheme:

Lemma 8 *If a pattern m is primitive for a quorum q, a characteristic position p^* can be found in $C^{q-1}_{|pos(m,t)|-1}$ sets P such that $|P| = q$ and $p^* \in P$.*

The number is obtained by fixing the first element of P with p^*, and then we just have to find $q-1$ positions among the remaining $|pos(m, t)| - 1$.

As for the case $q = 2$, we generate all the possible 2-uplets, and $C^{q-1}_{|pos(m,t)|-1} = |pos(m, t)| - 1$, we just have to count a position in $|pos(m, t)| - 1$ hearts to say that a 2-pattern is primitive. The counters are updated each time a pattern is extracted, in constant time. Still for the case $q = 2$, checking the counters is done in time $O(n)$ for each pattern, that is in time $O(n^2)$ at most: this is the same complexity as the generation step (it does not dominate the time complexity).

In that special case, the algorithm ExtractPrimitiveQPatterns $(t, q, \mathcal{P}(t, q-1))$ becomes ExtractPrimitive2Patterns (t). We associate to each pattern the counters, that are automatically updated as a new heart of the pattern is found.

```
ExtractPrimitive2Patterns(text)
 1  ▷ Input : the text
 2  ▷ Output : result, a trie containing all the primitive q-patterns
 3  result ← NewTrie()
 4  for shift ← 2 to |text| do
 5          pattern, δ ← m ⊕^max suff(shift, text)
 6          ▷ δ ≥ 0 is the shift between
            ▷    m ⊕^max suff(shift, text) and m ⊕ suff(shift, text)
 7          hearts(m) ← hearts(m) ∪ {1 + δ, shift + δ}
 8          Push(stack, pattern)
 9          Insert(result, pattern)
10  while stack ≠ ∅ do
11          pattern ← Pop(stack)
12          pos ← ComputePatternPositions(text, pattern)
13          if MaxCount(hearts(m)) = q − 1 then
14                  ▷ The pattern is primitive
15                  Affect pos to the final state
16          else    Remove(pattern, result)
17                  ▷ The pattern is not primitive
18  return result
```

That idea does not generalize trivially, since two possibilities do not appear in the case $q = 2$: we may have patterns that are primitive in $\mathcal{P}(t, q)$ and in $\mathcal{P}(t, q-1)$ (which complicates the incrementation of the counter); and we may have patterns extracted for $\mathcal{P}(t, q-1)$, and declared to be non primitive (and then, some of the q-uplets are not formed). In the present version of this extended abstract, we prefered to describe a more simpler algorithm (given in the next section) for doing the primitivity test.

The lemma given above, and the lemma 6 lead to an other approach to compute the primitive q-patterns. It is possible to generate all the sets P, with $1 \in P$ and $|P| = q$. Then for each set, we compute the corresponding maximal pattern, its position list, and we decide whether it is primitive or not. Considering a set, the two steps (computing the maximal pattern, and its position list) can be done in time $O(|\Sigma| n \log^2 n \log\log n)$ using the technique we describe in the following section. The decision of the primitivity is a counting process given by the previous lemma. Such an algorithm would have a complexity in time $O(C_{|t|-1}^{q-1} |\Sigma| n \log^2 n \log\log n)$.

6 Algorithm for deciding the primitivity of a pattern

In this section, we introduce a new operation on sets that can be done using a simple translation of the sets into integers that are to be multiplied. This is the main idea of the Fisher and Paterson's algorithm [5] for computing the position list of a pattern with don't cares in time $O(|\Sigma| n \log^2 n \log\log n)$.

Then, we explain how it is possible to decide, given only a text, a pattern and a quorum, whether the pattern is primitive or not.

6.1 Multisets and products

A finite set of integers is classically represented with an array of booleans, whose indices belong to a range of integers, where the value at index i is true if and only if the value i is in the set. Integers may also be used to represent a set. For example, the set $E = \{1, 4, 5\}$, can be represented by an array a, in which the indices range from 1 to 6, $a = [true; false; false; true; true; false]$, and an integer 011001.

We introduce the notion of multiset, in which elements are valued. For example, in the multiset $E = \{(1, 5), (4, 2), (5, 1)\}$, the element 4 is valued 2. We use the value as the number of time the element is present in the set. A multiset can also be represented with an array, of integer this time. As an example, we may have $E = \{(1, 5), (4, 2), (5; 1)\}$ represented as $a = [5; 0; 0; 2; 1; 0]$, and the corresponding integer 012005.

The operation $\textsc{toInteger}(E, i, j)$ translates the set E, in which the values are bounded by i and j into an integer where the rightmost coefficient stands for the value of the bound i. The reverse operation, $\textsc{toMultiSet}(k, i)$, translates the integer k into a multiset with lower bound i.

Adding a constant to a multiset consists in adding the constant to each value: $E + c = \{x + c/x \in E\}$. Realized on an integer, that operation consists in multiplying the integer by b^c, where b is tha basis of the integer.

In the following, we need to produce the union/intersection of a multiset with itself, with a shift of its elements. For example, $E \cup (E + 1) \cup (E + 4) = \{(1, 5), (4, 2), (5, 1)\} \cup \{(2, 5), (5, 2), (6, 1)\} \cup \{(5, 5), (8, 2), (9, 1)\}$. That is $E \cup (E + 1) \cup (E + 4) = \{(1, 5), (2, 5), (4, 2), (5, 8), (6, 1), (8, 2), (9, 1)\}$. We denote by $\odot$ the operation that takes a multiset in the left side, and a set of shifts in the rightside, and computes the union of the multiset with itself at the different shifts: $E \odot \{0, 1, 4\} = E \cup (E + 1) \cup (E + 4)$.

As we use multisets, the result contains both the union and the intersection: the value associated to the element contains the number of time the element is present in the sets. If we set the values to 1 before doing the $\odot$ operation, for each element in the result, we know in how many shifts the element was present. On the example, $E \odot \{0, 1, 4\} = \{(1, 1), (4, 1), (5, 1)\} \odot \{0, 1, 4\} = \{(1, 1), (2, 1), (4, 1), (5, 3), (6, 1), (8, 1), (9, 1)\}$. The element 1 is present in exactly one shift. The element 5 is present in all the three shifts, and thus in the intersection of the three sets.

Since the number of shifts can be equal to the number of elements in the set, this operation can be done with a $O(n^2)$ time algorithm with n the size of the set, and the number of shifts/union to do.

But it is possible to write the operations with integers: $E \odot \{0, 1, 4\}$ can be translated into $\textsc{toInteger}(E, 1, 6) + \textsc{toInteger}(E, 1, 6) \times b^1 + \textsc{toInteger}(E, 1, 6) \times b^4 = 011001 + 0110010 + 0110010000 = 0110131011$, where b is the basis used to encode the integers. Which is in fact a simple product of 011001 by 10011.

It is known that a product of integers of k bits can be done with a $O(k \log k)$ time algorithm, using for example the Schönhage-Strassen algorithm [14] (there

are numerous fast product algorithms. A technique commonly used is the Fast Fourier Transform).

The basis in which integers are written has to be carrefully chosen, since no retenue shall be propagated in the product. A good choice is to have the basis $\log n$, where n is the maximal integer encoded. Thus, the integer is encoded on $n \log n$ bits, and the product is computed in time $O(n \log^2 n \log\log n)$.

6.2 Fisher and Paterson's algorithm

The algorithm [5] consists in computing the positions of a pattern with don't cares in a given text t.

In exact pattern matching, the positions of the occurrences of a pattern m in a text t can be seen as an intersection of shifted letter positions: $pos(m,t) = \cap_{i=1}^{|m|}(pos(m[i],t) - i - 1)$. This gives a naive time $O(|m||t|)$ algorithm for computing all the positions of m in t. For example, with $t = AAATAAGTAGAT$, and $m = AA$, we have $pos(m,t) = pos(A) \cap (pos(A) - 1) = \{1,2,3,5,6,9,11\} \cap \{1,2,4,5,8,10\} = \{1,2,5\}$.

Adapted to pattern matching with don't cares, this could be written as $pos(m,t) = \cap_{i=1,m[i]\neq\_}^{|m|}(pos(m[i],t) - i - 1)$, which is still a quadratic algorithm. With the same text, and the pattern $m = AA\_\_AG$, we have $pos(m,t) = pos(A,t) \cap (pos(A,t)-1) \cap (pos(A,t)-4) \cap (pos(G,t)-5) = \{1,2,3,5,6,9,11\} \cap \{1,2,4,5,8,10\} \cap \{1,2,5,6\} \cap \{2,5\} = \{2,5\}$.

Therefore, it is possible to do only $|\Sigma|$ intersections, by factorising with the letters of the alphabet : $pos(m,t) = \cap_{c\in\Sigma}(pos(c,t) \odot -(pos(c,m) - 1))$. On the example, we have $pos(AA\_\_AG,t) = (pos(A,t) \odot \{-0,-1,-4\}) \cap (pos(G,t) \odot \{-5\})$.

As $\odot$ can be done by a simple product, we just need to encode the position lists. A recurrent problem is to address: positions in a text are numbered from left to right, and digits in an integer are numbered from right to left. Thus, the notation $\widehat{pos}(m,t)$ will be the set of all the position of m in t, where the numbering of the positions of t begin from the end of the text. As the numbering changes, the negative sign for the shifts disappears. The operation to do becomes $\widehat{pos}(m,t) = \cap_{c\in\Sigma}(\widehat{pos}(c,t) \odot (pos(c,m) - 1))$. We compute $\widehat{pos}(m,t)$ instead of $pos(m,t)$, but it is easy to retrieve $pos(m,t)$.

The algorithm is the following:

1. choose a basis, for example equal to $\log n$ if n is the maximal integer to be represented (the length of the text);
2. for each letter $c \in \Sigma$, compute $P_c = \widehat{pos}(c,t) \odot (pos(c,m) - 1)$ using the integer conversion;
3. do the sum of the Σ products $S = \Sigma_{c\in\Sigma} P_c$;
4. look for every coefficient in S that is equal to the number of fixed letters in the pattern to find the positions.

The time complexity is clearly $O(|\Sigma| n \log^2 n \log\log n)$ (dominated by the step 2).

For example, with the text $t = AAATAAGTAGAT$, and the pattern $m = AA\_\_AG$: $\widehat{pos}(m,t) = (\widehat{pos}(A,t) \odot (pos(A,m)-1)) \cap (\widehat{pos}(G,t) \odot (pos(G,m)-1))$. We do the products $111011001010 \times 10011 = 1111331131111110$ and $100100 \times 100000 = 10010000000$. Then we do the sum, obtaining 1111341141111110. There are two values equal to 4, the number of fixed letters in m, which are at positions 2 and 5 (see figure 1).

t				A	A	A	T	A	A	G	T	A	G	A	T
P_A	1	1	1	1	3	3	1	1	3	1	1	1	1	1	0
P_G						1	0	0	1	0	0	0	0	0	0
S	1	1	1	1	3	4	1	1	4	1	1	1	1	1	0
$pos(m,t)$						1			1						

Fig. 1. The positions of $m = AA\_\_AG$ in $t = AAATAAGTAGAT$.

6.3 Decision algorithm

We propose in the following a method that takes a text t, a maximal pattern m and an integer q for the quorum, and decides whether the pattern is primitive or not. With minor changes it is possible to test for maximality, or non-redundancy.

A pattern can be extended with jokers at the both sides. Each joker, inside and outside the pattern, can be replaced by a letter in each occurrence of the pattern in the text. The idea is to count, for each joker, how many times each letter of the alphabet replaces the joker in the text. If a joker can be replaced more than q times by a same letter c, it means that there exists a pattern that has all the fixed letters of the pattern of interest, plus one, the letter c : it is a covering pattern. In that case, the corresponding positions of m are covered, and thus have to be marked. If all the positions of m in t are marked, it means that all the positions are covered, and then m is not primitive (with theorem 2).

Informally speaking, the algorithm consists in two steps:

- step 1: compute the counters $count_c(z,m,t)$. $\forall z \in [-|t|;|t|], count_c(z,m,t) = |\{p/p \in pos(m,t)+z, t[p] = c\}|$;
- step 2: compute the covered positions. $p \in pos(m,t)$ is covered if there exists z and a letter $c \in \Sigma$ such that $count_c(z,m,t) \geq q$.

In figure 2, we see the text $t = AAATAAGTAGAT$ aligned with it self three times, using each occurrence of the pattern $m = A\_\_T$ as anchor for the alignment. The counters are represented on the same figure. For example, we may remark that the column standing for the A in the pattern counts three times that letter. We also see that the first joker inside the pattern can be replaced twice by a A: there exists a covering pattern (if we consider $q = 2$). There is not enough information to tell what pattern it is, but there is one that look like

z	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7	8	9	10	11
m									A	_	_	T								
$p_1 = 1$									A	A	A	T	A	A	G	T	A	G	A	T
$p_2 = 5$					A	A	A	T	A	A	G	T	A	G	A	T				
$p_3 = 9$	A	A	A	T	A	A	G	T	A	G	A	T								
$count_A$	1	1	1	0	2	2	1	0	3	2	2	0	2	1	1	0	1	0	1	0
$count_T$	0	0	0	1	0	0	0	2	0	0	0	3	0	0	0	2	0	0	0	1
$count_G$	0	0	0	0	0	0	1	0	0	1	1	0	0	1	1	0	0	1	0	0

Fig. 2. All the occurrences of the pattern aligned, with the associated counters.

AA_T. It is not interesting to retrieve the pattern, but just to mark the positions p_1 and p_2.

If $q = 2$, the column -3 allows to mark positions p_2 and p_3, and the column 7 let positions p_1 and p_2 marked. As all the positions are marked, the pattern is not primitive: we have $pos(A\_\_T, t) = (pos(A\_\_A\_T, t) + 3) \cup (pos(A\_T\_\_T, t))$. (we do not show here all the possible covering patterns, since two are sufficient to show that m is not primitive.)

It is worth to remark that this sketch of algorithm takes a maximal pattern in input (all the patterns we have to test are maximal), but we could easily check if the pattern is maximal because if a joker can be replaced by a single letter in each position of the pattern, it is clear that the pattern is not maximal.

Step 1: Compute the counters

First, we will show that if a counter for the letter c is higher than the quorum, it means that there exists a covering pattern. Then, we show how to compute efficiently the counters.

Proposition 1 *If there is a value in the counter of the letter c that higher than q and lower than $pos(m, t)$, there exists a covering pattern for the positions where, in the text, the corresponding joker is replaced by the letter c.*

The proof is, in fact, trivial: in such a case, a pattern m' would occur at some of the positions of m (with eventually a shift), and thus having all the fixed letters of m, plus one other fixed letter. We have clearly $pos(m', t) + \delta \subsetneq pos(m, t)$ and $|pos(m', t)| \geq q$.

Computing the counters is done by $count_c = \widehat{pos}(c, t) \odot (pos(m, t) - 1)$, for a letter $c \in \Sigma$.

Step 2: Compute the covered positions

On the example (fig. 2), we see that the first joker in the pattern A_T can be replaced twice by a A. In order to discover what positions are covered, we could build the pattern AA_T, and look for its positions. As the second joker is also replaced twice by a A, we could build A_AT, and look for its positions, and so

on for each joker that can be replace by a A. This method would give all the covered positions, but with a bad time algorithm.

Instead of computing the positions of the pattern AA_T, we can build the pattern $\_A\_$. The positions we find are at least the ones that AA_T has, and others. We can do so for each pattern (see fig 3), and then build an aggregate pattern, m_A.

$z = -4$	A _ _ _ A _ _ T
$z = -3$	_ A _ _ A _ _ T
$z = 1$	A A _ T
$z = 2$	A _ A T
$z = 4$	A _ _ T A
m_A	A A _ _ _ A A _ A

Fig. 3. the pattern m_A

Then it is possible to apply the algorithm in order to find its positions, $\widehat{pos}(A,t) \odot (pos(A, m_A) - 1)$, but using a different interpretation of the result. In the normal case, we look for values equal to the number of fixed letters, since we want all the letters of the pattern to appear. In our case, we just want that there exists one letter, thus, we are just looking for non null values.

On the example, we build the pattern $m_A = AA\_\_AA\_A$, and the pattern $m_T = \_\_T\_\_\_\_T$ relatively to $A\_\_T$ (see fig. 4). Note that as the positions of m_A cover the positions of m with a shift equal to 4, we have to divide the integer by b^4, that is remove the last four values (or in terms of multiset, we have to add the value 4 to each element).

m	A _ _ T
m_A	A A _ _ _ A A _ A
m_T	_ _ _ T _ _ _ _ _ _ _ T

Fig. 4. Step 2: building patterns

The result of the products are presented in figure 5. Note that to simulate the division, we add four 0s to the integer representing the positions of m (the whole integers obtained after the products are given here).

In figure 5, the underlined values stands for positions of m, and show that all of them are covered.

For each counter, we can easily know what letters belong to covering patterns. We denote by $CovPat(m, c, count_c)$ the operation that build the pattern m_c, in which all the jokers that can be replaced by the letter c are really replaced by it, and the fixed letters in m are removed. We denote by k_c the maximal distance

$pos(\_\_m, t)$	0 0 0 0 $\underline{1}$ 0 0 0 $\underline{1}$ 0 0 0 $\underline{1}$ 0 0 0 0 0 0 0
$\widehat{pos}(A,t) \odot pos(A, m_A) - 1$	1 1 2 2 $\underline{3}$ 2 1 3 $\underline{4}$ 2 3 2 $\underline{3}$ 2 0 1 1 1 1 0
$\widehat{pos}(T,t) \odot pos(T, m_T) - 1$	1 0 0 0 $\underline{1}$ 0 0 0 $\underline{2}$ 0 0 0 $\underline{1}$ 0 0 0 1 0 0 0

Fig. 5. Step 2: products and covered positions

from the first letter of the pattern to the first joker replaced by a letter c, such that the replaced joker is present *before* the first letter of the pattern, $k_c = 0$ if no joker before the first letter of m is replaced.

Such patterns m_c may not occur in the text, but when computing $P_c = (\widehat{pos}(c,t) \odot pos(c, m_c) - 1) + k_c$, if an element is present in the set, whatever is the corresponding value, it means that there is a covering pattern at the corresponding position. The operation $\textsc{ToSet}(E)$ converts a multiset into a set.

Proposition 2 *Let m be a maximal pattern in a text t. We consider the letters $c \in \Sigma$ such that there exists z with $count_c(z, m, t) \geq q$. Let m_c be the patterns $m_c = CovPat(m, c, count_c)$, and $k \geq 0$ the shifts of m_c relatively to m. All the covered positions of m are in the set:*

$$\textsc{ToSet}(\widehat{pos}(m,t)) \cap \textsc{ToSet}(\bigcup_{c \in \Sigma} (\widehat{pos}(c,t) \odot (pos(c, m_c) - 1)) + k_c)$$

Algorithm and Complexity
Finally, the algorithm $\textsc{IsPrimitive}(m, t, q)$, where m is a maximal pattern in the text t, is the following:

1. Step 1: for $c \in \Sigma$, compute $count_c = \widehat{pos}(c,t) \odot (pos(m,t) - 1)$;
2. Step 2:
   (a) for $c \in \Sigma$, compute $m_c = CovPat(m, c, count_c)$, and k_c;
   (b) for $c \in \Sigma$, compute $E_c = \widehat{pos}(c,t) \odot (pos(c, m_c) - 1)$;
   (c) convert the multisets E_c in sets S_c (forget the values associated to each element);
   (d) compute $I = \widehat{pos}(m,t) \cap (\cup_{c \in \Sigma}(S_c + k_c))$;
   (e) if $|I| = |pos(m,t)|$, the pattern is not primitive, it is primitive otherwise.

Theorem 3 (Complexity) *The complexity in time for deciding whether a pattern is primitive or not in a text t of size n with quorum q is $O(|\Sigma| n \log^2 n \log\log n)$.*

Proof 7 *The first step consists in Σ products. The step (2.a) allows to build patterns of size $O(n)$ in linear time. The step (2.b) computes the product of each pattern. Finally, as the $|\Sigma| + 1$ intersections are computed using additions of integers, it is clear that the complexity is dominated by the $|\Sigma|$ products.*

□

A slightly modified version of this algorithm can be used to find the letters composing a pattern, given a text and a set of positions : once the position list if computed, we can either keep the letters of the pattern, or the position list (or just an heart of the pattern, to reduce memory space), since one can be computed from the other.

7 Complexity

Theorem 4 *For a quorum $q \geq 2$, the algorithm* ExtractPrimitiveQPatterns *for extracting the primitive q-patterns from a text t of size n takes a time $O(N|\Sigma|n^2 \log^2 n \log\log n)$, where N is the number of primitive (q-1)-patterns.*

Proof 8 *The first step of* ExtractPrimitiveQPatterns *for generating the pattern generation takes $O(Nn^2)$ time as for each if the N patterns that are $(q-1)$ primitive, we study the n positions, and we need to compute the letters compising the pattern, and to insert it into our data structure.*

The second step, which consists in computing the set of all the positions of all the extracted patterns, dominates the overall complexity. Computing the position list of each extracted pattern takes $O(|\Sigma|n \log^2 n \log\log n)$ time using Fischer and Paterson's [5] algorithm. As we have $O(Nn)$ patterns to study, this step takes $O(N|\Sigma|n^2 \log^2 n \log\log n)$.

The last step consists in deciding whether a pattern is primitive or not: this can be done in time $O(|\Sigma|n \log^2 n \log\log n)$. This has to be done for each pattern having more than q positions, that is $O(N|\Sigma|n^2 \log^2 n \log\log n)$.

□

Corollary 2 *For the case where $q = 2$, the algorithm* ExtractPrimitive2Patterns *for extracting the primitive 2-patterns from a text t of size n takes a time $O(|\Sigma|n^2 \log^2 n \log\log n)$.*

8 Conclusion and perspectives

We give an algorithm to compute incrementally the primitive patterns in time $O(N|\Sigma|n^2 \log^2 n \log\log n)$, with N the number of primitive patterns for the previous quorum. As we seen, the number N of primitive patterns, at worst, is not linear (a bound is given in [13]). Therefore, it could be useful to study the average case, and the properties of the texts for which there is not such an explosion. Preliminary tests show that this approach could be of interest.

9 Acknowledgments

The authors would like to thank the reviewers for their remarks and corrections. We would like to thank the bio-informatics team at Exonhit. Thanks are given also to Laxmi Parida, who answered to a lot of questions and requests for examples, and to Gérard Duchamp for the useful talk we had about the product.

References

1. A. Apostolico. Pattern discovery and the algorithmics of surprise. In P. Frasconi and R. Shamir, editors, *Proceedings of the NATO ASI on Artificial Intelligence and Heuristic Methods for Bioinformatics*, October 2001.
2. A. Apostolico and L. Parida. Compression and the wheel of fortune. In *Proceedings of Data Compression Conference (DCC)*, Snowbird, Utah, march 2003.
3. M. Crochemore, C. Hancart, and T. Lecroq. *Algorithmique du Texte*. Vuibert, 2001.
4. M. Crochemore and M.-F. Sagot. Motifs in sequences: localization and extraction. In A. Konopka and al., editors, *Handbook of Computational Chemistry*. Marcel Dekker, Inc, 2001.
5. M. J. Fischer and M. S. Paterson. String matching and other products. *SIAM-AMS proceedings*, pages 113–125, 1974.
6. I. Jonassen, J. Collins, and D. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science*, pages 1587–1595, 1995.
7. C. Lawrence, S. Altschul, M. Boguski, J. Liu, A. Neuwald, and J. Wootton. *Science*, volume 262, page 208. 1993.
8. L. Marsan and M.-F. Sagot. Extracting structured motifs using a suffix tree - Algorithms and application to consensus identification. In S. Minoru and R. Shamir, editors, *Proceedings of the 4th Annual International Conference on Computational Molecular Biology (RECOMB)*, Tokyo, Japan, 2000. ACM Press.
9. B. Morgenstern, A. Dress, and T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. In *Proceedings of the National Academy of Sciences USA*, pages 12098–12103, 1996.
10. L. Parida, I. Rigoutsos, A. Floratos, D. Platt, and Y. Gao. Pattern discovery on character sets and real-valued data: linear bound on irredundant motifs and an efficient polynomial time algorithm. In *Proceedings of the 11th Symposium on Discrete Algorithms*, pages 297–308, 2000.
11. J. Pelfrêne. Indexation de motifs approchés. rapport de DÉA, september 2000.
12. J. Pelfrêne, S. Abdeddaïm, and J. Alexandre. Un algorithme d'indexation de motifs approchés (poster and short talk). In *Journées Ouvertes Biologie Informatique Mathématiques, Saint-Malo*, pages 263–264, June 2002.
13. N. Pisanti, M. Crochemore, R. Grossi, and M.-F. Sagot. Bases of motifs for generating repeated patterns with don't cares. Technical report, Università di Pisa, february 2003.
14. A. Schönhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computer (Arch. Elektron. Rechnen)*, 7:281–292, 1971.
15. G. Schuler, S. Altschul, and D. Lipman. *Proteins: Structure, Function, and Genetics*, volume 9, pages 180–190. 1991.
16. J. Wang, B. Shapiro, and D. Shasha. *Pattern Discovery in Biomolecular Data*. Oxford University Press, 1999.
17. M. Waterman and R. Jones. *Methods in enzymology*, page 221. Academic Press, London, 1990.