Repetitions in Two-Pattern Strings

František Franěk[1] **Weilin Lu**[1] **W. F. Smyth**[1,2]

[1] **Algorithms Research Group**
Department of Computing & Software
McMaster University
Hamilton, Ontario, Canada L8S 4L7

[2] **School of Computing**
Curtin University
GPO Box U-1987
Perth WA 6845, Australia

ABSTRACT

A recent paper shows that it can be determined in $O(n)$ time whether or not a given string x of length n is a substring of an infinite Sturmian string; further, if x is such a substring, the repetitions in x can be computed in $\Theta(n)$ time, generalizing a similar result for Fibonacci strings. This paper extends these results to "two-pattern" strings formed recursively from concatenations of strings $p^i q$ and $p^j q$, where p and q are so-called "suitable patterns". Sturmian strings thus constitute the special case of two-pattern strings when $p = a$, $q = b$, $|j–i| = 1$, while Fibonacci strings constitute the special case of Sturmian strings when $i = 1$. Thus we significantly extend the class of strings whose repetitions can be calculated in linear time. This result is a part of the ongoing two-pronged research effort to identify and describe the class of strings whose repetitions can be determined in linear time, and to show (or refute) that, in general, repetitions in any string can be listed in a list of linear length using the succinct notation of "runs" even though the list itself may not be possible to calculate in linear time.

Keywords: Repetition, Two-Pattern String, Suitable Pair of Patterns, λ-Canonical Reduction, Linear Configuration.

1. INTRODUCTION

It was recently shown in [4] that repetitions in Fibonacci strings can be calculated in linear time. This result was generalized to Sturmian strings, of which Fibonacci strings form a proper subclass, in [3]. Also, it was shown in [1] that finite Sturmian strings can be recognized in linear time. In this paper, we generalize these results to so-called "two-pattern" strings, of which block-complete Sturmian strings form a proper subclass.

Given a string $x = x[1..n]$ of length n, we will show that $O(n)$ time is required to decide whether or not a given string x is a two-pattern string. Further, in the case that x is indeed two-pattern, we will show that all the repetitions in x can be computed in $\Theta(n)$ time.

We begin by defining "suitable patterns":

Definition 1.1 *A binary string q is said to be p-***regular** *if corresponding to a binary string p, there exist binary strings $u \neq \varepsilon$ and v, and integers $l \geq 0$, $m \geq 1$ such that $q = up^l vp^{n_1} \cdots up^l vp^{n_m} u$ where $|\{n_1, \cdots, n_m\}| \leq 2$ (i.e. $n_1 \cdots n_m$ can attain at most 2 distinct values) and if $l = 0$, then $v = \varepsilon$.*

Definition 1.2 *An (ordered) pair of binary strings p, q is said to be a* **suitable pair of patterns** *if*

(1) p is primitive, i.e. has no non-trivial borders,
(2) p is neither a prefix nor a suffix of q,
(3) q is neither a prefix nor a suffix of p,
(4) q is not p-regular.

Definition 1.3 *A* **morphism** *is a mapping $\sigma : a \rightarrow p^i q,\ b \rightarrow p^j q,\ 0 < i < j,$ where p and q are a suitable pair of patterns. (We also call σ an* **expansion***.) We use the notation $[p, q, i, j]$ for σ and $[p, q, i, j]_\lambda$ to indicate in addition $|p| \leq \lambda$ and $|q| \leq \lambda$.*

Definition 1.4 *Let $\Sigma = \{\sigma_1, \sigma_2, \ldots \sigma_n\}$ denote an* **expansion sequence***, where the choices of p, q, i and j for distinct expansions σ_k and $\sigma_{k'}$ are not required to be the same. Suppose that $x = \sigma_n \cdots \sigma_2 \cdot \sigma_1(a)$. Then x is a* **two-pattern string of scope** λ

if each $\sigma_k : a \to p^i q, b \to p^j q$ satisfies $|p|, |q| \le \lambda$.

Note that the use of the scope does not really restrict the generality of the problem, since all of the possible two-pattern strings with scope λ are included in the set of two-pattern strings with scope $\lambda+1$.

2. THE RECOGNITION ALGORITHM

We begin by outlining our approach to recognizing whether or not x is a two-pattern string for a given scope λ. At each step we identify the elements of a 4-tuple $[p, q, i, j]$, if it exists, such that: (1) p and q is a suitable pair of patterns; (2) $|p|, |q| \le \lambda$; (3) for some string y, x is an expansion of y under the morphism σ defined by the 4-tuple. The 4-tuple then also defines an inverse morphism, or **reduction**, $\sigma^{-1} : \; p^i q \to a, p^j q \to b$, such that $\sigma^{-1}(x) = y$.

The recognition algorithm executes simply by performing successive reductions on x, while recording at each step the corresponding 4-tuple (if it exists). If at any step no 4-tuple can be found within the given scope λ, we conclude that x is not a two-pattern string for that scope; while if a sequence of reductions to the letter a can be found, then x is in fact a two-pattern string of scope λ.

In general, for a two-pattern string x there may exist more than one expansion sequence from a. Moreover, there might be a sequence of reductions $\{\delta_1^{-1}, \cdots, \delta_k^{-1}\}$ such that $y = \delta_k^{-1} \cdots \delta_1^{-1}(x)$ is not a two-pattern string. These indicate that the recognition algorithm should try all possible reductions on each level of recursions, blowing up the time complexity of such an algorithm to at least $O(n \log n)$. However, the following two lemmas prove that on each level of recursions we can use the so-called *canonical reduction* and hence assure that the recognition algorithm works in linear time.

DEFINITION 2.1 *A reduction σ^{-1} given by $\sigma = [p, q, i, j]_\lambda$ is called* **λ-canonical**

(1) whenever there exists a suitable pair (p_1, q_1) such that $|p_1| \le \lambda, |q_1| \le \lambda$ and $x = p_1 q_1$, then $|p| \ge |p_1|$ and $x = pq$; otherwise,

(2) whenever σ_1^{-1} given by $[p_1, q_1, i_1, j_1]_\lambda$ is another reduction of x, then either $|p| < |p_1|$, or $|p| = |p_1|$ and $|q| < |q_1|$.

LEMMA 2.1 *If x is a two-pattern string of scope λ and σ^{-1} is the λ-canonical reduction for x, then $\sigma^{-1}(x)$ is again a two-pattern string of scope λ.*

By using induction from Lemma 2.1, we can easily prove that

LEMMA 2.2 *For a two-pattern string x of scope λ, there is a unique expansion sequence $\{\sigma_1, \cdots, \sigma_n\}$ such that each σ_i^{-1} is a λ-canonical reduction.*

It follows that the recognition algorithm need only try the canonical reduction on each level of recursions without any need to try different possible reductions. A straightforward algorithm may then be outlined as shown in Figure 1. RECOGNIZE$_\lambda$ utilizes three algorithms: PRIMITIVE(p) that returns `true` whenever p is primitive (in $O(|p|)$ steps), SUITABLE(p,q) that returns `true` whenever p, q is a suitable pair of patterns (in $O(|p||q|)$ steps), and BASECASE(x) that returns `true` and outputs $(p, q, 1, 0)$ if it finds the largest p such that $x = pq$ where (p, q) is a suitable pair of patterns (in $O(\lambda^4)$ steps using algorithms PRIMITIVE and SUITABLE).

```
boolean RECOGNIZE_λ (x)
 — Deal first with trivial cases
if |x| = 1 then return true
if x = a^|x| or b^|x| then return false
if |x| ≤ 2λ and BASECASE(x) then
  return true
for r ← 1 to min(λ, |x|) do
  p ← x[1..r]   — a candidate for p
  if not PRIMITIVE(p) then
    continue forloop for next r
  else
    compute the maximum k such that
    x[1..kr] = p^k
  for s ← 1 to min(λ, |x| − kr) do
    q ← x[kr+1..kr+s]   — a candidate for q
    if not SUITABLE(p, q) then
      continue forloop for next s
    if for some 0 < i < j, x is formed
    from the concatenations of strings
    p^i q and p^j q then
      output [p, q, i, j]   — which defines σ
      return RECOGNIZE_λ(σ^{-1}(x))
    endif
  endfor
endfor
return false   — as we did not find a suitable
                 pair of patterns
```

Figure 1. Recursive Recognition of a Two-Pattern String

RECOGNIZE$_\lambda$ is a recursive algorithm that reduces the length of x by a factor of at least 2 at each step, and thus over all recursive calls at most $2|x|$ positions need to be scanned. Based on this discussion, and knowing that each call to PRIMITIVE requires at most λ steps, each call to SUITABLE at most λ^2 steps, and each call to BASECASE at most λ^4 steps, we state formally the main result of this section:

THEOREM 2.1 *For every fixed integer $\lambda \geq 1$, the algorithm RECOGNIZE$_\lambda$ determines in $O(2\lambda^4 n)$ time whether or not x is a two-pattern string of scope λ, and, if so, outputs its expansion sequence, also in $O(2\lambda^4 n)$ time.*

3. COMPUTING THE REPETITIONS

In this section, we state two theorems and the main idea of the algorithm $\mathcal{REP}_\lambda$ for computing all the repetitions in a two-pattern string. Due to the limited space, we skip the proofs for the theorems and the details of the algorithm $\mathcal{REP}_\lambda$.

The main idea of the algorithm $\mathcal{REP}_\lambda$ can be described in the following way: given a two-pattern string x of scope λ and its reduction sequence, if y is a reduction of x, then all repetitions in x can be derived through a formula from repetitions in y and other linear configurations in y. The repetitions in y and the linear configurations are themselves derived (in a recursive manner) from repetitions and linear configurations in its reduction. Since the required contribution at each level of reduction (recursion) is linear, the whole algorithm finishes the task in linear time.

The next theorem shows that runs with "small" generators (i.e. length smaller than or equal to a fixed κ, which for $\mathcal{REP}_\lambda$ will in fact be 3λ) can be calculated in time that depends on κ only, and thus we need to concentrate only on calculating runs with "large" generators (i.e. length greater than κ).

THEOREM 3.1 *There is an algorithm that for any $\kappa \geq 1$ and any input binary string x outputs all runs in x whose generator is of size $\leq \kappa$ in $\leq (2^{\kappa+1} - 2)8\kappa^2|x|$ steps.*

Now, we briefly introduce how to compute runs with "large" generators as illustrated in Figure 2. Actually, every run in a two-pattern string x with a generator of size $> 3\lambda$ is an expansion of a run from y, the reduction of x, or is derived from the linear configurations $avbva$, $bvavb$, $bvav$ (if a suffix of y), $bvaavb$, aa, ab, ba or bb in y; further, the linear configurations in y with $|v| > 3\lambda$ can be derived from the linear configurations in its reduction. For instance, every configuration $avbva$ in y with $|v| > 3\lambda$ can be derived from the configurations $aubua$, $buaaub$, aa, ab, ba or bb in its reduction. (Obviously, the configurations aa, ab, ba, bb and the other configurations in y with $|v| \leq 3\lambda$ can be calculated directly from y in linear time.)

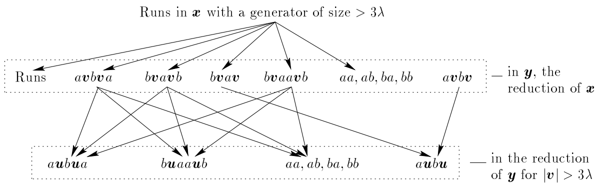

Figure 2. Computation of Runs and Linear Configurations in Two-Pattern Strings

We give an example as follows for illustration. The two-pattern strings of scope 2 generated from the sequence of expansions

$\sigma_1 :\ a \to ab,\ b \to (a)^3 b$
$\quad (p_1 = a, q_1 = b, i = 1, j = 3)$

$\sigma_2 :\ a \to (ab)^4 bb,\ b \to (ab)^5 bb$
$\quad (p_2 = ab, q_2 = bb, i = 4, j = 5)$

$\sigma_3 :\ a \to ab, b \to (a)^2 b$
$\quad (p_3 = a, q_3 = b, i = 1, j = 2)$

are:

$x_1 = \sigma_1(a) :\ \underline{ab}$

$x_2 = \sigma_2(x_1) :\ \underline{ababababbbabababa}babbb$

$x_3 = \sigma_3(x_2) :\ \underline{abaababaababaababaabaabaababaababa}$
$\quad \underline{ababaaba}baababaabaabaab$

To compute the runs in x_3 with a generator of size $> 3\lambda$ (where $\lambda = 2$), we can derive them from the configuration $avbva$ in x_2, which can be calculated directly from x_2 for $|v| \leq 3\lambda$ while for $|v| > 3\lambda$ being derived from the only possible source (configuration) ab (as underlined) in x_1, the reduction of x_2. After obtaining the configuration ab in x_1, we can derive from it the $avbva$ (as underlined where $v = b(ab)^3$) in x_2, from which we can further derive the corresponding run (whose generator as underlined is $(abaab)^4 a$) in x_3, the expansion of x_2.

For the details of how to compute all the runs and the linear configurations involved, the interested readers can refer to Theorem 3 – Theorem 8 in the full version of this paper with all proofs at the URL `http://www.cas.mcmaster.ca/~franek`. Also, the details of the algorithm $\mathcal{REP}_\lambda$ can be found there. The following is a theorem for this algorithm.

THEOREM 3.2 *For any integer $\lambda \geq 1$, there is an integer constant K_λ such that for any two-pattern string x with a scope $\leq \lambda$ and its reduction sequence $\mathcal{S}$, the algorithm $\mathcal{REP}_\lambda$ computes all the repetitions in x in time $\Theta(K_\lambda |x|)$, where the constant K_λ depends only on λ.*

4. CONCLUSIONS AND FUTURE WORK

We have shown how to recognize a two-pattern string with scope λ and how to compute all the repetitions in it in linear time. Actually, when λ is big enough, any string can be regarded as a two-pattern string as long as there exists a partitioning of the string where the left part and the right part form a suitable pair of patterns. To apply the "two-pattern" approach to DNA analysis, we can first group the four letters in a DNA sequence into two, with each group containing two letters; then map the DNA sequence to a binary string. After all the repetitions in the mapped sequence have been computed, we can filter out all the real repetitions according to the original DNA sequence. Our future work is to generalize the results in two-pattern strings to k-pattern strings (where k is a constant ≥ 2). At that time, it will be straightforward to apply "k-pattern" (when $k = 4$) to DNA analysis.

REFERENCES

[1] M. Boshernitzan & A. S. Fraenkel, **A Linear Algorithm for Nonhomogeneous Spectra of Numbers**, *Journal of Algorithms 5* (1984) 187-198.

[2] Maxime Crochemore, **An optimal algorithm for computing the repetitions in a word**, *IPL 12-5* (1981) 244-250.

[3] František Franěk, Ayşe Karaman & W. F. Smyth, **Repetitions in Sturmian strings**, *Theoretical Computer Science 249-2* (2000) 289-303.

[4] C. S. Iliopoulos, Dennis Moore & W. F. Smyth, **A characterization of the squares in a Fibonacci string**, *Theoretical Computer Science 172* (1997) 281-291.