Optimal Computation of all Repetitions in a Weighted String

Carl Barton Solon P. Pissis

Department of Informatics, King's College London, London, UK
{Carl.Barton|Solon.Pissis}@kcl.ac.uk

Abstract

A *repetition* in a string of letters consists of exact concatenations of identical factors of the string. Crochemore's repetitions algorithm, usually also referred to as Crochemore's partitioning algorithm, was introduced in 1981, and was the first optimal $\mathcal{O}(n \log n)$-time algorithm to compute all repetitions in a string of length n. A *weighted string* is a string in which a set of letters may occur at each position with respective probabilities of occurrence. In this article, we present a new variant of Crochemore's partitioning algorithm for weighted strings, which requires optimal time $\mathcal{O}(n \log n)$, thus improving on the best known $\mathcal{O}(n^2)$-time algorithm for computing *all* repetitions in a weighted string of length n.

1 Introduction

A fundamental structural characteristic of a string of letters is its periodicity. Closely related to periodicity is the notion of repetition. Repetitions in strings are highly periodic factors, that is, two or more adjacent identical factors. For instance, string `abab` is a repetition in string `aababba`. In 1981, it was shown by Crochemore that there could be $\mathcal{O}(n \log n)$ repetitions in a string of length n and an $\mathcal{O}(n \log n)$-time, thus optimal, algorithm was presented [1].

Single nucleotide polymorphisms, as well as errors from wet-lab sequencing platforms during the process of DNA sequencing, can occur in some positions of a DNA sequence. In some cases, these errors can be accurately modelled as a *don't care* letter. However, in other cases the errors can be more subtly expressed, and, at each position of the sequence, a probability of occurrence can be assigned to each letter of the nucleotide alphabet; this process gives rise to a *weighted string*.

Recently, the authors of [4] proposed an $\mathcal{O}(n^2)$-time algorithm for computing all repetitions in a weighted string x of length n. The efficiency of the proposed algorithm relies on the assumption of a given constant, the *cumulative weight threshold*, defined as the minimal probability of occurrence of factors in x.

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.

In: Costas S. Iliopoulos, Alessio Langiu (eds.): Proceedings of the 2nd International Conference on Algorithms for Big Data, Palermo, Italy, 7-9 April 2014, published at http://ceur-ws.org/

Our Contribution

We present the first optimal algorithm for computing all repetitions in a weighted string. We improve on the best-known algorithm for computing all repetitions in a weighted string of length n from time $\mathcal{O}(n^2)$ to an optimal $\mathcal{O}(n \log n)$.

2 Preliminaries

An *alphabet* Σ is a finite non-empty set of size σ, whose elements are called *letters*. A *string* on an alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence is called the *empty string*, and is denoted by ε. The *length* of a string x is defined as the length of the sequence associated with the string x, and is denoted by $|x|$. We denote by $x[i]$, for all $0 \leq i < |x|$, the letter at index i of x. Each index i, for all $0 \leq i < |x|$, is a position in x when $x \neq \varepsilon$. It follows that the ith letter of x is the letter at position i in x.

The *concatenation* of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by xy. A string x is a *factor* of a string y if there exist two strings u and v, such that $y = uxv$. Consider the strings x, y, u, and v, such that $y = uxv$. If $u = \varepsilon$, then x is a *prefix* of y. If $v = \varepsilon$, then x is a *suffix* of y. Let x be a non-empty string and y be a string. We say that there exists an *occurrence* of x in y, or, more simply, that x *occurs in* y, when x is a factor of y. Every occurrence of x can be characterised by a position in y. Thus we say that x occurs at the *starting position*.

A weighted string x on an alphabet Σ is a finite sequence of n sets. Every $x[i]$, for all $0 \leq i < n$, is a set of ordered pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having letter s_j at position i. Formally, $x[i] = \{(s_j, \pi_i(s_j)) | s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}$. A letter s_j occurs at position i of a weighted string x if and only if the *occurrence probability* of letter s_j at position i, $\pi_i(s_j)$, is greater than 0. A string u of length m is a factor of a weighted string if and only if it occurs at starting position i with *cumulative occurrence probability* $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. Given a *cumulative weight threshold* $1/z \in (0, 1]$, we say that factor u is *valid*, or equivalently that factor u has a valid occurrence, if it occurs at starting position i and $\prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$.

For every string x and every natural number n, we define the nth power of the string x, denoted by x^n, by $x^0 = \varepsilon$ and $x^k = x^{k-1}x$, for all $1 \leq k \leq n$. A string is said to be *primitive* if it cannot be written as v^e, where $e \geq 2$. A repetition in x is a non-trivial power of a primitive string occurring in x. Formally, a *repetition* u^e, $e \geq 2$, in x is defined as a triple (i, p, e) such that: $u = x[i \,..\, i + p - 1] = x[i + p \,..\, i + 2p - 1] = \ldots = x[i + (e-1)p \,..\, i + ep - 1]$; u^{e+1} does not occur at position i; and u is primitive. A repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i - p]$. The integers p and e are called the *period* and the *exponent* of the repetition, respectively. If $e = 2$ the repetition is called *square*.

A *repetition* $v = u^e$, $e \geq 2$, in a weighted string x is defined as a quadruple (i, p, b, e) such that $u = v[0 \,..\, p - 1] = v[p \,..\, 2p - 1] = \ldots = v[(e-1)p \,..\, ep - 1]$, where v is a factor of length ep of x occurring at position i, and each occurrence of u in v is a valid factor of x; u^{e+1} does not occur at position i; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. A

repetition is maximal if $i - p < 0$ or u^e does not occur at $x[i - p]$. In this article, we are mainly concerned with the following problem.

Problem 2.1 *Given a weighted string x of length n and a cumulative weight threshold $1/z$, find all repetitions in x.*

3 Algorithm

We first perform a colouring stage on x, similar to the one before the construction of the weighted suffix tree [3], which assigns a colour to every position in x according to the following scheme: mark position i *black*, if *none* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$; mark position i *grey*, if *one* of the possible letters at position i has probability of occurrence greater than $1 - 1/z$; mark position i *white*, if *one* of the possible letters at position i has probability of occurrence 1.

Lemma 3.1 ([3]) *A valid factor of x contains at most $\lceil \log z / \log(\frac{z}{z-1}) \rceil$ black positions.*

We then perform a generation stage, as the one performed during the construction of the weighted suffix tree, where a set of factors of x is generated. We refer to this set as *extended factors* (for a definition, see [3]).

Lemma 3.2 ([3]) *A valid factor of x occurs in at least one of its extended factors.*

An *extended repetition* is a repetition occurring in an extended factor of x. A *valid repetition* $v = u^e$, $e \geq 2$, in x is defined as a quadruple (i, p, b, e) such that $u = v[0\,..\,p-1] = v[p\,..\,2p-1] = \ldots = v[(e-1)p\,..\,ep-1]$, where v is a *valid* factor of length ep of x occurring at position i; u^{e+1} is not a valid factor of x; u is primitive; and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$. We define the following subproblem.

Problem 3.3 *Given a weighted string x of length n and a cumulative weight threshold $1/z$, find all valid repetitions in x.*

Lemma 3.4 *Every valid repetition in x occurs in at least one extended factor.*

For each generated extended factor, we run Crochemore's partitioning algorithm for maximal repetitions; the result is all the maximal extended repetitions in x. After computing all the maximal extended repetitions we cannot simply report all of these as valid repetitions. All valid factors must occur in an extended factor but extended factors may contain factors which are not valid. This is a consequence of treating grey positions as white during the generation of extended factors [3]. Since not all maximal extended repetitions are valid repetitions, we must therefore break up these maximal extended repetitions into valid repetitions to solve Problem 3.3.

In order to break up the maximal extended repetitions, we must compute some additional information. To determine how long any valid repetition should be, we must know, for each position i in an extended factor, the length of the longest valid factor starting at position i. The computation is based on the observation that

the longest factor with probability greater than or equal to $1/z$ for the position $i+1$ has length greater than or equal to that of position i. To compute this, we maintain an additional cumulative weight threshold π. We store the computed lengths in an array LF of integers.

We start with the first position in an extended factor and naively compute the longest factor within the threshold by multiplying together the probability of the letters we encounter and storing this in π. If multiplying the probability of some position $j > 0$ causes $\pi < 1/z$ we set $\mathsf{LF}[0] := j - 1$. To proceed, we remove by division the occurrence probability of the first letter from π. If $\pi < 1/z$ then $\mathsf{LF}[1] = j - 1$; otherwise, we continue as before multiplying the probability of $j+1, j+2$, and so on, until the threshold is once again violated. For each extended factor this takes time and space proportional to its length. The sum of lengths of the extended factors is linear in n by the following statement.

Lemma 3.5 ([3]) *The sum of lengths of the extended factors of x is $\mathcal{O}(n)$.*

The next step is to determine the set b for each maximal extended repetition. This can be done in constant time per maximal extended repetition. We compute an array NB of integers of size n, such that for each position i in x, $\mathsf{NB}[i]$ stores the index of the leftmost black position $j > i$; this can be done in linear time in n. For each maximal extended repetition u^e, we check all black positions in the first occurrence of u. There can only be a constant number of black positions in u; finding the black positions using NB takes time proportional to their number. It is now a simple case of recording the position and the letter present in the extended factor; this takes constant time per maximal extended repetition, so time proportional to the number of maximal extended repetitions in total.

Given all the maximal extended repetitions, we can now begin to break them up into valid repetitions. To achieve this, we can check the length of the longest factor starting at position i of the extended factor, and then determine the longest possible repetition starting from i. We can continue checking the maximal extended repetition in this manner reporting the length as we go. Note that in the worst case, for each maximal extended repetition u^e, we may check the starting position of each occurrence of u. As we show later (Lemma 3.7), this can be done efficiently. We now establish the maximal number of extended repetitions in x. Note that the work done by the algorithm so far is no more than the maximal number of extended repetitions.

Lemma 3.6 *There could be $\mathcal{O}(n \log n)$ extended repetitions in x.*

As previously mentioned, whilst breaking some maximal extended repetition u^e into valid repetitions, we may need to check up to e positions. The maximum number of checks required will be the sum of the exponents of all maximal extended repetitions returned by the partitioning. Now we establish the maximal sum of the exponents of maximal extended repetitions in a weighted string.

Lemma 3.7 *The sum of exponents of maximal extended repetitions in x is $\mathcal{O}(n \log n)$.*

Note that an analogous version of Lemma 3.6 holds for valid repetitions.

Theorem 3.8 *Problem 3.3 can be solved in optimal time $\mathcal{O}(n \log n)$.*

At this point, we have solved the subproblem which forms the basis for our solution. Intuitively, the subproblem finds repetitions $v = u^e$, where factor v occurs with probability $\geq 1/z$. The idea behind our solution to Problem 2.1 is based on the observation that a repetition of exponent $e \geq 3$ is composed of overlapping occurrences of smaller repetitions. We intend to compute smaller repetitions and, from this, derive larger ones. Part of the process of computing valid repetitions was to break up maximal extended repetitions below the threshold into smaller valid repetitions. To determine the repetitions specified in Problem 2.1, we reverse this process and *compose* longer repetitions from small valid repetitions.

In order to solve Problem 2.1, we start by solving Problem 3.3 for threshold $k = 1/z^2$. The number of valid repetitions reported for k can be shown to be $\mathcal{O}(n \log n)$ by the same argument as for Lemma 3.6; and the number of black positions in a valid factor is only a constant amount higher than for the original threshold by a similar argument to the proof of Lemma 3.1. We pick $k = 1/z^2$ as we wish to guarantee that we will at least find squares such that each half may have probability greater than or equal to $1/z$. We may also find repetitions with a higher exponent and repetitions which have a probability less than $1/z$, but we will explain how to filter these out using the same techniques as for Problem 3.3.

We alter the solution to Problem 3.3 to simplify the solution to Problem 2.1. Instead of breaking up maximal extended repetitions into valid repetitions, we break them into all their valid overlapping squares. There are no more than $\mathcal{O}(n \log n)$ valid squares by [2]. This can be shown by an almost identical argument as Lemma 3.6. To split maximal extended repetitions into their valid overlapping squares, we process them one by one and create a new square for each overlapping square in the maximal extended repetition. We only need to perform this on maximal extended repetitions of exponent $e \geq 3$, and this will take time proportional to the sum of the exponents which, by Lemma 3.7, is $\mathcal{O}(n \log n)$.

To perform the filtering step, we must check if both halves of the square are above the threshold $1/z$. To check each half, we compute, for each position i in an extended factor, the length of the longest valid factor starting at position i. During the generation of extended factors for the threshold k, we at the same time determine the longest factor with probability greater than or equal to $1/z$ by computing an array LF' which stores the analogous information. Filtering the squares in time proportional to their number can be done by checking that the length stored in the array is greater than or equal to the period of the square.

After the filtering step, we have a set of quadruples (i, p, b, e) representing all primitive squares such that each half of a square has a probability of occurrence at least $1/z$. Now, for every position i in x, we declare an array A_i of linked lists, such that the linked list $\mathsf{A}_i[f_i(j)]$, $f_i : [1, \lfloor n/2 \rfloor] \to [0, \mathcal{O}(\log_\phi n)]$, stores all the squares which occur at position i with period $j \in [1, \lfloor n/2 \rfloor]$. We now wish to establish the size of A_i and the size of the linked lists stored at any $\mathsf{A}_i[f_i(j)]$, but first we state a property of valid factors required to show properties of A_i.

Lemma 3.9 *A valid factor of x is in $\mathcal{O}(1)$ extended factors of x.*

Lemma 3.10 *A_i is of size $\mathcal{O}(\log_\phi n)$, where $\phi = (1+\sqrt{5})/2$, and the size of any linked list $A_i[f_i(j)]$ is $\mathcal{O}(1)$.*

We can now construct the repetitions specified in Problem 2.1. For each position i, we iterate through the linked lists of array A_i. We iterate through each linked list $\mathsf{A}_i[f_i(p)]$, where p is the considered period. We process each square element $(i, p, b, e) \in \mathsf{A}_i[f_i(p)]$ to extend the corresponding square as much as possible, by checking for an occurrence of the square at position $i + p$. For a linear string, it is simple to determine this. For each pair of overlapping squares, the second half of the first square is the first half of the second square; so it suffices to check whether there exists a square at position $i + p$ with the same period.

Example Consider $y =$ `ababab` that contains the following primitive squares: $(0, 2, 2)$, $(1, 2, 2)$, and $(2, 2, 2)$; we wish to find the repetition $(0, 2, 3)$. We start at position 0 of y with $(0, 2, 2)$ and check if there is a square of period 2 starting at position 2. A matching square exists so we extend the repetition and check position 4. There is no square at position 4 so we report the repetition $(0, 2, 3)$.

For weighted strings the approach is very similar, with the addition of a few, constant-time, checks. We must check, for each pair of overlapping squares, if the black positions from the first square match with the black positions from the second square. There is a constant number of black positions so this takes constant time. Each time we find such overlapping squares, we extend our repetition and delete the square at position $i + p$ from the corresponding list. As soon as we find a position where we cannot extend the repetition we stop. We continue doing this until we have found all repetitions.

Each time we iterate through a linked list, a square may be added to the repetition we are extending; this takes constant time per list by Lemma 3.10. After each square is added to the repetition, it is deleted so is not considered again. There are $\mathcal{O}(n \log n)$ squares in the array and from the above description we can see that each square is considered a constant number of times. It is clear that we construct no more repetitions than there are primitive squares, so the number of constructed repetitions is also $\mathcal{O}(n \log n)$. These repetitions will be maximal, and to report repetitions specified in Problem 2.1, we may check the start of each occurrence in the repetition and report them. This takes no more than the sum of exponents which is $\mathcal{O}(n \log n)$. We can now state the main result of this article.

Theorem 3.11 *Problem 2.1 can be solved in optimal time $\mathcal{O}(n \log n)$.*

References

[1] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[2] M. Crochemore, L. Ilie, and W. Rytter. Repetitions in strings: Algorithms and combinatorics. *Theoretical Computer Science*, 410(50):5227 – 5235, 2009. Mathematical Foundations of Computer Science (MFCS 2007).

[3] C. S. Iliopoulos, C. Makris, Y. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. *Fundam. Inf.*, 71(2,3):259–277, Feb. 2006.

[4] H. Zhang, Q. Guo, and C. S. Iliopoulos. Locating tandem repeats in weighted sequences in proteins. *BMC Bioinformatics*, 14(S-8):S2, 2013.