DOI:
10.1007/s00453-016-0266-0

Document Version
Peer reviewed version

Noname manuscript No.
(will be inserted by the editor)

Crochemore's Partitioning on Weighted Strings & Applications

Carl Barton · Solon P. Pissis

the date of receipt and acceptance should be inserted later

Abstract Given a string on alphabet Σ the *partitioning* problem is to compute classes of equivalences on the set of positions of the input string. These classes implicitly memorise identical factors of the string and, hence, their efficient computation is essential for a wide range of string processing applications. We study this problem for a *weighted string*: for every position of the weighted string and every letter of the alphabet a probability of occurrence of this letter at this position is given. Thus a weighted string may represent many different strings, each with probability of occurrence equal to the product of probabilities of its letters at subsequent positions. In this article, we present a non-trivial generalisation of Crochemore's partitioning algorithm (IPL, 1981) that works on weighted strings requiring time $\mathcal{O}(\upsilon n \log \upsilon n)$, where n is the length of the string, $\upsilon = \min\{z^2, zn, \sigma^n\}$, σ is the size of Σ, and $1/z$ is a *cumulative weight threshold*, defined as the minimal probability of occurrence of factors in the string. Our contributions can be summarised as follows: (a) we design the first algorithm to solve the partitioning problem on weighted strings for arbitrary z and σ in time $\mathcal{O}(\upsilon n \log \upsilon n)$ and space $\mathcal{O}(\upsilon n)$ improving the state of the art for $z = \mathcal{O}(1)$; (b) we improve the state of the art for numerous other string processing problems; and (c) we show further combinatorial insight into the relation between weighted and indeterminate strings, that is, sequences of alphabet subsets without associated occurrence probabilities.

Carl Barton
European Bioinformatics Institute (EBI), Wellcome Trust Genome Campus, Hinxton, Cambridge, UK
E-mail: carl@ebi.ac.uk

Solon P. Pissis
Department of Informatics, King's College London, The Strand, London, UK
E-mail: solon.pissis@kcl.ac.uk

1 Introduction

We start by outlining some definitions and notation required to explain our results as well as previous results. An *alphabet* Σ is a finite non-empty set of size σ, whose elements are called *letters*. A *string* on the alphabet Σ is a finite, possibly empty, sequence of elements of Σ. The zero-letter sequence, denoted by ε, is called the *empty string*. The *length* of a string x is defined as the length of the sequence associated with 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 i-th letter of x is the letter at position $i-1$ 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* i in y when $y[i\,..\,i+|x|-1] = x$.

A weighted string x of length n 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* $\Pi_i(u) = \prod_{j=0}^{m-1} \pi_{i+j}(u[j]) > 0$. 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 $\Pi_i(u) = \prod_{j=0}^{m-1} \pi_{i+j}(u[j]) \geq 1/z$. For clarity of presentation, in the rest of this article, a set of ordered pairs in a weighted string is denoted by $\{(s_0, \pi_i(s_0)), \ldots, (s_{\sigma-1}, \pi_i(s_{\sigma-1}))\}$.

Sequences of this type are common in various applications for a number of reasons: (i) data measurements such as imprecise sensor measurements; (ii) flexible modelling of DNA sequences such as DNA binding profiles; (iii) observations are private and thus sequences of observations may have artificial uncertainty introduced deliberately. Consider, for example, the process of DNA sequencing, where single nucleotide polymorphisms (SNPs) can occur. In some cases, these polymorphisms can be accurately modelled as a *don't care* letter. However, sometimes they can be more subtly expressed, and, at each position of the sequence, a probability of occurrence can be assigned to each letter of the alphabet; this process gives rise to a weighted string. For instance, the SNPs present in a population can be incorporated to transform a sequence into a weighted sequence. Consider a IUPAC-encoded [29] DNA sequence, where the ambiguity letter `M` occurs at some position of the sequence, representing

either base A or base C. This gives rise to a weighted DNA sequence, where at the corresponding position of the sequence, we can assign to each of A and C an occurrence probability of 0.5.

The fundamental problem of *pattern matching* on weighted strings has been intensively studied [6–8,23]. A great deal of research has also been conducted for computing various types of *regularities* in a weighted string x of length n (cf. [22,19,12,31,32,5,15]). Most of the algorithms for computing regularities rely on Crochemore's partitioning algorithm [13] adapted to weighted strings; that is, the computation of an equivalence relation on the set of positions of x such that all identical *valid* factors are in the same equivalence class. The best-known algorithms for this computation on weighted strings are the $\mathcal{O}(n^2)$-time algorithms of [22,32] and the $\mathcal{O}(n \log d)$-time algorithm of [12] for computing classes of valid factors of length d. The efficiency of these algorithms relies completely on the assumption of a *constant* cumulative weight threshold and often a fixed-sized alphabet.

It was thus commonly thought that, even under these assumptions, the partitioning scheme could not be efficiently applied to weighted strings [32, 12]. In this article, we show the opposite: it is possible to efficiently compute Crochemore's partitioning under those assumptions; moreover, our approach generalises efficiently for arbitrary z and σ.

Our Contributions. We present the first algorithm for computing equivalence classes of valid factors of a weighted string x of length n for arbitrary z, in time $\mathcal{O}(\upsilon n \log \upsilon n)$, where $\upsilon = \min\{z^2, zn, \sigma^n\}$, improving the best-known algorithm, for $z = \mathcal{O}(1)$, from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$, whilst efficiently generalising for arbitrary σ as well. We then show numerous applications of this technique. Note that the σ^n term only applies in the most unrealistic cases such as when every possible factor of x is valid.

2 The Algorithm

2.1 Properties and Auxiliary Techniques

We start by describing some known properties and techniques used for the processing of weighted strings. In the following discussion, we assume that at each position of the weighted string at least one letter occurs with probability greater than or equal to $1/z$. If this is not the case, the weighted string can be split around this position, and each resulting weighted string can be processed separately. We additionally assume that the string has been filtered so that at each position of the weighted string we keep only those letters with occurrence probability not less than $1/z$; trivially, these are at most $\min\{z, \sigma\}$ per position. This is simply done for convenience as clearly no letter with probability less than $1/z$ is of interest. For clarity of presentation, in the rest of this article, we assume that the weighted string resulting from this filtering step is the input weighted string x of length n. We then apply a simple colouring

scheme on x, similar to the one presented in [19,2], which assigns a colour to every position of x:

- mark position i *black*, if *none* of the possible letters at position i has occurrence probability greater than $1 - 1/z$.
- mark position i *grey*, if *one* of the possible letters at position i has occurrence probability greater than $1 - 1/z$.
- mark position i *white*, if *one* of the possible letters at position i has occurrence probability of 1.

It is straightforward to note that this scheme only applies when $z \geq 2$; for $1 \leq z < 2$ all positions are grey, white or contain no valid letter. Next we define the extended maximal factors of a weighted string.

Definition 1 ([2]) Given a weighted string x and a cumulative weight threshold $1/z \in (0, 1]$, a *maximal factor* of x is a factor f of length h that occurs at position i of x, such that:

- $\Pi_i(f) \geq 1/z$;
- If $i \geq 1$, then $\pi_{i-1}(a) \times \Pi_i(f) < 1/z$, for all $a \in \Sigma$;
- If $i + h < n$, then $\Pi_i(f) \times \pi_{i+h}(a) < 1/z$, for all $a \in \Sigma$.

Informally, a *maximal factor* of a weighted string is a valid factor that cannot be extended to the left nor right and remain valid.

We define the *solid transform* of a weighted string x, denoted by $\mathsf{ST}(x)$, as the weighted string created by: (i) replacing all grey positions by their only valid letter; (ii) setting the occurrence probability of that letter to 1 (all other letters are removed). An *extended maximal factor* of x is then defined as a maximal factor of $\mathsf{ST}(x)$.

Example 1 Let the following string x and $1/z = 0.1$.

$$\mathtt{a}\{(\mathtt{a}, 0.5), (\mathtt{c}, 0.1), (\mathtt{g}, 0.2), (\mathtt{t}, 0.2)\}\mathtt{t}\{(\mathtt{a}, 0.5), (\mathtt{g}, 0.5)\}\mathtt{c}.$$

From position 0, the following extended maximal factors are generated: `aatac`, `aatgc`, `act`, `agtac`, `agtgc`, `attac`, `attgc`.

Lemma 1 ([2]) *Given a weighted string x and a cumulative weight threshold $1/z \in (0, 1]$, any valid factor of x occurs in at least one extended maximal factor.*

Consider some position i of the weighted string x that is a black position and, without loss of generality, assume we start with one factor with probability 1. As the factor is extended and split when extending at black positions, the sum of probabilities of the factors generated from this position is no more than 1 at all times. We show that as more black positions are considered, the sum of probabilities must drop below 1 in a predictable way. First we start by restating an important lemma on extended maximal factors.

Lemma 2 ([2]) *Given a weighted string x and a cumulative weight threshold $1/z \in (0, 1]$, no more than $\lfloor z \rfloor$ extended maximal factors are generated from i, where i is a black position of x.*

We wish to show that a position i of x is contained in $\mathcal{O}(\min\{z^2, zn, \sigma^n\})$ extended maximal factors. We make a distinction between black positions causing at least one extended maximal factor to be split and those which do not; we refer to the former as *branching positions*. By Lemma 2, there are no more than $\lfloor z \rfloor$ of these factors for any black position. We assume that at most z of the black positions after i do not alter the sum of probabilities of the factors generated from i. We now claim that after extending the factors to contain an addition of α black positions, the sum of the probabilities will be no greater than $(1 - \frac{1}{z})^\alpha$.

Lemma 3 *Given a weighted string x, a cumulative weight threshold $1/z \in (0, 1]$, a black position i of x, and the set $\mathcal{M}$ of extended maximal factors generated from i, there are no more than $z(1-1/z)^\alpha$ extended maximal factors containing $z + \alpha$ black positions.*

Proof By Lemma 2, there can be no more than $\lfloor z \rfloor$ extended maximal factors generated from a black position, and therefore there can be no more than $\lfloor z \rfloor$ branching positions. Assume these do not alter the sum of the probabilities of the extended maximal factors. There are now α black positions remaining and by definition no letter at these positions has a probability of occurrence more than $1-1/z$, so every extended maximal factor will have a probability of occurrence reduced by at least this amount. Therefore, the sum of probabilities will be at most $(1 - 1/z)^\alpha$ and this proves the statement. □

Now we are ready to show the following lemma.

Lemma 4 *Given a weighted string x, a cumulative weight threshold $1/z \in (0, 1]$, a black position i, and the set $\mathcal{M}'$ of extended maximal factors containing i, then $|\mathcal{M}'| = \mathcal{O}(\min\{z^2, zn, \sigma^n\})$ and this bound is tight.*

Proof Consider the extended maximal factors containing the k-th black position. By Lemma 3, at most $z(1 - 1/z)^i$ of them start at (i.e., immediately after) the $(k - z - i)$-th black position and at most z of them start at the $(k-i)$-th black position (for $i < z$). So the total number of extended maximal factors containing the k-th black position is

$$\sum_{i=1}^{z-1} z + \sum_{i=0}^{\infty} z(1 - 1/z)^i \leq 2z^2.$$

It was shown in [2] that it is possible to generate a set of extended maximal factors such that a black position is contained in at least z^2 extended maximal factors. The zn comes from Lemma 2 and the observation that no extended maximal factor can be of length greater than n; and σ^n can only occur if every possible factor of x is valid. □

For a factor consisting of white and grey positions, it is clear that the number of times it appears in extended maximal factors is upper bounded by the sum of the number of times the closest black position to the left and right appear

in extended maximal factors. Therefore, by Lemma 4, the sum of lengths of extended maximal factors is $\mathcal{O}(\min\{z^2, zn, \sigma^n\}n)$.

We also compute an array LF for each extended maximal factor by starting at the first position in the extended maximal factor and computing the longest valid factor starting from this position by multiplying together the occurrence probability of the letters we encounter and storing this in π'. If multiplying the probability of some letter at 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 we set $\mathsf{LF}[1] := j - 2$; otherwise, we continue as before multiplying the occurrence probability of letters at positions $j, j+1, j+2$, and so on, until the threshold is once again violated. In general, for an extended maximal factor u and some position i, we set $\mathsf{LF}[i] := \max\{r : \prod_{j=0}^{r-1} \pi_{i+j}(u[j]) \geq 1/z\}$. The LF arrays allow us to efficiently determine the length of the longest valid string beginning at any position of any extended maximal factor. This will allow us to filter out positions from the partitioning scheme when they become invalid. We are now ready to present our algorithm.

2.2 Crochemore's Partitioning for Weighted Strings

In this section, we use the tight bound shown in Lemma 4, along with some painful, however, necessary technical details to achieve the main result of this article. These technical details mainly involve the usage of a series of simple data structures to apply tricks similar to Crochemore's partitioning in the standard setting.

We start by providing a brief description of Crochemore's partitioning algorithm on a regular string y of length n [13]. For a factor w in y, the set of starting positions of all the occurrences of w in y gives us the *start set* of w. We define an equivalence relation $\approx_p$ on the set of positions on y, such that $i \approx_p j$ if and only if $y[i\,..\,i+p-1] = y[j\,..\,j+p-1]$. Therefore, depending on the length of the factor, we get equivalence classes for each length p, for all $1 \leq p \leq n$. Equivalence classes for $p = 1$ are found by going through y once, and keeping the occurrences of each letter in separate sets. For $2 \leq p \leq n$, we consider classes of the previous level, refine them, and compute the classes of level p. On level p, where $2 \leq p \leq n$, we refine a class C with respect to a class DF by splitting C in the classes $\{i \in \mathsf{C} \mid i+1 \in \mathsf{DF}\}$ and $\{i \in \mathsf{C} \mid i+1 \notin \mathsf{DF}\}$. In order to achieve an optimal $\mathcal{O}(n \log n)$ runtime, only classes of the previous level, which were split two levels before, are used for refinement. From those, we can omit the largest siblings of each family, using only the *small* classes for the computation; this is known as the *smaller class trick* [13]. The algorithm terminates when all classes are singletons.

We are now in a position to describe our generalisation for a weighted string x of length n. Let $\mathcal{E}$, $|\mathcal{E}| = s$, represent the multiset of all extended maximal factors of x. We can use the starting position of each factor to distinguish between identical factors and assign a unique label ℓ to each factor from 0 to $s-1$. From this point, when referring to an extended maximal factor it is

referred to as $u_\ell[i\,..\,i+|u_\ell|-1]$ such that u_ℓ occurs at position i in x. We can uniquely identify valid factors of a weighed string by the triple: starting position, length, and the label of the extended maximal factor it is from. An extended maximal factor implicitly represents the black positions contained within the factor and all valid combinations of black positions in the weighted string are represented by extended maximal factors.

Example 2 Consider the weighted string $x = \mathtt{aba}\{(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)\}\mathtt{bab}$ with the cumulative probability threshold $1/z = 0.25$. It generates the following extended maximal factors: `abaabab` and `ababbab` are generated from position 0. We assign them the following labelling: (`abaabab`,0) and (`ababbab`,1).

The approach we take is to generate the initial equivalence classes E_1 from extended maximal factors and then compute an equivalence relation on all extended maximal factors simultaneously. E_1 is generated over all extended maximal factors. Sets of tuples (i, h) are produced, where i is the position in the weighted string and h is the label of the extended maximal factor it was extracted from. The equivalence relation can then be defined as follows:

$$((i,h),(j,k)) \in \mathsf{E}_p \textit{ iff } ((i,h),(j,k)) \in \mathsf{E}_{p-1}$$
$$\text{and } ((i+1,h),(j+1,k)) \in \mathsf{E}_{p-1}.$$

To ensure validity, we further put the following restriction on each equivalence class:

$$\text{for all } ((i,h),(j,k)) \in \mathsf{E}_p,\ \Pi_i(u_h[i\,..\,i+p-1]), \Pi_j(u_k[j\,..\,j+p-1]) \geq 1/z.$$

This means that a pair of tuples $((i, h), (j, k))$ belongs to E_p if and only if the following hold:

- $u_h[i\,..\,i+p-1] = u_k[j\,..\,j+p-1]$;
- $\Pi_i(u_h[i\,..\,i+p-1]) \geq 1/z$;
- $\Pi_j(u_k[j\,..\,j+p-1]) \geq 1/z$.

In the above definitions related to the probabilities of the factors, we now consider the actual probability of occurrence at grey positions again. The main problem considered in this article can then be formally defined as follows.

Problem 1 (Partitioning) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, compute $\mathsf{E}_1, \mathsf{E}_2, \ldots, \mathsf{E}_N$ such that $1 \leq N \leq n$ and $\mathsf{E}_{N-1} \neq \mathsf{E}_N = \mathsf{E}_{N+1}$.

E_1 is computed from the extended maximal factors by creating a tuple (j, k) for every position in every extended maximal factor where j is the index in the weighted string and k is the label of the extended maximal factor it was extracted from. We represent the equivalence relation in two ways: an array EQ; and a linked list EC.

The array EQ gives for each tuple of extended maximal factors x the index of its current equivalence class. EQ is two dimensional, for each extended maximal factor u_ℓ there is an array with size $|u_\ell|$; more formally, for all $0 \leq \ell < s$,

$\mathsf{EQ}[0\,..\,|u_\ell| - 1, \ell]$. Let $\{\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_q\}$ be the equivalence classes of E_p. We define EQ for some tuple (j, k) as follows:

$$\mathsf{EQ}[j, k] = r \text{ } \textit{iff} \text{ } (j, k) \in \mathsf{C}_r.$$

The j indices of this array are offset by the starting position of the extended maximal factor it refers to, to ensure it is consistent with how we reference extended maximal factors.

The equivalence relations are also stored as a doubly-linked list EC. Each equivalence class $\mathsf{EC}[i]$ stores a list of the tuples in the class C_i. We also store a two dimensional array EP (the same size as EQ) to store for each tuple a pointer to the corresponding element in EC. We compute E_1 such that the elements of EC are stored in increasing order of element j of each tuple.

We define a difference function on the tuples that allows for the efficient computation of the nearest *distinct* tuple in the same equivalence class.

$$\mathsf{D}_p(i, h) = \min \begin{cases} \text{the least integer } g \geq 0 & \text{s.t } ((i, h), (i + g, k)) \in \mathsf{E}_p \textit{ iff } h \neq k \\ \text{the least integer } g > 0 & \text{s.t } ((i, h), (i + g, k)) \in \mathsf{E}_p \textit{ iff } h = k \\ \infty & \text{if no such integer exists.} \end{cases}$$

The difference function is represented in a similar way as the equivalence relation: (i) a two dimensional array DF; (ii) a doubly-linked list DC such that $\mathsf{DC}[r]$ gives the list of tuples which satisfy $\mathsf{DF}[i, h] = r$; and (iii) a two dimensional array DP which for each tuple points to its corresponding element in DC.

To refine the equivalence relation efficiently it is important to be able to quickly determine when a position should be excluded because the factor it represents has a probability of occurrence below $1/z$. For each position i in an extended maximal factor, we compute the length of the longest valid factor starting at position i; this is what we compute in the LF array (see Section 2.1). A single LF array is computed for each extended maximal factor and denoted by LF_ℓ for the extended maximal factor with label ℓ.

An additional array of linked lists PR must be computed to efficiently filter out those tuples representing factors that are not valid. Let PR be an array of $n + 1$ linked lists and $\mathsf{PR}[i]$ the list of all pairs (j, k) such that $\mathsf{LF}_k[j] = i$, for $0 \leq k < s$ and $0 \leq i \leq n$. PR specifies the tuples which break the probability threshold at each partitioning step. After computing partitioning step p, $1 \leq p \leq n$, $\mathsf{PR}[p - 1]$ is processed and the tuples which violate the probability threshold are removed. In our algorithm, the partitioning step corresponds to satisfying the first two conditions of the equivalence relation and processing the PR array satisfies the probabilistic condition.

An overview of the proposed method is presented in algorithm WPART below.

```
WPART(weighted string x)
E ← set of extended maximal factors;
Compute array PR on E;
p ← 1;
Define EQ to be E_p on E; define DF to be D_p on E;
Initialise list SMALL to be E_p;
while SMALL ≠ ∅ do
  p ← p + 1;
  {Let ((i,h),(j,k)) ∈ S' iff ((i+1,h),(j+1,k)) exists in a small E_p class}
  EQ ← EQ ∩ S';
  Update DF;
  Remove from EQ and DF tuples stored in PR[p − 1];
  SMALL ← set of indices of small E_p classes;
```

2.3 Correctness and Complexity Analysis

The first steps of the algorithm are to colour x and to generate extended maximal factors. The colouring can be done in time $\mathcal{O}(zn)$ and, by Lemma 4, the generation of extended maximal factors and the size of the initial equivalence classes are in $\mathcal{O}(\min\{z^2, zn, \sigma^n\}n)$ (For a more detailed description see [2]). We know by Lemma 1 that every valid factor occurs in an extended maximal factor, so computing the equivalence relation over extended maximal factors will consider all valid factors. Lemma 4 also tells us that the computation of both the LF_ℓ and PR arrays can be done within the same time complexity $\mathcal{O}(\min\{z^2, zn, \sigma^n\}n)$.

To use the smaller class trick we make a distinction between equivalence classes that are *big* and those that are *small*. Initially, for $p = 1$, all classes are considered *small*, and when a class is refined during the partitioning, all resulting classes are *small* except for the largest which is *big*. The idea behind the efficient computation of the equivalence relation is to only partition with respect to the *small classes*. The indices of the *small* classes will be stored in a linked list SMALL and partitioning is only done with respect to the classes identified here.

The representation of the equivalence relation allows us to move an element from one class to another in constant time. We can directly access any tuple in the EQ array and change the index of its equivalence class in constant time. The EP array can then be used to directly access any element in EC and, as EC is a linked list, move an element in constant time. We now show that partitioning with respect to the *small* classes is correct. We define the following equivalence S_p on tuples:

$$((i,h),(j,k)) \in \mathsf{S}_p \textit{ iff } \begin{cases} ((i,h),(j,k)) \in \mathsf{E}_p \\ \text{or, both } (i,h) \text{ and } (j,k) \text{ are in big } \mathsf{E}_p \text{ classes.} \end{cases}$$

Equivalently:

$$((i,h),(j,k)) \in \mathsf{S}_p \textit{ iff} \text{ for any small } \mathsf{E}_p \text{ class } \mathsf{C}, (i,h) \in \mathsf{C} \textit{ iff } (j,k) \in \mathsf{C}.$$

It now suffices to show the following lemma over tuples of positions.

Lemma 5 *For any step $p \geq 1$ in the partitioning, $((i,h),(j,k)) \in \mathsf{E}_{p+1}$ iff $((i,h),(j,k)) \in \mathsf{E}_p$ and $((i+1,h),(j+1,k)) \in \mathsf{S}_p$.*

Proof ($\Rightarrow$) E_p is a refinement of S_p therefore $\mathsf{E}_{p+1} \subset \mathsf{E}_p \cap \mathsf{S}_p$.
($\Leftarrow$) Let $((i,h),(j,k))$ be two positions such that

$$((i,h),(j,k)) \in \mathsf{E}_p \text{ and } ((i+1,h),(j+1,k)) \in \mathsf{S}_p.$$

- If $(i+1,h)$ is in a small E_p class then $(j+1,k)$ is in the same E_p class; so $((i,h),(j,k)) \in \mathsf{E}_{p+1}$.
- If $(i+1,h)$ is in a big E_p class then $(j+1,k)$ is also in a big E_p class. From $((i,h),(j,k)) \in \mathsf{E}_p$ we can deduce that $((i+1,h),(j+1,k)) \in \mathsf{E}_{p-1}$ so they are in the same big E_p class. So again we have that $((i,h),(j,k)) \in \mathsf{E}_{p+1}$.

□

Lemma 5 establishes that the smaller class trick can be used for partitioning. It also shows that partitioning may be performed by computing the following:

$$((i,h),(j,k)) \in \mathsf{S}'_p \text{ iff } ((i+1,h),(j+1,k)) \in \mathsf{S}_p$$

$$\mathsf{E}_{p+1} = \mathsf{E}_p \cap \mathsf{S}'_p.$$

When a class is split, we keep track of how many elements are in each new equivalence class. When the partitioning step has been completed the class with the maximum number of elements is labelled as *big*, the rest are labelled *small*, and the indices of *small* classes are stored in the linked list SMALL. Partitioning therefore takes time proportional to the sum of the sizes of all *small* classes. In the following lemma, we establish the sum of sizes of all *small* classes and the time required to partition them.

Lemma 6 *Let R be the sum of the sizes of all* small *classes. It holds that $R = \mathcal{O}(vn \log vn)$, where $v = \min\{z^2, zn, \sigma^n\}$.*

Proof Consider a tuple (i,h), for a partitioning step p, in a *small* E_p class F and let F' be its E_{p-1} class. By definition of the *small* classes:

$$|\mathsf{F}| \leq |\mathsf{F}'|/2.$$

Thus no position can be in more than $\mathcal{O}(\log vn)$ *small* classes since by Lemma 4 the E_1 class of i has cardinality less than $\mathcal{O}(vn)$; by Lemma 4 there are $\mathcal{O}(v)$ tuples of the form (i,h) and this proves the claim. □

There are a few final technical considerations:

- Efficiently perform the pruning step so each E_p class only contains factors which have a probability of occurrence no less than $1/z$;
- Ensure the pruning step does not cause problems for partitioning;
- Ensure that the difference function DF_p is correctly updated.

To realise the pruning step we make use of arrays PR and EP. After partitioning step p, we process the list $\mathsf{PR}[p-1]$ which gives the elements which require pruning. An element contained in the PR array can be removed from its E_p class in constant time by setting its class to -1 in the EQ array and removing it from the list EC using the EP array to access the element in constant time. The initial E_1 classes only have $\mathcal{O}(\upsilon n)$ elements so the pruning step cannot do any more work than this.

Consider that some tuple (i,h) has been removed, we must be able to guarantee that partitioning with respect to tuple $(i-1,h)$ is still computed correctly. If (i,h) was removed at step p then it must be the case that either $(i-1,h)$ has already been removed or that $(i-1,h)$ will become invalid when considering step $p+1$. The only reason $(i-1,h)$ must be included in the partitioning at step $p+1$ is to guarantee the correct partitioning of the tuple $(i-2,h)$ which may be valid at step $p+1$ but will not be at $p+2$. The E_{p+1} class that $(i-1,h)$ ends up in after partitioning is therefore irrelevant as it will be removed immediately. By the correctness of the pruning stage all empty classes can simply be ignored.

It remains to show that DF and DC are correctly computed in the same time complexity as partitioning. We compute DF_1 after the computation of E_1; as E_1 is stored in ascending order of the first element of the tuple, this requires constant time per tuple. Consider that during a partitioning step, each tuple that requires moving from some E_p class to another, is performed one at a time. Let (i,h) and (j,k) be two tuples such that (j,k) is a tuple that has been moved and (i,h) is the tuple that preceded (j,k) before it was moved; then the following must be true where DF' denotes the updated DF:

$$\mathsf{DF}'[i,h] = \mathsf{DF}[i,h] + \mathsf{DF}[j,k].$$

Furthermore, let (ℓ,m) be the tuple preceding (j,k) in its new equivalence class, then the following holds:

$$\mathsf{DF}'[\ell,m] = j - \ell.$$

Updating DC can be done by transferring a tuple to a new DC at the same time it is moved from an EC. We update the DF array entries of the affected tuples, as specified above, and update DC by moving a tuple (i,h) that has been updated into $\mathsf{DC}[\mathsf{DF}[i,h]]$. This can be achieved in constant time by using the DP array. Note that DF_p, for a partitioning step p, can be correctly updated during the pruning step in the same way as specified above.

With everything described above, we have presented a new $\mathcal{O}(\upsilon n \log \upsilon n)$-time algorithm, where $\upsilon = \min\{z^2, zn, \sigma^n\}$, for the computation of equivalence classes in weighted strings and get the following result.

Theorem 1 *The* Partitioning *problem can be solved in time* $\mathcal{O}(\upsilon n \log \upsilon n)$ *and space* $\mathcal{O}(\upsilon n)$*, where* $\upsilon = \min\{z^2, zn, \sigma^n\}$.

An example of the proposed method is shown in Figure 1.

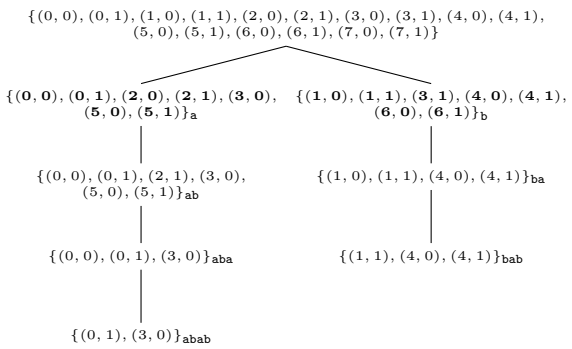

Fig. 1 Classes of equivalence with their refinements for weighted string $x = \mathtt{aba}\{(\mathtt{a}, 0.5), (\mathtt{b}, 0.5)\}\mathtt{bab}$ and $1/z = 1/4$. The small classes of the partitioning are shown in bold. To improve readability all singletons have been omitted, and we assume a unique terminating letter for each extended maximal factor.

3 Applications to Weighted and Indeterminate Strings

An *indeterminate string* w of length n on an alphabet Σ is a finite sequence of n sets, such that $w[i] \subseteq \Sigma$, $w[i] \neq \emptyset$, for all $0 \leq i < n$. If $|w[i]| = 1$, that is, $w[i]$ represents a single letter of Σ, we say that $w[i]$ is a *solid* position. Any indeterminate string w of length n can be represented by a weighted string x of length n such that $(a, \frac{1}{|w[i]|}) \in x[i]$ *iff* $a \in w[i]$ and setting z to the probability of the lowest probability factor of x.

In this section, we outline a number of applications of our techniques to solving various problems in weighted and indeterminate strings. In many cases, we provide the first algorithms for these problems on weighted and indeterminate strings and for others we improve on the state of the art. For some of the problems considered below, it is possible to check if the string is indeterminate, and if so we can optimise the computation of extended maximal factors by only computing extended maximal factors of length $\lceil \frac{n}{2} \rceil$ giving us $\upsilon = \min\{z^2, zn, n\sigma^{\frac{n}{2}}\}$.

3.1 Computing Covers and Seeds

A string u is a *cover* of a string y if every position of y lies within some occurrence of u in y and $u \neq y$. A string u is a cover of a weighted string x if every position in x is lies within a *valid* occurrence of u and $|u| \leq |x|$. A cover u of x, occurring at position 0 of x, can be represented as a pair (p, b), where p is the length of the cover and b is a set of ordered pairs (j, a), where $0 \leq j < p$ and $a \in \Sigma$, denoting $u[j] = a$; b is used to uniquely define a cover as there may be ambiguous positions in x. The AllCovers problem can be therefore defined as follows.

Problem 2 (AllCovers) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all covers of x.

Covers were first introduced by Apostolico *et al.* in [3] for regular strings, where the authors presented a linear-time algorithm to test the superprimitivity of a string and to give its shortest cover. Since the introduction of covers, their efficient computation and combinatorial properties have been extensively studied for regular strings. Breslauer [10] presented an online linear-time algorithm for the computation of not just the shortest cover of some string but also the shortest cover of all its prefixes. Moore and Smyth [28] presented a linear-time algorithm for computing *all* the covers of a string; this was later [26] improved to an online linear-time algorithm for the computation of all covers of every prefix of a string. Flouri *et al.* presented a linear-time algorithm to compute all enhanced covers, a generalisation of the notion of cover [16].

For the case of weighted strings, there exists an $\mathcal{O}(n)$-time algorithm for computing all covers [19]. This result only holds for $z = \mathcal{O}(1)$ and $\sigma = \mathcal{O}(1)$; for other values of z and σ little is known. For indeterminate strings a simple $\mathcal{O}(n\sigma^{k/2}k)$-time algorithm and a fixed parameter tractable $2^{\mathcal{O}(k \log k)} + nk^{\mathcal{O}(1)}$-time algorithm was presented in [14], where k is the number of non-solid positions and $1 < \sigma \leq n$. In [1], an $\mathcal{O}(n^2)$-time algorithm was presented for some, but not all, covers of an indeterminate string.

With the computation of EQ, computing all covers in a weighted string is fairly straightforward. For an equivalence class C_r, we maintain a variable MG_r storing the maximum gap between any two consecutive tuples in C_r, where tuples are ordered by their first element. More formally, let $\mathsf{C}_r = \{(x_1, e_1), (x_2, e_2), \ldots, (x_a, e_a)\}$ be a set of tuples sorted by their first element; MG_r is therefore:

$$\mathsf{MG}_r = \max\{x_{i+1} - x_i\}, \text{ for } 0 < i < a.$$

By definition of the equivalence classes, MG_r remains the same or increases. Let $\{\mathsf{C}_1, \mathsf{C}_2, \ldots, \mathsf{C}_q\}$ be the equivalence classes of E_p. In order to compute all covers, we maintain a variable MG_r for each equivalence class C_r which contains a tuple of the form $(0, k)$, for any r and k, as any cover must be a prefix of x. The initial values of MG_r for the equivalence classes of E_1 are computed by brute force when the initial equivalence relation is computed. We only update the value of a MG_r variable if C_r is split and $(0, k) \in \mathsf{C}_r$ for some k. When splitting a class, the appropriate MG_r variable can be updated for both classes in time proportional to the size of the smallest class. At each step p in the partitioning, a class, such that $(0, k) \in \mathsf{C}_r$, is a cover if and only if the following conditions hold:

- $\mathsf{MG}_r \leq p$;
- $(n - p + 1, h) \in \mathsf{C}_r$, for any h.

Each cover is represented as a pair: length and index of an extended maximal factor, thereby implicitly specifying the set b and allowing reporting in constant time. The checks above are only applied when a class is split. Therefore we obtain the following result.

Theorem 2 *The* AllCovers *problem can be solved in time* $\mathcal{O}(\upsilon n \log \upsilon n)$ *and space* $\mathcal{O}(\upsilon n)$*, where* $\upsilon = \min\{z^2, zn, \sigma^n\}$.

A related notion to covers is that of *seeds*. Intuitively, seeds are similar to covers, but a seed may additionally be a cover of a superstring of y, rather than a cover of y itself. Seeds in weighted strings can then be defined as follows. A string u is a seed of a weighted string x if there exists a weighted superstring of x for which u is a cover, every occurrence of u is valid, and $|u| < n$. A seed u can be represented, similar to cover, as a tuple (i, p, b), where the additional element i is the starting position of a valid occurrence of u in x. The AllSeeds problem can be therefore defined as follows.

Problem 3 (AllSeeds) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all seeds of x.

Computing all seeds in weighted strings was considered in [31], where an $\mathcal{O}(n^2)$-time algorithm was presented for $z = \mathcal{O}(1)$. To the best of our knowledge, no other algorithm exists for computing *all* seeds in a weighted string. To quickly compute seeds, we must be able to efficiently determine if a string u, represented by an equivalence class of E_r, is a seed; this is true if and only if the following conditions hold:

1. $\mathsf{MG}_r \leq |u|$;
2. There exists a suffix s of u that is a valid prefix of x and $|s| \geq \mathsf{first}(u)$;
3. There exists a prefix p of u that is a valid suffix of x and $|p| \geq n - \mathsf{last}(u) - 1$;

where $\mathsf{first}(u)$ denotes the *starting* position of the first valid occurrence of u in x, and $\mathsf{last}(u)$ denotes the *ending* position of the last valid occurrence of u in x. It is known that Conditions 1-3 need only be checked as classes are split at each level. The first condition is checked trivially, but Conditions 2 & 3 are more complex.

Let i be a position in an extended maximal factor t and assume we have computed the ending position j of the longest valid match between a prefix of x and the i-th suffix of t, along with the starting position k of the longest valid match between a suffix of x and a factor ending at i. For each extended maximal factor we compute this information for all i representing it as intervals $[i, j]$ and $[k, i]$ respectively. We compute this information for all extended maximal factors and for each extended maximal factor store the prefix information and suffix information in two separate arrays. These arrays are sorted by the first element of each entry; we then additionally construct a Range Maximum Query (RMQ) data structure on the second element of each entry.

This structure allows us to answer in constant time [9] queries such as: Does string u occurring at position i in extended maximal factor t have a suffix (prefix) of length at least m that occurs as a prefix (suffix) of the weighted string x? Clearly checking Conditions 2 & 3 can be cast as one of these queries.

For clarity we will only discuss how to answer if u has a suffix of length at least m that is a prefix as the other case is symmetric. For u to have a suffix of length at least m that is a prefix of x there must exist an interval $[z_1, z_2]$ in our prefix information that satisfies:

1. $i \leq z_1 \leq i + |u| - m$; and
2. $z_2 \geq i + |u| - 1$.

These two conditions ensure that the longest valid prefix starting at z_1 is at least of length m. To check such an interval exists we perform two binary searches for those intervals with a first element satisfying the first bullet point, and then perform an RMQ for this range. If the result has a second element at least $i + |u| - 1$ then an appropriate interval exists and the query is answered. The prefix/suffix information can be computed through the partitioning algorithm taking time $\mathcal{O}(\upsilon n \log \upsilon n)$. When splitting an EQ class, we first check if the class contains a prefix; the class is then split as normal; we then find if any class no longer contains a prefix and record this information. Clearly the suffix condition can be updated as each suffix tuple becomes a singleton. Construction of all the relevant data structures takes time $\mathcal{O}(\upsilon n \log \upsilon n)$ as sorting the intervals dominates the time complexity. We can run the entire partitioning algorithm once to compute all the interval information and then a second time to detect seeds. Therefore we obtain the following result.

Theorem 3 *The* AllSeeds *problem can be solved in time* $\mathcal{O}(\upsilon n \log \upsilon n + \alpha)$, *where* $\upsilon = \min\{z^2, zn, \sigma^n\}$ *and* α *is the size of the output.*

3.2 Maximal Quasiperiodicities

Maximal quasiperiodicities are a further notion of periodicity that has been studied in strings and are strongly related to covers. Informally, maximal quasiperiodicities are to covers what repetitions are to periods: they are a *local* version of covers. Informally, a string is called *quasiperiodic* if it has a cover. A factor u of a string y is a *maximal quasiperiodicity* if it has a cover v and no extension of u can be covered by v or va, where a is the letter following u in x. Similarly, we can define maximal quasiperiodicities in weighted strings as follows. A factor u of a weighted string x is a maximal quasiperiodicity if it has a cover v and no extension of u can be covered by v or va, where a is the letter following u in x. The MaximalQuasiperiodicities problem can be therefore defined as follows.

Problem 4 (MaximalQuasiperiodicities) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all maximal quasiperiodicities of x.

For regular strings, the fastest algorithms are the $\mathcal{O}(n \log n)$-time algorithms presented in [21,11,20]. These algorithms were later shown to be optimal [18]. The algorithm of [21,20] is based on Crochemore's partitioning for regular strings and a left/right gap array defined as follows. Let $\mathsf{C}_r = \{(x_1, e_1), (x_2, e_2), \ldots, (x_a, e_a)\}$ be an equivalence class then:

$$\mathsf{LG}[i-1] = x_i - x_{i-1} \text{ for all } i \in \{2, \ldots, a\}$$

$$\mathsf{RG}[i-1] = x_{i+1} - x_i \text{ for all } i \in \{1, \ldots, a-1\},$$

with $\mathsf{LG}[0] = 0$ and $\mathsf{RG}[a-1] = \infty$. Clearly these arrays can be easily maintained during the partitioning. We can then apply the algorithm of [21,20] to obtain the following result.

Theorem 4 *Let $\upsilon = \min\{z^2, zn, \sigma^n\}$, the* MaximalQuasiperiodicities *problem can be solved in time $\mathcal{O}(\upsilon n \log \upsilon n + \alpha)$, where α is the size of the output.*

3.3 Computing Squares and Runs

Here we define a *square* as any non-empty string of the form yy. A *square* in a weighted string x is defined as a factor of the form yy such that both occurrences of y are valid. The AllSquares problem can be therefore defined as follows.

Problem 5 (AllSquares) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all squares of x.

There exists an optimal $\mathcal{O}(n \log n)$-time algorithm for computing all squares of x for $z = \mathcal{O}(1)$ [5], matching the time complexity of the algorithm for squares in the standard setting [27]. To the best of our knowledge, there is no known algorithm for computing squares in indeterminate strings or weighted strings for arbitrary z.

By Lemma 4, there may be many tuples with the same first element occuring consecutively in the difference function, this makes retrieving the squares more complicated than the non-weighted case. We solve this by storing a refined version of the difference function that we call the *compressed* difference function, denoted by CDF. CDF is the same as DF but where there exists consective tuples with the same first element, only the first is stored.

For each element of DF we store a pointer between its element in CDF and vice versa. The computation of this additional array is initially done by brute force for E_1 and can be easily updated as DF is updated. From the compressed difference function we compute an additional pointer for each tuple that will allow us to easily report squares. These pointers are in some sense a special *skip list* on CDF, allowing us to skip to the entry we are interested in. We store it by augmenting CDF so that each element also has an additional *skip* pointer. For some element (i, h) in CDF we define the target of the skip pointer as (j, k) such that it is the least such j satisfying the invariant $j - i \geq p$ such that $(i, h), (j, k) \in \mathsf{E}_p$ and (j, k) exists in CDF. If no such tuple exists then the skip pointer is undefined. Now we state an important fact about skip pointers.

Fact 1 *Let (i, h) and (j, k) be elements in CDF and $(i, h), (j, k) \in \mathsf{E}_p$. If $j > i$, the skip pointer of (i, h) points to (ℓ, m), and the skip pointer of (j, k) points to (o, q), then either $(\ell, m) = (o, q)$ or $\ell \leq o$.*

The skip pointers can be trivially computed for E_1. Now we must show that the skip pointers can be easily updated at each stage and that we can correctly report squares. Assuming at some stage p that the skip pointers are correctly computed then at stage $p+1$ there are two cases where a skip pointer may need to be updated: 1) if the equivalence class is split at stage $p+1$; or 2) if the invariant is violated. In Case 1, we split CDF as normal and for the new class we scan through CDF to find the correct skip pointer for the first element; from here each subsequent element can be updated easily by Fact 1. For the undefined skip pointers in the class that was split, their correct target is the closest tuple with a larger first element with respect to the previous target. To make this easy to update we perform the partitioning in decreasing order of position. In Case 2, we can update the skip pointer to the tuple that immediately follows its current target in CDF.

It remains to show that we can detect when the invariant is violated. This is achieved by augmenting CDF with an array CDP such that $\mathsf{CDP}[p]$ gives the tuples in CDP such that (i, h) and (j, k) are in CDF and $j - i = p$. Clearly these are only updated at the same time a skip pointer is updated. Finally, it should be clear to see that, by the definition of the skip pointers, $j - i = p$ is a condition that is satisfied only when two identical factors are next to each other; equivalently, when they form a square. Given this information at each stage p, we read $\mathsf{CDP}[p]$ to report squares and update CDF. Therefore we obtain the following result.

Theorem 5 *The* AllSquares *problem can be solved in time* $\mathcal{O}(\upsilon n \log \upsilon n + \alpha)$ *and space* $\mathcal{O}(\upsilon n)$*, where* $\upsilon = \min\{z^2, zn, \sigma^n\}$ *and* α *is the output size.*

The various (equivalent) definitions of *runs* in regular strings are not readily generalisable to weighted strings due to notion of validity. Rytter [30] defines a run in a string w as an interval $\beta = [i \,..\, j]$ such that $w[i \,..\, j]$ is a periodic word with the period p and this period is not expendable to the left or right. An alternative definition is the following: a factor $u^r t = x[i \,..\, i + rp + |t| - 1]$ is a *run* in x if and only if u is primitive, t is a proper prefix of u and no w^y exists at position $i - 1$ or ends at position $i + rp + |t|$ where $|w| = p$. A direct extension of the definition of Rytter may lead us to require that the entire run in a weighted string is valid. Such a definition would seem to run counter to a natural definition of a square, where each occurrence is required to be valid. Extending the second definition may seem more natural, and allows us to define runs in the following way. A factor $v = u^r t$ is a run in a weighted string x if and only if u is primitive and u^r occurs at position i of x with period p and there neither exists a factor $va = w^r s$ occurring at i nor a factor $av = w^r t'$ occurring at $i - 1$ with period p, for some $a \in \Sigma$, such that each w and t' are valid and t' is a proper prefix of w. However, this also seems insufficient as the notion of extending a run by a single letter does not seem to capture runs with the concept of validity. Specifically, extending to the left of a run changes the root factor u, which could destroy the validity of the root entirely, even though such an extension exists. Here we consider runs as being defined as groups of coalescing primitively rooted squares, a property implicit in all

definitions of squares in regular strings. A primitively rooted square is a factor yy such that y is not the power of any other string. As such we define a *run* in a weighted string x as a factor ww' such that there is a primitively rooted square of period p at each position in w, each primitively rooted square is valid, w' is a proper valid prefix of w, $|w'| < p$, and ww' cannot be extended. This definition captures the original definition of runs and generalises it to weighted strings. The AllRuns problem can be therefore defined as follows.

Problem 6 (AllRuns) Given a weighted string x of length n and a cumulative weight threshold $1/z \in (0, 1]$, find all runs of x.

Runs computation is well-studied in the standard setting with linear-time algorithms known, but there is no known algorithm for weighted or indeterminate strings. Kolpakov and Kucherov, in [25], showed that the number of runs in a string is linear in its length; they additionally provided the first linear-time algorithm for computing runs in a regular string. After many years of work, it was recently shown by Bannai *et al.* [4] that the number of runs in a string is strictly less than its length and that their characterisation provides a far simpler algorithm for runs computation than what was previously known. Prior to the development of the algorithm by Bannai *et al.*, a number of super-linear-time algorithms were developed, including the $\mathcal{O}(n \log n)$-time algorithm by Franek *et al.* [17], which is based primarily on Crochemore's partitioning. The main idea of this algorithm is computing all primitively rooted squares and then post processing these to combine them into runs. Clearly our squares algorithm computes at least this information. By filtering non-primitive squares, using the data structure of [24], the algorithm of Franek *et al.* can be applied. By combining this algorithm with our partitioning algorithm, we give the first algorithm to compute runs in weighted strings and obtain the following result.

Theorem 6 *The* AllRuns *problem can be solved in time* $\mathcal{O}(\upsilon n \log \upsilon n + \alpha)$ *and space* $\mathcal{O}(\upsilon n)$*, where* $\upsilon = \min\{z^2, zn, \sigma^n\}$ *and* α *is the output size of solving the* AllSquares *problem on* x*.*

4 Final Remarks

In this article, we presented a non-trivial generalisation of Crochemore's partitioning algorithm for weighted strings that requires time $\mathcal{O}(\upsilon n \log \upsilon n)$ and space $\mathcal{O}(\upsilon n)$, where $\upsilon = \min\{z^2, zn, \sigma^n\}$. Our algorithm improves on the best-known algorithm, for $z = \mathcal{O}(1)$, from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.

As a direct result of this generalisation, we provided more efficient algorithms for numerous other string processing problems. Due to the limited results on the combinatorics of weighted strings, most of the additional problems we have presented have no known bounds on the size of the output. As an immediate target, we plan to investigate the output size of the considered problems and the combinatorics associated with this.

Acknowledgements

We warmly thank Panagiotis Charalampopoulos (King's College London) for making many helpful comments on this manuscript.

References

1. Alatabbi, A., Rahman, M.S., Smyth, W.F.: Computing covers using prefix tables. Discrete Applied Mathematics **212**, 2–9 (2016). DOI 10.1016/j.dam.2015.05.019
2. Amir, A., Chencinski, E., Iliopoulos, C.S., Kopelowitz, T., Zhang, H.: Property matching and weighted matching. Theor. Comput. Sci. **395**(2-3), 298–310. (2008). DOI 10.1016/j.tcs.2008.01.006
3. Apostolico, A., Farach, M., Iliopoulos, C.S.: Optimal superprimitivity testing for strings. Inf. Process. Lett **39**(1), 17–20 (1991)
4. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. arXiv:1406.0263v7 (2014). URL `http://arxiv.org/abs/1406.0263`
5. Barton, C., Iliopoulos, C., Pissis, S.: Optimal computation of all tandem repeats in a weighted sequence. Algorithms For Molecular Biology **9** (2014). DOI 10.1186/s13015-014-0021-5
6. Barton, C., Kociumaka, T., Pissis, S.P., Radoszewski, J.: Efficient Index for Weighted Sequences. In: R. Grossi, M. Lewenstein (eds.) 27th Annual Symposium on Combinatorial Pattern Matching (CPM 2016), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 54, pp. 4:1–4:13. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). DOI 10.4230/LIPIcs.CPM.2016.4
7. Barton, C., Liu, C., Pissis, S.P.: Linear-time computation of prefix table for weighted strings & applications. Theor. Comput. Sci. **656**(Part B), 160–172 (2016). DOI http://dx.doi.org/10.1016/j.tcs.2016.04.029. Stringology: In Celebration of Bill Smyth's 80th Birthday
8. Barton, C., Liu, C., Pissis, S.P.: On-line pattern matching on uncertain sequences and applications. In: T.H. Chan, M. Li, L. Wang (eds.) Combinatorial Optimization and Applications - 10th International Conference, COCOA 2016, Hong Kong, China, December 16-18, 2016, Proceedings, *Lecture Notes in Computer Science*, vol. 10043, pp. 547–562. Springer (2016). DOI 10.1007/978-3-319-48749-6_40
9. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: G.H. Gonnet, D. Panario, A. Viola (eds.) LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings, *Lecture Notes in Computer Science*, vol. 1776, pp. 88–94. Springer (2000). DOI 10.1007/10719839_9
10. Breslauer, D.: An on-line string superprimitivity test. Inf. Process. Lett **44**(6), 345–347 (1992)
11. Brodal, G.S., Pedersen, C.N.S.: Finding maximal quasiperiodicities in strings. In: R. Giancarlo, D. Sankoff (eds.) Combinatorial Pattern Matching, 11th Annual Symposium, CPM 2000, Montreal, Canada, June 21-23, 2000, Proceedings, *Lecture Notes in Computer Science*, vol. 1848, pp. 397–411. Springer (2000). DOI 10.1007/3-540-45123-4_33
12. Christodoulakis, M., Iliopoulos, C.S., Mouchard, L., Perdikuri, K., Tsakalidis, A.K., Tsichlas, K.: Computation of repetitions and regularities of biologically weighted sequences. Journal of Computational Biology **13**(6), 1214–1231. (2006)
13. Crochemore, M.: An optimal algorithm for computing the repetitions in a word. Inf. Process. Lett. **12**(5), 244–250 (1981). DOI 10.1016/0020-0190(81)90024-7
14. Crochemore, M., Iliopoulos, C.S., Kociumaka, T., Radoszewski, J., Rytter, W., Walen, T.: Covering problems for partial words and for indeterminate strings. In: H. Ahn, C. Shin (eds.) Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8889, pp. 220–232. Springer (2014). DOI 10.1007/978-3-319-13075-0_18
15. Cygan, M., Kubica, M., Radoszewski, J., Rytter, W., Walen, T.: Polynomial-time approximation algorithms for weighted LCS problem. Discrete Applied Mathematics **204**, 38–48 (2016). DOI 10.1016/j.dam.2015.11.011

16. Flouri, T., Iliopoulos, C.S., Kociumaka, T., Pissis, S.P., Puglisi, S.J., Smyth, W.F., Tyczynski, W.: Enhanced string covering. Theor. Comput. Sci. **506**, 102–114 (2013). DOI 10.1016/j.tcs.2013.08.013
17. Franek, F., Jiang, M., Weng, C.: An improved version of the runs algorithm based on crochemore's partitioning algorithm. In: J. Holub, J. Zdárek (eds.) Proceedings of the Prague Stringology Conference 2011, Prague, Czech Republic, August 29-31, 2011, pp. 98–105. Prague Stringology Club, Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague (2011)
18. Groult, R., Richomme, G.: Optimality of some algorithms to detect quasiperiodicities. Theor. Comput. Sci. **411**(34–36), 3110–3122 (2010). DOI 10.1016/j.tcs.2010.04.039
19. Iliopoulos, C.S., Makris, C., Panagis, Y., Perdikuri, K., Theodoridis, E., Tsakalidis, A.: The weighted suffix tree: An efficient data structure for handling molecular weighted sequences and its applications. Fundam. Inf. **71**(2,3), 259–277 (2006)
20. Iliopoulos, C.S., Mouchard, L.: Fast local covers. Tech. Rep. TR98-03 (1998)
21. Iliopoulos, C.S., Mouchard, L.: Quasiperiodicity: From detection to normal forms. Journal of Automata, Languages and Combinatorics **4**(3), 213–228. (1999)
22. Iliopoulos, C.S., Mouchard, L., Perdikuri, K., Tsakalidis, A.K.: Computing the repetitions in a biological weighted sequence. Journal of Automata, Languages and Combinatorics **10**(5/6), 687–696. (2005)
23. Kociumaka, T., Pissis, S.P., Radoszewski, J.: Pattern Matching and Consensus Problems on Weighted Sequences and Profiles. In: S.H. Hong (ed.) 27th International Symposium on Algorithms and Computation (ISAAC 2016), *Leibniz International Proceedings in Informatics (LIPIcs)*, vol. 64, pp. 46:1–46:12. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2016). DOI http://dx.doi.org/10.4230/LIPIcs.ISAAC.2016.46
24. Kociumaka, T., Radoszewski, J., Rytter, W., Walen, T.: Efficient data structures for the factor periodicity problem. In: L. Calderón-Benavides, C.N. González-Caro, E. Chávez, N. Ziviani (eds.) String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings, *Lecture Notes in Computer Science*, vol. 7608, pp. 284–294. Springer (2012). DOI 10.1007/978-3-642-34109-0_30
25. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA, pp. 596–604. IEEE Computer Society (1999). DOI 10.1109/SFFCS.1999.814634
26. Li, Y., Smyth, W.F.: Computing the cover array in linear time. Algorithmica **32**(1), 95–106 (2002). DOI 10.1007/s00453-001-0062-2
27. Main, M.G., Lorentz, R.J.: An $O(n \log n)$ algorithm for finding all repetitions in a string. J. Algorithms **5**(3), 422–432 (1984). DOI 10.1016/0196-6774(84)90021-X
28. Moore, D., Smyth, W.F.: An optimal algorithm to compute all the covers of a string. Inf. Process. Lett. **50**(5), 239–246 (1994). DOI 10.1016/0020-0190(94)00045-X
29. Nomenclature Committee of the International Union of Biochemistry (NC-IUB): Nomenclature for incompletely specified bases in nucleic acid sequences. Recommendations 1984. Eur J Biochem **150**(1), 1–5. (1985)
30. Rytter, W.: The number of runs in a string. Inf. Comput. **205**(9), 1459–1469 (2007). DOI 10.1016/j.ic.2007.01.007
31. Zhang, H., Guo, Q., Iliopoulos, C.S.: Varieties of regularities in weighted sequences. In: B. Chen (ed.) Algorithmic Aspects in Information and Management, 6th International Conference, AAIM 2010, Weihai, China, July 19-21, 2010. Proceedings, *Lecture Notes in Computer Science*, vol. 6124, pp. 271–280. Springer (2010). DOI 10.1007/978-3-642-14355-7_28
32. Zhang, H., Guo, Q., Iliopoulos, C.S.: Locating tandem repeats in weighted sequences in proteins. BMC Bioinformatics **14**(S-8), S2. (2013)