Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 395 (2008) 298–310
Theoretical Computer Science
www.elsevier.com/locate/tcs

# Property matching and weighted matching☆

Amihood Amir[a,*], Eran Chencinski[a], Costas Iliopoulos[b], Tsvi Kopelowitz[a], Hui Zhang[b]

[a] *Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel*
[b] *Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom*

**Abstract**

In many pattern matching applications the text has some properties attached to its various parts. Pattern Matching with Properties (Property Matching, for short), involves a string matching between the pattern and the text, and the requirement that the text part satisfies some property. Some immediate examples come from molecular biology where it has long been a practice to consider special areas in the genome by their structures.

It is straightforward to do sequential matching in a text with properties. However, indexing in a text with properties becomes difficult if we desire the time to be output dependent. We present an algorithm for indexing a text with properties in $O(n \log |\Sigma| + n \log \log n)$ time for preprocessing and $O(|P| \log |\Sigma| + tocc_\pi)$ per query, where $n$ is the length of the text, $P$ is the sought pattern, $\Sigma$ is the alphabet, and $tocc_\pi$ is the number of occurrences of the pattern that satisfy some property $\pi$.

As a practical use of Property Matching we show how to solve Weighted Matching problems using techniques from Property Matching. Weighted sequences have recently been introduced as a tool to handle a set of sequences that are not identical but have many local similarities. The weighted sequence is a "statistical image" of this set, where we are given the probability of every symbol's occurrence at every text location. Weighted matching problems are pattern matching problems where the given text is weighted.

We present a reduction from Weighted Matching to Property Matching that allows off-the-shelf solutions to numerous weighted matching problems including indexing, swapped matching, parameterized matching, approximate matching, and many more. Assuming that one seeks the occurrence of pattern $P$ with probability $\epsilon$ in weighted text $T$ of length $n$, we reduce the problem to a property matching problem of pattern $P$ in text $T'$ of length $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.
© 2008 Elsevier B.V. All rights reserved.

*Keywords:* Pattern matching; Weighted indexing; Position-weight-matrices; Weighted swap matching

## 1. Introduction

The classical string matching problem of seeking all occurrences of a length $m$ pattern string $P$ in a length $n$ text string $T$ is one of the historically important problems in computer science. The last few decades, however, witnessed

☆ A preliminary version of this paper was presented at CPM 2006 (Barcelona, July 2006).

* Corresponding author.
*E-mail addresses:* amir@cs.biu.ac.il (A. Amir), chenche@cs.biu.ac.il (E. Chencinski), csi@dcs.kcl.ac.uk (C. Iliopoulos), kopelot@cs.biu.ac.il (T. Kopelowitz), hui@dcs.kcl.ac.uk (H. Zhang).

0304-3975/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.01.006

a tremendous development in the field of pattern matching due to the amazing proliferation of digital libraries and the Internet. The new applications gave rise to a plethora of interesting generalizations, approximations, and new models for pattern matching problems.

One of the technical problems that pattern matching has had to deal with is that of matching a pattern in a text with properties. The idea is that the pattern matching itself is insufficient, but the particular text substring that is matched also needs to satisfy a desired property. Some examples come from molecular biology, where it has long been a practice to consider special genome areas by their structure. Examples are repetitive genomic structures [25] such as *tandem repeats*, *LINEs* (Long Interspersed Nuclear Sequences) and *SINEs* (Short Interspersed Nuclear Sequences) [24]. Many problems in biology can be expressed as property matching problems, e.g., finding occurrences of a given pattern in a genome, provided it appears in a SINE, or LINE.

It is straightforward (as we show later) to solve sequential pattern matching with properties since the intersection of the properties and matching can be done in linear time. However, the problem becomes more complex when it is required to *index* a text with properties. The classical pattern matching problem [13,27] is that of finding all occurrences of *pattern* $P = p_1 p_2 \cdots p_m$ in *text* $T = t_1 t_2 \cdots t_n$, where $T$ and $P$ are strings over alphabet $\Sigma$. In the *indexing problem* we are given a large text that we want to preprocess in a manner that allows fast solution of the following queries: "Given a (relatively short) pattern $P$ find all occurrences of $P$ in $T$ in time proportional to $|P|$ and the number of occurrences".

The indexing problem and its many variants have been central in pattern matching [37,21,19,32,15]. It is also central to information retrieval [31]. However, when it comes to indexing a text with properties, we are now presented with a dilemma. If we use the conventional indexing techniques and then do the intersection with the properties, our worst case time may be very large in case where the pattern appears many times, and there may not be any final matches in case all the indexed matches do not satisfy the property.

In this paper we give a precise definition of pattern matching with properties and provide a data-structure that preprocesses the text in $O(n \log |\Sigma| + n \log \log n)$ time and supports queries in $O(|P| \log |\Sigma| + tocc_\pi)$ time per query, where $n$ is the text length, $P$ is the sought pattern, $|\Sigma|$ is the alphabet, and $tocc_\pi$ is the number of occurrences of $P$ that satisfy some property $\pi$. These are almost the same bounds that exist in the literature for ordinary indexing [37, 29,34,26].

We now turn to an apparently unrelated problem. Among the challenges that the pattern matching field is currently grappling with are those of *motif discovery*, and *local alignment*. Recently, the concept of *weighted sequences* was introduced as a suggested method of satisfying the above needs. A *weighted sequence* is essentially what is also called in the biology literature *Position Weight Matrix* (PWM for short) [33]. The *weighted sequence* of length $m$ is a $|\Sigma| \times m$ matrix that reports the frequency of each symbol in finite alphabet $\Sigma$ (nucleotide, in the genomic setting) for every possible location.

Originally, PWM sequences were used for relatively short sequences, e.g. binding sites, sequences resulting from multiple alignments, etc. Iliopoulos et al. [22] considered building very large Position Weight Matrices that correspond, for example, to complete chromosome sequences that have been obtained using a whole-genome shotgun strategy [35]. By keeping all the information the whole-genome shotgun produces, it should be possible to identify information that was previously undetected after being faded during the consensus step. This concept is true for other applications where local similarities are thus encoded. It is therefore necessary to develop adequate algorithms on weighted sequences, that can be an aid to the application researchers for solving various problems they are liable to encounter.

It turns out that handling weighted sequences is algorithmically challenging [23,14,22] even for simple tasks such as exact matching. It is certainly challenging to be able to answer more ambitious questions, such as *scaled weighted matching*, *swapped weighted matching*, *parameterized weighted matching* [28,36,30,10,16,2,12,7,8] as well as to *index* a weighted sequence.

*Scaled Matching* refers to the problem of finding all locations in the text where the pattern, proportionally enlarged according to an arbitrary scale, appears. Scaled matching is an important problem that was originally inspired by problems in Vision, yet has significance in motif extraction as well. For non-weighted matching, much work was done both in one-dimensional scaling [3,5] and two-dimensional scaling [11,6,4]. The general framework presented in this paper provides the first known algorithm for scaled matching in weighted sequences. Moreover, the algorithm's time is equivalent to the time of the scaled algorithm for non-weighted sequences.

*Swapped Matching* was motivated by a common error in typing. A more complex version of the swapped matching problem occurs in nature. The phenomenon of swaps occurs in gene mutations and duplications. The problem is defined as follows. Let a text string $T$ of $n$ symbols and a pattern string $P$ of $m$ symbols from alphabet $\Sigma$ be given. A *swapped version* $T'$ of $T$ is a length $n$ string derived from $T$ by a series of *local swaps*, (i.e. $t'_\ell \leftarrow t_{\ell+1}$ and $t'_{\ell+1} \leftarrow t_\ell$) where each element can participate in *no more than one swap*.

The *Pattern Matching with Swaps* problem is that of finding all locations $i$ for which there exists a swapped version $T'$ of $T$ where there is an exact matching of $P$ in location $i$ of $T'$. Extensive work has been done on efficient algorithms for swapped matching [28,36,30,10,16,2,12,7,8]. It is of interest to develop efficient algorithm for swapped matching in weighted sequences, especially in light of its importance in biology. Our general framework enables using the swapped algorithms for non-weighted sequences without a degradation in time.

*Parameterized Matching* was primarily motivated by software maintenance, where programs are to be considered "identical" even if variable names are different. Strings in this model are comprised of symbols from two disjoint sets $\Sigma$ and $\Pi$ containing fixed symbols and variable/parameter symbols respectively. In this paradigm, we seek *parameterized occurrences* i.e., occurrences up to renaming of the variable symbols, of a string in another. It turns out that the parameterized matching problem arises in other applications such as in image processing and computational biology, see [1].

Formally, parameterized pattern matching is defined as follows. A *p-string* is a string over $\Sigma \cup \Pi$. Two $p$-strings $s_1$ and $s_2$ of same length are said to *p-match* if there exists a bijection $f : \Pi_1 \leftrightarrow \Pi_2$, where $\Pi_1$ and $\Pi_2$ are the symbols from $\Pi$ in $s_1$ and $s_2$ respectively, such that the following holds: $s_1$ ($s_2$, respectively) equals $s_2$ ($s_1$, respectively) when any occurrence $x \in \Pi_1$ ($\Pi_2$, respectively) is replaced by $f(x)$ ($f^{-1}(x)$, respectively).

Finally, our framework enables *indexing* weighted sequences. The classical pattern matching problem is that of finding all occurrences of *pattern* $P = p_1 p_2 \cdots p_m$ in *text* $T = t_1 t_2 \cdots t_n$, where $T$ and $P$ are strings over alphabet $\Sigma$. Many optimal $O(n)$ time algorithms exit for that problem (e.g. [13,27]).

Two interesting variants were considered. The *dictionary matching problem* and the *indexing problem*. In the *dictionary matching problem*, a large set of patterns $D$, called the *dictionary* is given. The algorithm preprocesses $D$ in a manner that allows fast answers to queries of the form:

> Given input text $T$, find all locations of $T$ where an occurrence of some pattern of $D$ starts.

The *indexing problem* is, in some sense, the converse of the dictionary matching problem. Here we are given a large text that we want to preprocess in a manner that allows fast solution of the following queries:

> Given a (relatively short) pattern $P$ find all occurrences of $P$ in $T$ in time proportional to $|P|$ and the number of occurrences.

The indexing problem and its many variants has been central in pattern matching (e.g. [37,21,19,32,15]) since most current methods for handling weighted matching use techniques that are not conducive to indexing (such as convolutions), it is surprising that our framework enables indexing weighted sequences with the same time bounds as in the non-weighted case.

We develop a general framework that allows solving all the problems mentioned above. In particular this presents the first known algorithms for problems such as scaled matching, swapped matching and parameterized matching in weighted sequences. Since most current methods for handling weighted matching use techniques that are not conductive to indexing (e.g., convolutions), it is surprising that our framework also enables indexing weighted sequences with the same query time as in the non-weighted case.

These results are all enabled by a reduction of weighted matching to property matching. This reduction creates an ordinary text of length $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$ for the weighted matching problem of length $n$ text and desired probability $\epsilon$. Since the outcome of the reduction is an ordinary text with a property, then **all** pattern matching problems that can be solved in ordinary text and pattern can have their weighted versions solved with the time degradation of the reduction.

The indexing problem for weighted text becomes a problem of indexing an ordinary (longer) text with properties. We can now use the indexing text with properties result to solve weighted indexing as well.

The contributions of this paper are twofold. We first present the first solution for the property indexing problem and then we provide a reduction of the weighted matching problem to property matching problem allowing efficient solutions to many hitherto unsolved problems in weighted matching.

## 2. Property matching — definitions

For a string $T = t_1 \cdots t_n$, we denote by $T_{i\ldots j}$ the substring $t_i \cdots t_j$. The suffix $T_{i\ldots n}$ is denoted by $T^i$, and the suffix tree of $T$ is denoted by $ST(T)$. The leaf corresponding to $T^i$ in $ST(T)$ is denoted by $leaf(T^i)$. The label of an edge $e$ in $ST(T)$ is denoted by $label(e)$.

For a node $u$ in the suffix tree of a string $T$, we denote by $ST_u$ the subtree of the suffix tree rooted by $u$. The label of $u$ is the concatenation of the labels of the edges on the path from the root of the suffix tree to $u$, in the order they are encountered, and is denoted by $label(u)$.

We are now ready to define a property for a string.

**Definition 1.** A property $\pi$ of a string $T = t_1 \cdots t_n$ is a set of intervals $\pi = \{(s_1, f_1), \ldots, (s_t, f_t)\}$ where for each $1 \leq i \leq t$ it holds that: (1) $s_i, f_i \in \{1, \ldots, n\}$, and (2) $s_i \leq f_i$. The size of property $\pi$, denoted by $|\pi|$, is the number of intervals in the property (or in other words — $t$).

It is essential to realize how a property is given to us as input. A property is given in explicit form if we are given all of the pairs in the property. Due to lack of space, we omit the discussion about the input form of the properties from here, and simply assume that the properties are all in standard form as defined below.

**Definition 2.** A property $\pi$ for a string of length $n$ is said to be in standard form if: (1) it is in explicit form, (2) for any $1 \leq i \leq n$, there is at most one $(s_k, f_k) \in \pi$ such that $s_k = i$, and (3) $s_1 < s_2 < \cdots < s_{|\pi|}$.

## 3. General pattern matching with properties

In this section we will define and formulate the notion of general pattern matching with properties. This is mainly engulfed by the following definition.

**Definition 3.** Given a text $T = t_1 \cdots t_n$ with property $\pi$, a pattern $P = p_1 \cdots p_m$, and a definition of a matching $\alpha$, called $\alpha$-matching, we say that $P$ $\alpha$-matches $T_{i\ldots j}$ under property $\pi$ if $P$ $\alpha$-matches $T_{i\ldots j}$, and there exists $(s_k, f_k) \in \pi$ such that $s_k \leq i$ and $j \leq f_k$.

This definition explains why we previously required that in a standard form only one interval in a property begins at specific location, as we will only be interested in the longest interval at a given location.

The following definition will assist us in solving property matching problems.

**Definition 4.** For a property $\pi$ of a string $T = t_1 \cdots t_n$, the end location of $1 \leq i \leq n$, denoted by $end(i)$, is defined to be the maximal $f_k$ such that $(s_k, f_k) \in \pi$ and $s_k \leq i \leq f_k$. If no such $f_k$ exists, we say that $end(i) = NIL$.

Note that $end(i)$ can easily be calculated for all locations $i$ in $T$ in time $O(n)$ (recall that $\pi$ is given in standard form). Now, given a text $T = t_1 \cdots t_n$ and a pattern $P = p_1 \cdots p_m$, if there exists an algorithm for an $\alpha$-matching problem that runs in time $O(g_\alpha(n, m))$, then given a text $T$ with property $\pi$, and pattern $P$, we can find all $T_{i\ldots j}$ that $\alpha$-match $P$ under $\pi$ in time $O(g_\alpha(n, m) + n) = O(g_\alpha(n, m))$ (we assume the entire text was at least scanned in order to solve the matching problem). This is done as follows. First, we find $end(i)$ for each location $i$ in the text. Then, we find all the $T_{i\ldots j}$ that $\alpha$-match $P$, and for each such $T_{i\ldots j}$ we check whether $end(i) \geq j$. If so, clearly $T_{i\ldots j}$ $\alpha$-matches $P$ under $\pi$, and if not, clearly $T_{i\ldots j}$ does not $\alpha$-match $P$ under $\pi$.

However, the above reduction does not suffice for the property indexing problem (defined below). Before explaining why, we first provide a formal definition of the property indexing problem.

**Definition 5** (*Property Indexing Problem (PIP)*). Given a text string $T = t_1 \cdots t_n$ with property $\pi$, preprocess $T$ such that on-line queries of the form "find all locations where a pattern string $P$ occurs in $T$ under $\pi$" can be answered in time proportional to the size of the *pattern* (rather than the text) and the output.

The problem with the PIP is that known indexing data-structures do not suffice. For example, given a suffix tree for $T$, we can find all of the occurrences of $P$ in $T$ in time $O(|P| \log |\Sigma| + tocc)$ where $tocc$ is the number of the occurrences. However, $tocc$ is not the number of occurrences of $P$ in $T$ under $\pi$; it includes also the occurrences of $P$ in $T$ that are not occurrences under $\pi$. We could solve this problem by also preprocessing $end(i)$ for all locations $i$ in $T$ as we did before. However, this would require scanning all of the occurrences of $P$ in $T$ (taking $O(tocc)$ time), and

we would like to answer indexing queries in time dependent on $tocc_\pi$, where $tocc_\pi$ is the number of occurrences of $P$ in $T$ under $\pi$, which might be much smaller than $tocc$. Also, keep in mind that we want a solution that takes minimal preprocessing time, and requires only linear space. This is the problem addressed by our new data-structure.

In the next sections we will define our data-structure, show how it is constructed in time $O(n \log |\Sigma| + n \log\log n)$, and finally, show how an indexing query can be answered in time $O(m \log |\Sigma| + tocc_\pi)$.

## 4. The property suffix tree

We now define the data-structure used for solving the PIP. The data-structure we present is based on the suffix tree — thus, we name it the Property Suffix Tree, or PST for short. The construction is for a text $T = t_1 \cdots t_n$ with property $\pi$. The idea is based on a lemma that we provide following the next definition.

**Definition 6.** For a string $T$ with property $\pi$ and a node $u$ in the suffix tree of $T$, we denote by $S_u^\pi$ the maximal set of locations $\{i_1, \ldots, i_\ell\} \subseteq \{1, \ldots, n\}$ such that for every $i_j \in S_u^\pi$ we have that: (1) $leaf(T^{i_j})$ is in $ST_u$, and (2) if $end(i_j) \neq NIL$ then $end(i_j) - i_j > |label(u)|$.

**Lemma 1.** *Let $T$ be a string with property $\pi$, and let $u$ and $v$ be two nodes in the suffix tree of $T$ such that $v$ is $u$'s parent, then $S_u^\pi \subseteq S_v^\pi$.*

**Proof.** The proof follows from Definition 6. For any location $i_j \in S_u^\pi$ we know $leaf(T^{i_j})$ is in $ST_u$, thus it is also in $ST_v$. We also know that $end(i_j) - i_j > |label(u)|$. Being that $|label(u)| > |label(v)|$, we have that $end(i_j) - i_j > |label(u)| > |label(v)|$. Due to the maximality of $S_v^\pi$, it must be that $i_j \in S_v^\pi$. □

**Corollary 1.** *For a string $T$ with property $\pi$, the path from the root of $ST(T)$ to $leaf(T^i)$ can be split into the following two paths: (1) the path consisting of all nodes $u$ such that $i \in S_u^\pi$, and (2) the path consisting of all nodes $u$ such that $i \notin S_u^\pi$.*

**Definition 7.** Consider the $i$th suffix of $T$ and the two paths from Corollary 1. Let $v$ be the deepest node on the first path. The location of $i$ in the PST of $T$ is defined as follows. If $end(i) - i = |label(v)| - 1$ or $end(i)$ is $NIL$ then $loc(i) = v$. Otherwise, $loc(i)$ is the edge connecting the two paths.

The idea behind the PST is to move each suffix $T^i$ in $ST(T)$ up to $loc(i)$. We will later show why this solves the PIP. We now define the PST using an overview construction. First, we construct $ST(T)$ using, for example, [37]. Then, for every suffix $T^i$ find $loc(i)$, and maintain a list of locations for each edge $e$ consisting of all $i$ such that $e = loc(i)$ and for each node $u$ consisting of all $i$ such that $u = loc(i)$. We denote these lists by $suf(e)$ and $suf(u)$ respectively. Next, we mark each node $u$ in $ST(T)$ such that either $suf(u)$ is not an empty list, or $u$ is connected to some edge $e$ where $suf(e)$ is not an empty list, or $u$ is an ancestor of a marked node. Now, we delete all of the nodes that are not marked, and compress non-branching paths in the remaining tree to one edge (like we do in suffix trees). Of course, during the compression of a path into an edge, we must concatenate all of the $suf(u)$ and $suf(e)$ for all nodes $u$ and edges $e$ on the path, except for the last node. The concatenation of all of those lists forms the list of locations $loc(e')$ for the new edge $e'$ that will replace the non-branching path. Finally, we will be interested in ordering $suf(e)$ for the remaining edges in order to allow efficient querying. This will be explained later.

Note that except for the stage in which we construct $suf(e)$ and $suf(u)$ for the edges $e$ and nodes $u$ in $ST(T)$ and the ordering of the lists of locations, the rest of the algorithm can be easily implemented to take $O(n \log |\Sigma|)$ by building a suffix tree and using a constant number of depth-first searches (DFS). Also note that the size of the data-structure is clearly linear in the size of $T$. Thus, it remains to show how to construct $suf(e)$ and $suf(u)$ for the edges $e$ and nodes $u$ in $ST(T)$, and how to order them while allowing us to answer queries efficiently. This is explained in the next two subsections.

### 4.1. Constructing lists of locations

We now show how to construct $suf(e)$ and $suf(u)$ for every edge $e$ and every node $u$ in $ST(T)$. In the following subsection we show how to order $suf(e)$ in a way that will allow efficient querying.

In order to find $loc(i)$ for every suffix $T^i$, we use the weighted ancestor queries that were presented in [18], and improved upon in [9]. The weighted ancestor problem is defined as follows:

**Definition 8.** Let $T$ be a rooted tree where each node $u$ has an associated value $value(u)$ from an ordered universe $U$ such that if $v$ is the parent of $u$ then $value(v) < value(u)$. The weighted ancestor problem is given a query of the form $WA(u, i)$ where $u$ is a node in $T$ and $i \in U$, return the node $v$ that is the lowest ancestor of $u$ such that $value(v) < i$.

Clearly, if we set the value of a node $u$ to be $|label(u)|$, then given a leaf $leaf(T^i)$, the answer to the query $WA(leaf(T^i), end(i) - i)$ will either give us a node that is $loc(i)$, or a node that is connected to the edge that is $loc(i)$. In the later case, we can easily find $loc(i)$ in $O(\log|\Sigma|)$ time. In [9] the weighted ancestor problem was solved for suffix trees taking $O(n)$ preprocessing time, and $O(\log\log n)$ query time. Thus, we can find $loc(i)$ for all $T^i$'s in $O(n(\log\log n + \log|\Sigma|))$ time. However, the suffixes on the edges are not ordered in a way that would allow efficient indexing queries. We cannot simply order the suffixes by descending $loc(i) - i$ because this would require sorting, and would take too much time (we would need to sort the locations on every edge in the tree according to the appropriate values). To solve this problem, we show in Section 4.2 how to preprocess a set of $n'$ elements in $O(n')$ time such that given a value whose rank[1] in the set is $k$, we can find all of the elements less than or equal to that value in $O(k)$ time. In Section 4.3 we will show how this helps us answer indexing queries efficiently. Thus, we will run this algorithm on every edge in the tree, taking a total of linear time. Finally, the time required for constructing the PST is $O(n\log|\Sigma| + n\log\log n)$. Note that for constant size alphabets we are dominated by the $n\log\log n$ factor.

### 4.2. Ordering the suffixes on an edge

As we previously mentioned, we require a scheme such that given a set of $n'$ elements we can preprocess those elements in $O(n')$ time such that given a value whose rank in the set is $k$, we can find all of the elements less than or equal to that value in $O(k)$ time. To solve this algorithm we use the fact that finding the median of a set of numbers can be done in linear time (e.g., by [17]). The preprocessing is as follows. First find the median of the set, and separate the set to the set of values smaller than the median, and the set of the values that greater than the median (for simplicity, we assume all values are distinct). For the set of items with value greater than the median, we put them in an array of size $n'$, in the second part of the array. We recursively do the same for the elements less than the median, each time putting the items greater than the median in the rightmost part of the unfilled array, until we reach a set of size one, and we put the remaining element in the first location in the array. Note that the time required is $O(\sum_{i=0}^{\log n'} \frac{n'}{2^i}) = O(n')$.

Now, given a query value $t$ with rank $k$, we proceed as follows. We begin by comparing $t$ with the first location. If $t$ is smaller, than we output an empty set. If $t$ is larger, we output the first element as part of the output and continue on to scan the next two elements in the array. If they are both less than or equal to $t$, we output them both, and continue on to the next four elements. We continue on such that at the $i$th iteration, if all of the $2^{i-1}$ elements are less than or equal to $t$, we output them all, and continue to the next $2^i$ items. This continues until we reach some item whose value is greater than $t$. Say this happens at iteration number $i'$. In such a case, we continue to scan all of the $2^{i'-1}$ items of the iteration, outputting only those items with value less than or equal to $t$, and then we are done.

Clearly, we output all elements that are less than or equal to $t$, as once we find an element that is greater than $t$ in the $i'$ iteration, we know that all the rest of the elements in the array (located after the $2^{i'-1}$ elements of the current iteration) have value greater than $t$ (this follows directly from the way we arranged the array, dividing it around the median). Moreover, the running time is $O(k)$ as if we stop at iteration $i'$, this means we output at least $\sum_{i=1}^{i'-1} 2^{i-1} = \Omega(2^{i'})$, and the running time is at most $\sum_{i=1}^{i'} 2^{i-1} = O(2^{i'})$. Finally, note that the same type of technique can be used if we are interested in finding all the elements that have value larger or equal to $t$. We will actually be interested in this version of the problem for ordering the suffixes on the edges.

### 4.3. Answering indexing queries

In this section we describe how to answer indexing queries in $O(m\log|\Sigma| + tocc_\pi)$. But first, for a node $u$ in the PST we denote by $PST_u$ the subtree of the PST rooted by $u$. The indexing query is answered as follows. We first begin by searching the PST like we search a suffix tree, until we reach a node or an edge. If we reach a node $u$, we run a DFS on $PST_u$, outputting $suf(w)$ and $suf(e')$ for every node $w$ and every edge $e'$ in $PST_u$. If when searching we reach an edge $e = (u, v)$ where we match the first $\ell$ characters of $label(e)$, then we first output $suf(w)$ and $suf(e')$

[1] The rank of a value in a set is the number of elements in the set less than or equal to the value.

for every node $w$ and every edge $e'$ in $PST_v$ using a DFS, and we also output every location $i$ in $suf(e)$ such that $end(i) - i > |label(u)| + \ell$. In order to accomplish the second part, we use the scheme from Section 4.2. it remains to show that the additional amount of time spent (i.e. except for the search part that takes $O(m \log |\Sigma|)$) is linear in the size of the output. This follows from the following lemma.

**Lemma 2.** *Let $PST(T)$ be the PST of a string $T$ under property $\pi$. Then in the subtree of any node in $PST(T)$, the size of the subtree is linear in the number of locations in the union of $suf(w)$ and $suf(e')$ for every node $w$ and every edge $e'$ in the subtree.*

**Proof.** Let $u$ be a node in $PST(T)$. For any leaf $\ell$ in the subtree rooted at $u$ there is either a location in $suf(\ell)$ or a location in $suf(e)$ where $e$ is the only edge touching $\ell$. This is because otherwise $\ell$ would not be in the PST (by definition). Thus, the number of locations in question is at least the number of leaves in the subtree. Now, being that all of the internal nodes in $PST(T)$ are branching nodes (they have at least two children), then the size of the subtree rooted by $u$ is at most twice the number of leaves in the subtree. Therefore, the number of nodes is at most twice a lower bound on the number of locations — as needed. □

**Theorem 1.** *The PIP can be solved in $O(n \log |\Sigma| + nlog \log n)$ preprocessing time, using linear space, where the query time is $O(m \log |\Sigma| + tocc_\pi)$.*

In the following sections we consider weighted matching problems and show a general framework for solving weighted matching problems using property matching.

## 5. Weighted matching — definitions

**Definition 9.** A weighted sequence $T = t_1 \cdots t_n$ over alphabet $\Sigma$ is a sequence of sets $t_i$, $i = 1, \ldots, n$. Every $t_i$ is a set of pairs $(s_j,\ \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having symbol $s_j$ at location i. Formally,

$$t_i = \left\{ (s_j,\ \pi_i(s_j)) \mid s_j \neq s_l \text{ for } j \neq l,\ \text{and} \sum_j \pi_i(s_j) = 1 \right\}.$$

**Definition 10.** Given a pattern $P = p_1 \cdots p_m$ over alphabet $\Sigma$, we say that the solid pattern $P$ (or simply pattern $P$) occurs at location i of a weighted text $T$ with probability of at least $\epsilon$ if $\prod_{j=1}^{m} \pi_{i+j-1}(p_j) \geq \epsilon$, where $\epsilon$ is a given parameter which we call the threshold probability.

Notice that all characters having probability of appearance less than $\epsilon$ are not of interest to us, since any pattern using such a character will also have probability of appearance less than $\epsilon$, which is below the threshold probability. Therefore, we are only interested in characters having probability of appearance of at least $\epsilon$. We call such characters **heavy characters**.

Recently, there has been a philosophical debate about the weighted matching model. Clearly, there is an underlying assumption of independence of events, and the model is the Bernoulli Model, which fits some applications better than others. It is thought by some (e.g., [20]) that DNA is not modelled well by the Bernoulli model. Nevertheless, our concern in this paper is with the algorithmic and combinatorial issues, rather than the application. We do think it is of interest to extend our results to $k$-order Markov models, and we are working on such extensions, but this is not the aim of the current paper.

**Definition 11.** Given $0 < \epsilon \leq 1$, we classify each location $i$, $1 \leq i \leq n$, in the text into the following three categories: (1) Solid positions where there is one (and only one) character at location $i$ with probability of appearance exactly 1, (2) Leading positions where there is at least one character at location $i$ with probability of appearance greater than $1 - \epsilon$ (and less than 1), and (3) Branching positions where all characters at location $i$ have probability of appearance at most $1 - \epsilon$.

Notice that if $\epsilon \leq \frac{1}{2}$, then at every solid and leading position there is only one heavy character since only one character can have probability of appearance greater than $1 - \epsilon \geq \frac{1}{2}$, whereas in a branching position there maybe several heavy characters. However, if $\epsilon > \frac{1}{2}$ there are no heavy characters in a branching position since all characters have probability of appearance of at most $1 - \epsilon < \epsilon$.

In the following section we define the notions of Maximal Factors and Extended Maximal Factors and show how they are used in the reduction from weighted matching to property matching.

## 6. Maximal factors and extended maximal factors

A weighted pattern matching problem is a pattern matching problem where the text is weighted. The idea behind our framework is to create a regular text from the weighted text in a way that we can run regular pattern matching algorithms on the regular text while ensuring that the occurrences appear with probability of at least $\epsilon$. In order to do so, we first define the notion of maximal factor.

**Definition 12.** Let $T = t_1 \cdots t_n$ be a weighted text and let $X = x_1 \cdots x_l$ be a string. We denote $\pi_i(X) = \pi_i(x_1) \times \cdots \times \pi_{i+l-1}(x_l)$. Given $0 < \epsilon \leq 1$, we say that a string, $X$, is a maximal factor of $T$ starting at location $i$ if the following conditions hold: (1) $\pi_i(X) \geq \epsilon$, (2) *if* $i > 1$, *then* $\pi_{i-1}(s_j) \times \pi_i(X) < \epsilon$ for all $s_j \in \Sigma$, and (3) if $i + l \leq n$, then $\pi_i(X) \times \pi_{i+l}(s_j) < \epsilon$ for all $s_j \in \Sigma$.

In other words, a maximal factor starting at location $i$ is a string that when aligned to location $i$ has probability of appearance at least $\epsilon$. However, if we extend the string by even one character to the right and align it to location $i$ or if we extend the string by even one character to the left and align it to location $i - 1$, then the probability appearance of the string drops below $\epsilon$.

A straightforward approach for transforming the weighted text T to a regular text would be to simply find all the maximal factors of the text and concatenate them to a new regular text $T'$ (of course we will need some kind of a delimiter character to separate between the factors). The advantage of this approach is that every pattern that appears in $T'$ appears also in $T$ with probability of at least $\epsilon$, since a maximal factor has probability of appearance at least $\epsilon$ and so have all of its substrings. Unfortunately, this approach does not suffice. It can be shown (due to lack of space details are omitted) that the total length of all maximal factors of a weighted text $T = t_1 \cdots t_n$ could be $\Omega(n^2)$, which is rather large. Therefore, we define the notion of extended maximal factor, and show a better upper bound on the total length of all extended maximal factors. In order to define the extended maximal factor we use the Leading to Solid Transformation.

**Definition 13.** The Leading to Solid Transformation of a weighted sequence $T = t_1 \cdots t_n$ denoted LST(T), is a weighted sequence $T' = t'_1 \cdots t'_n$ such that:

$$t'_i = \begin{cases} t_i & \text{if } i \text{ is a solid or a branching position} \\ \{(\sigma, 1)\} & \text{if } i \text{ is a leading position and } \sigma \text{ is a heavy character} \\ \phi & \text{if } i \text{ is a leading position and there are no heavy characters} \end{cases}$$

In essence, $LST(T)$ is the same as $T$, where all leading positions become solid. The only exception is when all characters in a leading position are not heavy, thus, we ignore that location (set to by $\phi$) and treat each part of $LST(T)$ divided by $\phi$ separately. For the rest of this paper, we assume $LST(T)$ has no $\phi$'s.

Notice that this transformation is uniquely defined, since either $\epsilon \leq \frac{1}{2}$ in which case there is one (and only one) character with probability $> 1 - \epsilon$, thus, it is also the only heavy character at that location or $\epsilon > \frac{1}{2}$ in which case at every location there is at most one heavy character.

Another important observation is that the size of $LST(T)$ is linear in the size of $T$ and can easily be built in linear time. The LST transformation leads us to the following definition.

**Definition 14.** Given $0 < \epsilon \leq 1$ and a weighted text $T$, we say that a string $X$ is an extended maximal factor of $T$ starting at location $i$ if $X$ is a maximal factor of $LST(T)$ starting at location $i$.

We now prove a few properties on maximal factors and extended maximal factors, that will help us in bounding the total length of all extended maximal factors of a weighted text.

**Lemma 3.** *Given* $0 < \epsilon \leq 1$ *and a weighted text* $T$*, there are at most* $\lfloor \frac{1}{\epsilon} \rfloor$ *heavy characters at a branching position.*

**Proof.** A heavy character has probability of appearance of at least $\epsilon$. Therefore, at each location, at most $\lfloor \frac{1}{\epsilon} \rfloor$ characters have probability of appearance of at least $\epsilon$, otherwise the sum of all probabilities would be $>1$. □

**Definition 15.** Given $0 < \epsilon \leq 1$ and a weighted text $T$, we say that a maximal factor $X = x_1 \cdots x_l$ passes by location $i$ of $T$, if $X$ starts at location $i'$ such that $i' \in [i - l + 1, i]$.

**Lemma 4.** *Given $0 < \epsilon \leq 1$ and a weighted text $T$, a maximal factor of $T$ passes by at most $O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$ branching positions.*

**Proof.** Denote by $l_b$ the maximal number of branching positions passed by a maximal factor. By definition, $(1-\epsilon)^{l_b} \geq \epsilon$, since the maximal probability of a character in a branching position is $1 - \epsilon$.
Therefore, $l_b \leq \lfloor \frac{\log \epsilon}{\log(1-\epsilon)} \rfloor$. Using the fact that $\log(\frac{1}{1-\epsilon}) = \Theta(\epsilon)$ (see Lemma 5), we get that $l_b \leq \lfloor \frac{\log \epsilon}{\log(1-\epsilon)} \rfloor \leq \frac{\log \epsilon}{\log(1-\epsilon)} = \frac{-\log \epsilon}{-\log(1-\epsilon)} = \frac{\log \frac{1}{\epsilon}}{\log(\frac{1}{1-\epsilon})} = \frac{\log \frac{1}{\epsilon}}{\Theta(\epsilon)} = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$. □

**Lemma 5.** $\log(\frac{1}{1-\epsilon}) = \Theta(\epsilon)$.

**Proof.** We first denote $\frac{1}{\epsilon}$ by $k$. Now, we can formulate the claim in the following manner: $\log(\frac{1}{1-\frac{1}{k}}) = \log(\frac{1}{\frac{k-1}{k}}) = \log(k) - \log(k-1) = \Theta(\frac{1}{k})$.
It is a well known fact that $\lim_{k\to\infty} \sum_{i=1}^{k} \frac{1}{i} = \ln(k) + \gamma$, where $\gamma$ is Euler's constant. In other words, for every $\epsilon_0 > 0$ there exists some $k_0$ such that for every $k \geq k_0$, $|\ln(k) + \gamma - \sum_{i=1}^{k} \frac{1}{i}| < \frac{1}{2}\epsilon_0$.
$\Rightarrow |\ln(k) - \ln(k-1) - \frac{1}{k}| = |(\ln(k) + \gamma) - (\ln(k-1) + \gamma) - \frac{1}{k}|$
$= |(\ln(k) + \gamma) - (\ln(k-1) + \gamma) - (\sum_{i=1}^{k} \frac{1}{i} - \sum_{i=1}^{k-1} \frac{1}{i})|$
$= |(\ln(k) + \gamma - \sum_{i=1}^{k} \frac{1}{i}) - (\ln(k-1) + \gamma - \sum_{i=1}^{k-1} \frac{1}{i})|$
$\leq |(\ln(k) + \gamma - \sum_{i=1}^{k} \frac{1}{i})| + |(\ln(k-1) + \gamma - \sum_{i=1}^{k-1} \frac{1}{i})| < \frac{1}{2}\epsilon_0 + \frac{1}{2}\epsilon_0 = \epsilon_0$
$\Rightarrow \lim_{k\to\infty}(\ln(k) - \ln(k-1)) = \frac{1}{k}$
$\Rightarrow \ln(k) - \ln(k-1) = \Theta(\frac{1}{k})$.
Therefore, $\log(k) - \log(k-1) = \log_2(e) \cdot (\ln(k) - \ln(k-1)) = \Theta(\frac{1}{k})$. □

**Definition 16.** Given $0 < \epsilon \leq 1$ and a weighted text $T$, we say that location $i$ is a starting location of $T$, if $i$ contains at least one heavy character and either $i = 1$ or $i > 1$ and $i - 1$ is not a solid position.

Observe that a maximal factor of $T$ always starts at a starting location, otherwise it could be extended to the left with solid positions without decreasing the probability of appearance, which contradicts the maximality of the factor.

The following lemma bounds the number of maximal factors starting from a starting location in a weighted text $T$, such that $T$ has no leading positions. The fact that $T$ has no leading positions implies that this is true for $LST(T)$ of any weighted text $T$, and thus actually bounds the number of extended maximal factors starting from any location in $T$.

**Lemma 6.** *Given $0 < \epsilon \leq 1$ and a weighted text $T$ such that $T$ has no leading positions, there are at most $\lfloor \frac{1}{\epsilon} \rfloor$ maximal factors starting at a starting location.*

**Proof.** The number of maximal factors starting at location $i$ does not depend on solid positions since they do not split a maximal factor. Therefore, we discard solid positions and assume all locations are branching.
Denote by $P = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_t\}$ the set of probabilities of all the strings that appear in $T$ with positive probabilities, such that $\epsilon_1 > \epsilon_2 > \cdots > \epsilon_t$. We will first prove the lemma for every threshold probability $\epsilon \in P$, then we will prove it for every threshold probability $\epsilon \notin P$.

Let $i$ be some location in T, we will prove that for every threshold probability $\epsilon_j \in P$ the number of maximal factors starting at location $i$ is at most $\lfloor \frac{1}{\epsilon_j} \rfloor$, by induction on $j$.

For $j = 1$: Since $\epsilon_1$ is the largest probability of any string, it is clear that it is the probability of a string of length exactly one. Therefore, by Lemma 3, there are at most $\lfloor \frac{1}{\epsilon_1} \rfloor$ heavy characters at location $i$, which in our case are also the maximal factors starting at location $i$ with probability of appearance at lest $\epsilon_1$ (and since $\epsilon_1$ is maximal — the probability is exactly $\epsilon_1$).

For $j > 1$: Assuming the claim is true for all threshold probabilities $\epsilon_1, \ldots, \epsilon_{j-1}$, we will prove it for $\epsilon_j$. Denote by $s_1, \ldots, s_r$ the heavy characters at location $i$ and let $\alpha_1, \cdots, \alpha_r$ be matching probabilities of appearance of each character.

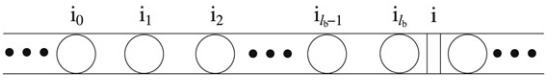


Fig. 1. Locations $i_0, \ldots, i_{l_b}$ are all branching positions. Since a maximal factor can pass by at most $l_b$ positions, only the starting locations $i_0 + 1, i_1 + 1, \ldots, i_{l_b} + 1$ are relevant.

Notice that $\sum_{v=1}^{r} \alpha_v \leq 1$ and since $s_1, \ldots, s_r$ are heavy characters, $\alpha_v \geq \epsilon_j$ for $v = 1, \ldots, r$. Denote by $MF_{\epsilon_j}(i)$ the number of maximal factors starting at location $i$ with probability of appearance at least $\epsilon_j$, and denote $MF_{\epsilon_j}(i, s_v)$ to be the number of maximal factors starting at location $i$ with probability of appearance at least $\epsilon_j$ after choosing character $s_v$ at location $i$. Clearly $\sum_{v=1}^{r} MF_{\epsilon_j}(i, s_v) = MF_{\epsilon_j}(i)$.

Notice that $MF_{\epsilon_j}(i, s_v) = MF_{\epsilon_j/\alpha_v}(i + 1)$ since after we choose character $s_v$ we can extend the maximal factor up to probability of appearance at most $\frac{\epsilon_j}{\alpha_v}$ (otherwise we will get total probability of appearance at location $i$ greater than $\alpha_v \cdot \frac{\epsilon_j}{\alpha_v} = \epsilon_j$).

Let $\epsilon_j^v$ be the smallest probability in $P$ such that $\epsilon_j^v \geq \frac{\epsilon_j}{\alpha_v}$, then by the induction hypothesis, since $\epsilon_j^v > \epsilon_j$ and since $\epsilon_j^v \in P$, we get that $MF_{\epsilon_j}(i) = \sum_{v=1}^{r} MF_{\epsilon_j}(i, s_v) = \sum_{v=1}^{r} MF_{\epsilon_j/\alpha_v}(i + 1) = \sum_{v=1}^{r} MF_{\epsilon_j^v}(i + 1) \leq \sum_{v=1}^{r} \lfloor \frac{1}{\epsilon_j^v} \rfloor \leq \sum_{v=1}^{r} \lfloor \frac{\alpha_v}{\epsilon_j} \rfloor \leq \sum_{v=1}^{r} \lfloor \frac{1}{\epsilon_j} \rfloor \cdot \alpha_v = \lfloor \frac{1}{\epsilon_j} \rfloor \cdot \sum_{v=1}^{r} \alpha_v \leq \lfloor \frac{1}{\epsilon_j} \rfloor$.

This concludes the proof for every $\epsilon \in P$. For a threshold probability $\epsilon' \notin P$, there are 3 possible cases:

**Case 1.** $\epsilon' < \epsilon_t$**:** In this case, any factor has probability of appearance greater than $\epsilon'$. Therefore, there are at most $\lfloor \frac{1}{\epsilon_t} \rfloor \leq \lfloor \frac{1}{\epsilon'} \rfloor$ maximal factors starting at each location.

**Case 2.** $\epsilon' > \epsilon_1$**:** In this case, no factor has probability of appearance greater than or equal to $\epsilon'$. Therefore, the number of maximal factors starting at each location is exactly $0 \leq \lfloor \frac{1}{\epsilon'} \rfloor$.

**Case 3.** $\epsilon_j > \epsilon' > \epsilon_{j+1}$ **for some** $1 \leq j < t$**:** In this case, since no factor has probability of appearance at least $\epsilon'$ and less than $\epsilon_j$, the number of maximal factors starting at location $i$ with probability of appearance at least $\epsilon'$ is exactly the number of maximal factors starting at location $i$ with probability of appearance at least $\epsilon_j$ which in turn is at most $\lfloor \frac{1}{\epsilon_j} \rfloor \leq \lfloor \frac{1}{\epsilon'} \rfloor$. □

**Lemma 7.** *Given* $0 < \epsilon \leq 1$ *and a weighted text* $T$ *such that* $T$ *has no leading positions, the number of maximal factors passing by each location* $i$ *in the text is at most* $O((\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.

**Proof.** Let us look at some location $i$ in the text. Notice that since $T$ has no leading positions, a starting location can only be at the beginning of a sequence of solid positions, and not in the middle of such a sequence (by definition of a starting location).

By Lemma 4 there are at most $l_b$ branching positions passed by each maximal factor, where $l_b = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon})$. In other words between location $i$ and every starting location that contains a maximal factor passing by location $i$ there are at most $l_b$ branching positions. Therefore, there are at most $l_b + 1$ starting locations that can have a maximal factor containing location $i$ (see Fig. 1).

By Lemma 6, at most $\lfloor \frac{1}{\epsilon} \rfloor$ maximal factors start at a starting location. Therefore, the total number of maximal factors passing by location $i$ is at most $(l_b + 1) \cdot \lfloor \frac{1}{\epsilon} \rfloor = O(\frac{1}{\epsilon} \log \frac{1}{\epsilon}) \cdot \lfloor \frac{1}{\epsilon} \rfloor = O((\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$. □

The following theorem bounds the total length of all extended maximal factors.

**Theorem 2.** *Given* $0 < \epsilon \leq 1$ *and a weighted text* $T$*, the total length of all extended maximal factors of* $T$ *is at most* $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.

**Proof.** This follows immediately from Lemma 7. □

We now show that this analysis is tight up to a logarithmic factor.

**Lemma 8.** *Given* $0 < \epsilon \leq 1$ *and a weighted text* $T$*, the total length of all extended maximal factors of* $T$ *is* $\Omega(n(\frac{1}{\epsilon})^2)$.

**Proof.** Consider the following example of a weighted text, where $\epsilon$ is the threshold probability:

Let $T = \alpha_{1_1}, \alpha_{1_2}, \ldots, \alpha_{1_g}, \beta_{1_1}, \beta_{1_2}, \ldots, \beta_{1_h}, \alpha_{2_1}, \alpha_{2_2}, \ldots, \alpha_{2_g}, \beta_{2_1}, \beta_{2_2}, \ldots, \beta_{2_h}, \ldots$ be the weighted text of length $n$, such that: $\alpha_{i_j} = [(s_a, 1/2), (s_b, 1/2)]$ where $s_a, s_b \in \Sigma$, $\beta_{i_j} = [(s_c, 1 - \epsilon), (s_d, \epsilon)]$ where $s_c, s_d \in \Sigma$, $g = \lfloor \log \frac{1}{\epsilon} - 1 \rfloor$, and $h = \lfloor \frac{1}{\log(1/(1-\epsilon))} \rfloor$.

Notice that all locations are branching positions. Therefore: (a) all extended maximal factors are also maximal factors, (b) all locations are starting locations, (c) from each starting location there is at least one extended maximal factor that passes by at least $g$ locations of the type $\alpha_{i_j}$ and $h$ locations of the type $\beta_{i_j}$ since $(1/2)^g \times (1-\epsilon)^h \geq (1/2)^{\log\frac{1}{\epsilon}-1} \times (1-\epsilon)^{\frac{1}{\log(1/(1-\epsilon))}} = 2\epsilon \times (1-\epsilon)^{\frac{-1}{\log(1-\epsilon)}} = 2\epsilon \times 2^{-1} = \epsilon$, and (d) there are $\Omega(\frac{1}{\epsilon})$ maximal factors starting at a starting location having length of $\Omega(\frac{1}{\epsilon})$ since by claim (c) each maximal factor passes by at least $g = \lfloor \log\frac{1}{\epsilon} - 1 \rfloor$ locations of the type $\alpha_{i_j}$ giving us $2^g = \Omega(\frac{1}{\epsilon})$ maximal factors and since we pass by at least $h = \lfloor \frac{1}{\log(1/(1-\epsilon))} \rfloor = \lfloor \frac{1}{\Theta(\epsilon)} \rfloor = \Theta(\frac{1}{\epsilon})$ (by Lemma 5) locations of the type $\beta_{i_j}$, each factor is of length $\Omega(\frac{1}{\epsilon})$.

Now, since there are $\Omega(\frac{1}{\epsilon})$ maximal factors of length $\Omega(\frac{1}{\epsilon})$ starting at each location (claim (d)) and since each location is a starting location (claim (b)) we get that the total length of all (extended) maximal factors of $T$ is $\Omega(n(\frac{1}{\epsilon})^2)$. □

In the next section we show how to efficiently find all extended maximal factors of a weighted sequence.

## 7. Finding all extended maximal factors in a weighted sequence

Let $T = t_1 \cdots t_n$ be a weighted sequence such that $\{s_i^1, s_i^2, \ldots, s_i^{k_i}\}$ is the set of characters appearing at location $i$ with positive probability, and $\{\alpha_i^1, \alpha_i^2, \ldots, \alpha_i^{k_i}\}$ is the matching set of probabilities of the $s_i^j$'s.

We present a simple brute-force algorithm that given a weighted text $T$ and a threshold probability $\epsilon$, outputs all extended maximal factors in $T$. The algorithm first calculates $T' \leftarrow LST(T)$ in linear time (as mentioned above). Then, starting from each starting location $i$ in $T'$, we begin by extending all possible substrings from location $i$ that appear with probability of at least $\epsilon$. Each time we check if some string that we have extended so far can be extended even more to the right. Once we cannot extend a string, it is outputted (of course, using delimiters between consecutive outputs of substrings).

Noting that finding $LST(T)$ from $T$ can be done in linear time, it is easy to see that the running time of this algorithm is linear in the size of the output, i.e. linear in the total length of all extended maximal factors. By combining this result with Theorem 1, the corollary follows.

**Corollary 2.** *Given a constant threshold $0 < \epsilon \leq 1$ and a weighted text $T$, the total length of all extended maximal factors of $T$ is linear in the length of $T$, and can be found in linear time.*

In the following section we show how to solve weighted matching problems by reducing weighted matching problems to property matching problems.

## 8. Solving weighted matching problems

Weighted matching problems are regular pattern matching problems where the text is weighted, and we say that a pattern appears in the text if the probability of appearance of the pattern is above some threshold probability $\epsilon$. We now show how to reduce this problem to the Property Matching Problem.

Given a weighted string $T$, we find the string of the extended maximal factors of $T$ as was described in Section 7. Denote this string by $\hat{T}$. $\hat{T}$ is a regular string, but each location has an associated probability that comes from the original location of that letter in $T$ (the delimiters are said to have probability 0). Thus, we can define a property as the set of all intervals $(s_k, f_k)$ where the product of the probabilities from location $s_k$ to location $f_k$ is at least $\epsilon$, and the product of the probabilities from location $s_k - 1$ to location $f_k$ and from location $s_k$ to location $f_k + 1$ is less than $\epsilon$. Clearly, if a pattern matches $\hat{T}$ at some location under the defined property, then the pattern weight matches $T$ at some location. Note that this location can be found simply by saving for each location in $\hat{T}$ the original location in $T$ that it came from (that will be the match location).

This reduction immediately gives us the following.

**Corollary 3.** *Weighted matching problems can be solved in the same running times as property matching except for an $O((\frac{1}{\epsilon})^2 \log\frac{1}{\epsilon})$ degradation, where $\epsilon$ is the threshold probability.*

Finally, we can also solve the indexing problem for weighted strings using the reduction above in $O(n(\frac{1}{\epsilon})^2 \log\frac{1}{\epsilon} \log|\Sigma| + n(\frac{1}{\epsilon})^2 \log\frac{1}{\epsilon}(\log\log\frac{1}{\epsilon} + \log\log n))$ preprocessing time, and $O(|P|\log|\Sigma| + tocc_\pi)$ query time, where $tocc_\pi$ is the number of occurrences of $P$ in $T$ with probability at least $\epsilon$.

## 9. Concluding remarks

We remark that our framework for solving weighted matching problems yields solutions to hitherto unsolved problems in weighted matching, such as scaled matching, swapped matching, parameterized matching and indexing, as well as efficient solutions to others such as exact matching and approximate matching.

Furthermore, we note that in practice, when dealing with weighted matching problems, $\epsilon$ is usually considered as a constant. Thus, solving problems such as exact matching, scaled matching, swapped matching, parameterized matching, approximate matching and many more on weighted sequences can be done, using our framework, in the same running times as those of the best known algorithms for the non-weighted versions, while weighted indexing can be done in $O(n(\log|\Sigma| + \log\log n))$ preprocessing time and $O(|P|\log|\Sigma| + tocc_\pi)$ query time for a text of length $n$, where $tocc_\pi$ is the number of occurrences of pattern $P$ in $T$ with probability of at least $\epsilon$.

## Acknowledgements

The first author was partly supported by NSF grant CCR-01-04494 and ISF grant 35/05.

## References

[1] A. Amir, Y. Aumann, R. Cole, M. Lewenstein, E. Porat, Function matching: Algorithms, applications and a lower bound, in: Proc. of the 30th International Colloquium on Automata, Languages and Programming, ICALP, 2003, pp. 929–942.
[2] A. Amir, Y. Aumann, G. Landau, M. Lewenstein, N. Lewenstein, Pattern matching with swaps, Journal of Algorithms 37 (2000) 247–266. Preliminary version appeared at FOCS 97.
[3] A. Amir, A. Butman, M. Lewenstein, Real scaled matching, Information Processing Letters 70 (4) (1999) 185–190.
[4] A. Amir, A. Butman, M. Lewenstein, E. Porat, Real two dimensional scaled matching, in: Proc. 8th Workshop on Algorithms and Data Structures, WADS, 2003, pp. 353–364.
[5] A. Amir, A. Butman, M. Lewenstein, E. Porat, D. Tsur, Efficient one dimensional real scaled matching, Journal of Discrete Algorithms 5 (2) (2007) 205–211.
[6] A. Amir, G. Calinescu, Alphabet independent and dictionary scaled matching, Journal of Algorithms 36 (2000) 34–62.
[7] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, E. Porat, Overlap matching, Information and Computation 181 (1) (2003) 57–74.
[8] A. Amir, E. Eisenberg, E. Porat, Swap and mismatch edit distance, in: Proc. 12th Annual European Symposium on Algorithms, ESA, 2004, pp. 16–27.
[9] A. Amir, D. Keselman, G.M. Landau, M. Lewenstein, N. Lewenstein, M. Rodeh, Text indexing and dictionary matching with one error, Journal of Algorithms 37 (2) (2000) 309–325.
[10] A. Amir, G.M. Landau, M. Lewenstein, N. Lewenstein, Efficient special cases of pattern matching with swaps, Information Processing Letters 68 (3) (1998) 125–132.
[11] A. Amir, G.M. Landau, U. Vishkin, Efficient pattern matching with scaling, Journal of Algorithms 13 (1) (1992) 2–32.
[12] A. Amir, M. Lewenstein, E. Porat, Approximate swapped matching, Information Processing Letters 83 (1) (2002) 33–39.
[13] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (1977) 762–772.
[14] M. Christodoulakis, C.S. Iliopoulos, L. Mouchard, K. Tsichlas, Pattern matchnig on weighted sequences, in: Proceedings of the Algorithms and Computational Methods for Biochemical and Evolutionary Networks, CompBioNets, KCL Publications, 2004.
[15] R. Cole, L. Gottlieb, M. Lewenstein, Dictionary matching and indexing with errors and don't cares, in: Proc. 36th annual ACM Symposium on the Theory of Computing (STOC), ACM Press, 2004, pp. 91–100.
[16] R. Cole, R. Harihan, Randomized swap matching in $o(m\log m\log|\sigma|)$ time, Technical Report TR1999-789, New York University, Courant Institute, September 1999.
[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, second edition, MIT Press, 2001.
[18] M. Farach, S. Muthukrishnan, Perfect hashing for strings: Formalization and algorithms, in: Proc. 7th Combinatorial Pattern Matching Conference, 1996, pp. 130–140.
[19] P. Ferragina, R. Grossi, Fast incremental text editing, in: Proc. 7th ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 531–540.
[20] R. Gesine, S. Sophie, M.S. Waterman, Probabilistic and statistical properties of words: An overview, Journal of computationnal Biology 7 (1–2) (2000) 1–46.
[21] M. Gu, M. Farach, R. Beigel, An efficient algorithm for dynamic text indexing, in: Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, 1994, pp. 697–704.
[22] C.S. Iliopoulos, L. Mouchard, K. Perdikuri, A. Tsakalidis, Computing the repetitions in a weighted sequence, in: Proceeding of the Prague Stringology Conference, 2003, pp. 91–98.
[23] C.S. Iliopoulos, K. Perdikuri, E. Theodoridis, A. Tsakalidis, K. Tsichlas, Motif extraction from weighted sequences, in: A. Apostolico, M. Melucci (Eds.), Proc. 11th Symposium on String Processing and Information Retrieval, SPIRE, in: LNCS, vol. 3246, Springer, 2004, pp. 286–297.
[24] J. Jurka, Origin and evolution of Alu repetitive elements, in: The Impact of Short Interspersed Elements (SINEs) on the Host Genome, 1995, pp. 25–41.

[25] J. Jurka, Human repetitive elements, in: Molecular Biology and Biotechnology, 1995, pp. 438–441.
[26] Juha Kärkkäinen, Peter Sanders, Simple linear work suffix array construction, in: Proc. 30th International Colloquium on Automata, Languages and Programming, ICALP 03, in: LNCS, vol. 2719, 2003, pp. 943–955.
[27] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM Journal on Computing 6 (1977) 323–350.
[28] R. Lowrance, R.A. Wagner, An extension of the string-to-string correction problem, Journal of the ACM (1975) 177–183.
[29] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal of the ACM 23 (1976) 262–272.
[30] S. Muthukrishnan, New results and open problems related to non-standard stringology, in: Proc. 6th Combinatorial Pattern Matching Conference, in: Lecture Notes in Computer Science, vol. 937, Springer-Verlag, 1995, pp. 298–317.
[31] S. Muthukrishnan, Efficient algorithms for document retrieval problems, in: Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 657–666.
[32] S.C. Sahinalp, U. Vishkin, Efficient approximate and dynamic matching of patterns using a labeling paradigm, in: Proc. 37th FOCS, 1996, pp. 320–328.
[33] J.D. Thompson, D.G. Higgins, T.J. Gibson, Clustal w: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice, Nucleic Acids Research 22 (1994) 4673–4680.
[34] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[35] J.C. Venter, Celera Genomics Corporation, The sequence of the human genome, Science 291 (2001) 1304–1351.
[36] R.A. Wagner, On the complexity of the extended string-to-string correction problem, in: Proc. 7th ACM STOC, 1975, pp. 218–223.
[37] P. Weiner, Linear pattern matching algorithm, in: Proc. 14 IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.