Property Matching and Weighted Matching

Amihood Amir[1], Eran Chencinski[2], Costas Iliopoulos[3], Tsvi Kopelowitz[2], and Hui Zhang[3]

[1] Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel and College of Computing, Georgia Tech, Atlanta, GA 30332-0280
+972 3 531-8770
amir@cs.biu.ac.il

[2] Dept. of Computer Science, Bar-Ilan U., 52900 Ramat-Gan, Israel
(972-3)531-8408
{chenche, kopelot}@cs.biu.ac.il

[3] Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom
{csi, hui}@dcs.kcl.ac.uk

Abstract. 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.

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, Σ is the alphabet, and $tocc_\pi$ is the number of occurrences of the pattern that satisfy some property π.

As a practical use of Property Matching we show how to solve Weighted Matching problems using techniques from Property Matching. Weighted sequences have 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 ϵ in weighted text T of length n, we reduce the problem to a property matching problem of pattern P in text T' of length $O(n(\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.

1 Introduction

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.

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 is that of finding all occurrences of *pattern* $P = p_1p_2 \cdots p_m$ in *text* $T = t_1t_2 \cdots t_n$, where T and P are strings over alphabet Σ. 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 and information retrieval. However, when it comes to indexing a text with properties, intersecting the pattern with the properties may give a worst case that is not output-dependent.

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 π. These are almost the same bounds that exist in the literature for ordinary indexing.

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) [5]. The *weighted sequence* of length m is a $|\Sigma| \times m$ matrix that reports the frequency of each symbol in finite alphabet Σ (nucleotide, in the genomic setting) for every possible location.

Iliopoulos et al. [4] 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. 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 [4] even for simple tasks such as exact matching. It is certainly desirable to be able to answer more ambitious questions, such as *scaled weighted matching*, *swapped weighted matching*, *parameterized weighted matching* as well as to *index* a weighted sequence.

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 ϵ. 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.

2 Property Matching – Definitions

For a string $T = t_1 \cdots t_n$, we denote by $T_{i \cdots j}$ the substring $t_i \cdots t_j$. The suffix $T_{i \cdots n}$ 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 π 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 π, denoted by $|\pi|$, is the number of intervals in the property (or in other words - t).*

We assume that the properties are given in standard form as defined below.

Definition 2. *A property π 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

This section defines the notion of general pattern matching with properties. following definition.

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

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

Definition 4. *For a property π 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 π 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 α-matching problem that runs in time $O(g_\alpha(n,m))$, then given a text T with property π, and pattern P, we can find all $T_{i \cdots j}$ that α-match P in time $O(g_\alpha(n,m)+n) = O(g_\alpha(n,m))$.

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 π, preprocess T such that on-line queries of the form "find all locations where a pattern string P occurs in T under π" 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 π; it includes also the occurrences of P in T that are not occurrences under π. 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 π, 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 π. The idea is based on a lemma that we provide following the next definition.

Definition 6. *For a string T with property π and a node u in the suffix tree of T, we denote by S_u^π the maximal set of locations $\{i_1, \cdots, i_\ell\} \subseteq \{1, \cdots, n\}$ 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 π, 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^π, it must be that $i_j \in S_v^\pi$. □

Corollary 1. *For a string T with property π, 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 two paths from Corollary 1, and the i^{th} suffix of T. Let v be the deepest node on the first path. The location of i in the PST of T is defined as follows. If $end(i) - i = |label(v)| - 1$ 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, [6]. 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 [3], and improved upon in [1]. 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 [1] 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 Subsection 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 Subsection 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 [2]). 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 left most 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 $O(\sum_{i=0}^{\log n'} \frac{n'}{2^i}) = O(n')$.

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

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 less 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 ℓ characters of $label(e)$, then we first output $suf(w)$ and $suf(e')$ 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 Subsection 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 π. 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.*

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 Σ is a sequence of sets t_i, $i = 1, \cdots, n$. 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,*

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

Definition 10. *Given a pattern $P = p_1 \cdots p_m$ over alphabet Σ, 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 ϵ if $\prod_{j=1}^{m} \pi_{i+j-1}(p_j) \geq \epsilon$, where ϵ is a given parameter which we call the threshold probability.*

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

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 ϵ. 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 ϵ. 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 ϵ.

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 ϵ, since a maximal factor has probability of appearance at least ϵ 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 at least $\Omega(n^2)$, 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:*

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

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 ϕ) and treat each part of $LST(T)$ divided by ϕ separately. For the rest of this paper, we assume $LST(T)$ has no ϕ'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.*

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.*

Definition 16. *Given $0 < \epsilon \leq 1$ and a weighted text T, we say that location i is a starting location of T, if either $i = 1$ or $i > 1$ and t_{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 5. *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.*

Lemma 6. *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 $O((\frac{1}{\epsilon})^2 \log \frac{1}{\epsilon})$.*

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 6. □

The following lemma shows that this analysis is tight up to a logarithmic factor.

Lemma 7. *Given $0 < \epsilon \leq 1$ and a weighted text T, the total length of all extended maximal factors of T is $\Omega(n(\frac{1}{\epsilon})^2)$.*

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 $\{s_i^1, s_i^2, \cdots, s_i^{k_i}\}$ is the set of characters appearing at location i with positive probability, and $\{\alpha_i^1, \alpha_i^2, \cdots, \alpha_i^{k_i}\}$ 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 ϵ, 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 ϵ. 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 an we say that a pattern appears in the text if the probability of appearance of the pattern is above some threshold probability ϵ. 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 ϵ, 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 ϵ. 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 ϵ 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 ϵ.

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, ϵ 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 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 text of length n, where $tocc_\pi$ is the number of occurrences of pattern P in T with probability of at least ϵ.

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

References

1. A. Amir, D. Keselman, G. Landau, M. Lewenstein, N. Lewenstein, and M. Rodeh. Indexing and dictionary matching with one error. *Journal of Algorithms*, 37:309–325, 2000. (Preliminary version appeared in WADS 99.).
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.
3. M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *Proc. 7th Combinatorial Pattern Matching Conference (CPM)*, pages 130–140, 1996.
4. C. S. Iliopoulos, L. Mouchard, K. Perdikuri, and A. Tsakalidis. Computing the repetitions in a weighted sequence. In *Proceeding of the Prague Stringology Conference*, pages 91–98, 2003.
5. J.D. Thompson, D.G. Higgins, and 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*, 22:4673–4680, 1994.
6. P. Weiner. Linear pattern matching algorithm. *Proc. 14 IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.