Approximate Matching in Weighted Sequences

Amihood Amir[1], Costas Iliopoulos[2], Oren Kapah[3], and Ely Porat[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] Department of Computer Science, King's College London, Strand, London WC2R 2LS, United Kingdom
(+44) 20 7848 2809
csi@dcs.kcl.ac.uk

[3] Department of Computer Science, Bar-Ilan University, 52900 Ramat-Gan, Israel
(972-3)531-7620
{kapaho, porately}@cs.biu.ac.il

Abstract. Weighted sequences have been recently 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 the probability of every symbol's occurrence at every text location is given.

We address the problem of approximately matching a pattern in such a weighted sequence. The pattern is a given string and we seek all locations in the set where the pattern occurs with a high enough probability. We define the notion of Hamming distance and edit distance in weighted sequences and give efficient algorithms for computing them. We compute two versions of the Hamming distance in time $O(n\sqrt{m}\log m)$, where n is the length of the weighted text and m is the pattern length. The edit distance is computed in time $O(nm)$ and $O(nm^2)$, depending on the edit distance definition used. Unfortunately, due to space considerations, the edit distance details are left to the journal version.

We also define the notion of weighted matching in infinite alphabets and show that exact weighted matching can be computed in time $O(s\log^2 s)$, where s is the number of text symbols having non-zero probability. The weighted Hamming distance over infinite alphabets can be computed in time $\min(O(kn\sqrt{s}+s^{3/2}\log^2 s), O(s^{4/3}m^{1/3}\log s))$.

1 Introduction

Recently, a new pattern matching paradigm was introduced. *Weighted sequences* was initially motivated by trying to reconcile the differences between the genomic sequences of different individuals. The great effort of the *genome project*, that is now winding down, has been to construct a "consensus sequence" of the human genome. Individual human genomes are very similar therefore such a "generic" consensus sequence can be achieved. Nevertheless, clearly no two individuals have the same DNA sequence. Several methods have been proposed for dealing with this polymorphism. One proposed idea is that of the *Position Weight Matrix* (PWM for short) [14]. The PWM is a more precise encoding that takes into account the relative frequency of each nucleotide. The *weighted sequence* of length m (the PWM of a set of strings 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.

Originally, PWM sequences were used for relatively short sequences, e.g. binding sites or sequences resulting of multiple alignments. Iliopoulos et al. [11, 9, 3, 10] 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 [15]. By keeping all the information the whole-genome shotgun produces, it should be possible to ferret out information that has been 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.

In this paper we develop algorithms for approximate search on weighted sequence. We handle the case of *Hamming distance* (*edit distance* is deleted from this paper due to page limits). In approximate matching it is assumed that there are errors in the data and matches with a small number of errors are sought. In classical pattern matching it usually does not matter if the assumption is that the text is error-free and the errors are all in the pattern or vice-versa, since there is a symmetry in most edit operations. It turns out that in weighted matching (as in, e.g., hypertext matching [2]) there is a distinction between cases. Assuming mismatch errors in the weighted text, there is always an approximate match at every location, depending on the number of errors. However, assuming a "clean" text and mismatch errors in the pattern, there may be locations where a match can not ever be found, no matter what the cost. We show efficient algorithms for computing the Hamming distance at every location for both models.

The contributions of this paper are as follows.

1. We formalize two possible definitions for *Hamming distance* in weighted sequences. Since we are dealing with a new paradigm, this formalism is very important. Special care should be given for a definition that captures natural traits that will be useful in applications.
2. We provide the first efficient algorithms for computing the Hamming distance on weighted sequences. Our algorithms run in time $O(n\sqrt{n \log m})$, where the length of the weighted text is $|\Sigma|n$ and the length of the pattern is m. This algorithm is achieved via a non-trivial bounded divide-and-conquer algorithm, coupled with some insights we prove on weighted sequences.
3. We formalize two possible definitions for *edit distance* in weighted sequences, and provide dynamic programming algorithms for the edit distance. One definition leads to a $O(nm)$, algorithm and the other has a $O(nm^2)$ algorithm. This treatment is left for the journal version due to page limitations.
4. We define weighted matching over infinite alphabets and provide the first efficient algorithm to solve the problem. Our algorithm runs in time $O(s \log^2 s)$ and uses superimposed coding techniques.

5. We provide the first known efficient algorithms for computing Hamming distance of a pattern in a weighted text over infinite alphabet. Our algorithm runs in time $\min(O(kn\sqrt{s} + s^{3/2}\log^2 s), O(s^{4/3}m^{1/3}\log s))$.

2 Preliminaries

Definition 1. *A* weighted sequence $T = t_0, ..., t_n$ *over alphabet* Σ *is a sequence of sets* t_i, $i = 0, ..., n$. *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 *in location* i. *Formally,*

$$t_i = \{(s_i, \pi_i(s_j)) \mid s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}.$$

For a finite alphabet $\Sigma = \{a_1, ..., a_{|\Sigma|}\}$ we can view a weighted sequence as a $|\Sigma| \times n$ matrix T of numbers in $[0,1]$, where $T[j,i] = \pi_i(a_j)$. For the rest of this paper we assume a finite fixed alphabet Σ.

Definition 2. $P = p_0, ..., p_m$ *is a* solid sequence *over alphabet* Σ *if* $p_i \in \Sigma$, $i = 0, ..., m$.

We say that solid pattern P *(or simply* pattern P) occurs in location i of weighted text T with probability at least α *if* $\prod_{j=0}^{m} \pi_j(p_j) \geq \alpha$.

Definition 3. *The* exact weighted matching problem *is defined as follows:*

INPUT: Weighted text T *over alphabet* Σ, *solid pattern* P *over alphabet* Σ, *and probability* $\alpha \in [0,1]$.
OUTPUT: All locations i *in* T *where pattern* P *occurs with probability at least* α.

Using convolutions, as introduced by Fischer and Paterson [8], as well as the observation that Solid pattern P occurs in location i of weighted text T with probability at least α if $\sum_{j=0}^{m} \log \pi_j(p_j) \geq \log \alpha$. we can efficiently solve the exact weighted matching problem in time $O(|\Sigma| n \log m) = O(n \log m)$. The idea is to use the Fast-Fourier-Transform (FFT) [6] to compute the sum of the log probabilities for every pattern symbol separately. This can be done in time $O(n \log m)$, in a computational model with word size $O(\log m)$.

We are now ready for the Hamming distance in weighted sequences problem.

3 Hamming Distance – Error in Text

Computing the Hamming distance between two (solid) strings assumes that a number of symbols were replaced. The Hamming distance is the number of these replaced symbols. In the case of weighted subsequences it makes a difference where these symbols were replaced. The simpler case, which we consider in this section, assumes replacement in the text. The assumption is that some text symbols are erroneous and, in fact, there should have been a probability 1 for the symbol that happens to match the pattern, rather than the probabilities that appear in the text.

Note that by this definition, allowing enough mismatches can guarantee a match at every location, no matter how close to 1 we choose α.

Definition 4. *The* Weighted Hamming Distance with Mismatches in the Text problem *is the following:*

INPUT: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ, and probability $\alpha \in [0, 1]$.
OUTPUT: For every location i in T, the minimum k such that if k text probabilities were changed to 1 then pattern P would occur at location i with probability at least α.

There does not seem to be a natural way to use the powerful constraint that the numbers in the weighted text are probabilities. However, it seems like we can solve the problem without it. We reduce the weighted Hamming distance with mismatches in the text problem to the *minimum ignored mask bits problem.* The idea is to consider a text whose elements are non-positive numbers, and a pattern which is a mask, i.e. its symbols are 0's and 1's. Suppose we are interested in finding out, for each text location i, the sum of the text numbers that are aligned with 1's in the pattern.

Clearly this is a simple convolution of the pattern and text. However, we add a complication, we also have a non-positive integer α and for every text location i we seek the smallest number of mask bits that, if set to 0, would make the sum of text numbers that are aligned with (the remaining) 1's in the pattern, be no less than α.

We formally define the problem.

Definition 5. *The* Minimum Ignored Mask Bits problem *is the following:*
INPUT: Solid text T of length $n + 1$ whose elements are non-positive integers, solid pattern P of length $m + 1$ over alphabet $\{0, 1\}$, and integer $\alpha \leq 0$.
OUTPUT: For every location i in T, the minimum k such that if k pattern bits are changed from 1 to 0, and M' is the pattern resulting from those k changes, then $\sum_{j=0}^{m} T[i + j]M[j] \geq \alpha$.

Claim. The weighted Hamming distance with mismatches in the text problem is linearly reducible to the minimum ignored mask bits problem.

Proof: Given weighted text T in matrix format, where the value in $T[i, j]$ is $\log \pi_j(s_i)$, let solid text T' be a linear listing of matrix T in column-major order, i.e. $T' = T[1, 0], T[2, 0], T[3, 0], ..., T[|\Sigma|, 0]$,
$T[1, 1], T[2, 1], T[3, 1], ..., T[|\Sigma|, 1], ...,$
$T[1, n], T[2, n], T[3, n], ..., T[|\Sigma|, n]$. Let M be a string of length $|\Sigma|(m + 1)$ over $\{0, 1\}$ where M is the concatenation of strings $B(p_0), B(p_1), ...B(p_m)$. $B(a)$ is defines as follows. Let $a = s_\ell$, where $\Sigma = \{s_1, s_2, ..., s_{|\Sigma|}\}$. Then $B(a)$ is a bit string of length Σ, where the ℓ-th element is 1 and all other elements are 0.

Example: If $\Sigma = \{A, B, C, D\}$ and $P = BBAD$, then $M = 0100\,0100\,1000\,0001$.

Clearly, the reduction is linear. It is also clear that turning a 1 bit in the mask M to 0, is equivalent to changing the probability in the text position corresponding to it to 1. Thus a solution to the minimum ignored mask bits problem will provide the solution to the weighted Hamming distance with mismatches in the text problem. □

Algorithm's Idea
We consider two limited cases and show an easy efficient solution for each of them. Subsequently, we use a bounded divide-and-conquer strategy, that splits a general input into the two straightforward cases, and thus solves each separately.

The first special case is one where the domain of numbers appearing in the text is bounded, i.e. there are only r different numbers that can appear as text elements.

Since we are interested in finding the smallest number k of mask 1 bits that, when turned to 0 will make the sum greater than α, and since **all numbers are non-positive**, the following observation is crucial to the algorithm:

Observation 1. *For any location i where $\sum_{j=1}^{r} S_{i,j} < \alpha$, the solution to the minimum ignored mask bits problem can be found by sequentially adding numbers that participate in the sum starting from the ones that contribute least to decreasing it, i.e. the largest (n_1). Stop adding them when the remaining sum is no longer less than α.*

This elimination would normally require $O(m)$ work per location. However, since there are only r different values, and we know how many instances of each value participate in the sum at location i ($S_{i,j}/n_j$), we can do this in time $O(r)$ per location.

Algorithm's Time: $O(rn \log m)$
A second special case we consider is when there is no bound on the number of different text elements, but we do know that for every text substring of length m there are at most r elements greater than α. This means that for location i, there is no point in even considering all elements except those r.

Algorithm's Time: $O(nr)$
We are now ready to present our divide-and-conquer algorithm. Assume first, that the text length is at most $2m$. This is a standard assumption and can be made without loss of generality (see e.g. [1]). We now sort all text elements and split them into r blocks of size at most $2|\Sigma|\frac{m}{r}$ each.

The idea is to use *Algorithm Bounded Alphabet* on the blocks, and *Algorithm Bounded Relevant Numbers* to find the border of the numbers participating in the sum within the block that tips under α. This can be done with a twist on Abrahamson's idea and produce the final algorithm.

Algorithm's Time: The time for this algorithm is $O(rf(m)) + O(m\frac{m}{r})$, where $f(m)$ is the time it takes to compute the block information. We do it by convolutions, as in *Algorithm Bounded Alphabet* so $f(m) = m \log m$. The optimal r is then the one where $r = \sqrt{\frac{m}{\log m}}$.. Thus the algorithm's time is $O(n\sqrt{m \log m})$.

4 Hamming Distance – Error in Pattern

The situation currently addressed is one where the weighted text is assumed to be error-free. The pattern, however, may have replacemet errors, i.e. it is possible that the “true” pattern symbol was replaced by another. This situation is different in a number of ways from the one considered in section 3.

The first difference between the two Hamming distance definitions is the following. The *errors in the text* definition can guarantee a match at every location, no matter how close to 1 we choose α. This is done simply by allowing enough mismatches. At the worst case $m+1$ mismatches give a probability of 1. This is not the case if we assume errors in the pattern.

We formally define our problem.

Definition 6. *The* Weighted Hamming Distance with Mismatches in the Pattern problem *is the following:*

INPUT: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ, and probability $\alpha \in [0,1]$.

OUTPUT: For every location i in T, the minimum k such that if k pattern symbols were replaced to create new pattern P' then pattern P' would occur at location i with probability at least α.

The difficulty presented by this definition is that we put the weight of change on the pattern, rather than the text. When a text is changed, by definition 4, that change improves the product of every match that this text location participates in. However, a pattern change may improve the probability in one occurrence but actually make it worse in another.

One may be tempted to say that even when a match is defined on the pattern, we can still tell which probability is *always* best for a given text location - the maximum probability at that location. This maximum probability will actually improve (or at least will never hurt) the probability of any location that it participates in. Perhaps, then, it is possible to sort the text by largest to smallest product improvement. Then it may be possible that the algorithm in section 4 could still be modified to find the largest possible sum of log probabilities and check if it is good enough. The idea would be the following. First make a replacement in the text location that introduces the largest unmatched text probability. Use that replacement only if it is necessary. Proceed by introducing the next largest, etc. The problem is that replacing elements in sorted order from largest to smallest does not guarantee the smallest number of replacements.

The following lemma does guarantee an order of replacement. Assume that the weighted text is given in matrix format T where $T[j,i] = \pi_i(a_j)$, where $\Sigma = \{a_1, ..., a_{|\Sigma|}$, and $\pi_i(a_j)$ is the probability of having symbol a_j at text location i. Let $\max(T[*,i])$ denote the value $\max\{T[j,i] \mid j = 1, ..., |\Sigma|\}$.

Lemma 1. *Consider text element $T[j,i]$ where $\frac{\max(T[*,i])}{T[j,i]}$ is the largest. Let P be a pattern where $a_j \in P$. Then the largest increase in the product of probabilities as a result of a single symbol replacement occurs by replacing every a_j in the pattern that matches text location i by a_ℓ, where $T[\ell,i] \geq T[j,i],\ j = 1, ..., |\Sigma|$.*

Proof: Let q be the product of probabilities at location i. Assume that the pattern has a_j at location $i+\ell$, $\ell \leq m$, but that symbol a_h has the largest text probability at location $i+\ell$. Then replacing a_j by a_h would cause the product of the probabilities at location i to be $(q/T[j,i+\ell])T[h,i+\ell]$. This means that the largest change will occur when $\frac{T[j,i+\ell]}{T[h,i+\ell]}$ is largest. □

Conclude: Let $T[j,i]$ be such that $\frac{\max(T[*,i])}{T[j,i]}$ is the largest. If we replace **text location** $[j,i]$ by the value $\max(T[*,i])$, the result will be equivalent to replacing every a_j in the pattern that matches text location i by a_ℓ, where $T[\ell,i] \geq T[j,i],\ j = 1, ..., |\Sigma|$. This leads to the idea that if we replace text elements by descending order of $\frac{\max(T[*,i])}{T[j,i]}$ (where necessary) we will guarantee the minimum number of replacements at every location.

Algorithm's Idea: Sort all $2m|\Sigma|$ text elements in non-increasing order of the ratio $\frac{\max(T[*,i])}{T[j,i]}$. As in section 3, split the text elements into $O(\frac{m}{\sqrt{m\log m}})$ groups of size $O(\sqrt{m\log m})$. For each text location i calculate the probabilities $O(\frac{m}{\sqrt{m\log m}})$ times. In the first time calculate the probability of the pattern in the text without replacements. In the second time calculate the probability of the pattern with replacing every element in the group of highest ratios. In the jth time, calculate the pattern probability with replacing every element in the $j-1$ highest ratio groups.

Each such calculation can be done by FFT in time $O(n\log m)$. In addition, we can calculate by FFT the number of replacements done in each location for the groups involved. Finally, in a manner similar to the one shown in section 3, we can fine tune the exact number of replacements for each text location i in time $O(n\sqrt{m\log m})$.

A detailed description of the algorithm will appear in the journal version.

5 Weighted Matching over Infinite Alphabets

The original motivation of weighted sequence matching was from computational biology, where the alphabets are quite small (size 4 for DNA and RNA, and size 20 for amino acids). Nevertheless, from a conceptual point of view, nothing prohibits the alphabet from being very large, or even infinite. The techniques for weighted matching need to be completely different over infinite alphabets, since we may no longer assume that all symbols appear as inputs. Rather, we only input the symbols whose probability is non-zero.

Our formal definition of weighted matching (Definition 1) did not assume a finite alphabet. We now provide an efficient algorithm for exact weighted matching over infinite alphabets. The key observation for our efficient algorithm utilizes *subset matching.* Subset matching was defined by Cole and Hariharan [4], as a tool to solve the *tree pattern matching problem* [12, 7] but meanwhile has proven to be an interesting problem in and of itself. The input of the problem is a text array of n sets totaling s elements and a pattern array of m sets totaling s' elements. There is a match of the pattern in a text location if every pattern set is a subset of the corresponding text set. Formally,

Definition 7. *The Subset Matching Problem is defined as follows.*
INPUT: Text $T = T_1, T_2, ..., T_n$ *of sets* $T_i \subseteq \Sigma, \quad i = 1, ..., n$ *and pattern* $P = P_1, P_2, ..., P_m$ *of sets* $P_i \subseteq \Sigma, \quad i = 1, ..., m$, *where* Σ *is a given alphabet.*
OUTPUT: All locations $i, \quad 1 \leq i \leq n-m+1$ *where* $\forall \ell = 1, ..., m, \quad P_\ell \subseteq T_{i+\ell-1}$.

Algorithm's Idea: Observe that in every text location where the pattern appears with non-zero probability, there is a subset-matching of the pattern. The algoritm's main idea is, then, to first find the subset matching of the pattern in the text and then calculate the probabilities of those locations. In order to accomplish that all the non zero probabilities will be mapped to a vector with size linear in the number of non-zeros. This mapping will be done using shifting where each symbol is assigned a different shift. The same shifting will be used in both the text and the pattern, thus wherever there is a *singleton* in the text which aligned with a *singleton* in the pattern in the positions where a subset matching was found it is guaranteed to be be the same character.

```
Algorithm Outline
1. Perform subset matching
2. Linearize the input to a vector of probabilities, and calculate the probability of
the pattern
   appearing in each text location
end Algorithm Outline
```

Step 1: Ignore the probabilities and consider only the symbols that have a non zero probability. This results in a set of symbols for each text location. Now run Cole and Hariharan's subset matching algorithm [5].

Time Complexity: $O(s \log s)$ where s is the total number of characters.

Step 2: Create a vector of probabilities from the text. This is done by assigning for each alphabet symbol σ a number that sets the shift of this letter. This means that for each location i where σ has a non zero probability, this probability will appear at the new vector at location $i + shift(\sigma)$. Each vector location where more than one value is assigned is referred to as a **multiple** location and is assigned a 0. Every position where only one value was assigned is referred to as a **singleton** and is assigned the log-probability the symbol assigned to it.

Using the same shift values we create a vector of the same size from the pattern. In the vector representing the pattern each symbol that appears as a

singleton in the pattern is replaced by a 1. Multiples are replaced by a 0. After a convolution of the text vector with the pattern vector each location holds the sum of all the probabilities where a **singleton** in the text was aligned with a **singleton** in the pattern.

Lemma 2. *For the locations where a subset matching is found, each* **singleton** *location in the text vector which is aligned with a* **singleton** *in the pattern vector, contains the probability of the letter which appeared as* **singleton** *in the pattern vector.*

Proof: In a situation where a subset matching occurs, clearly the pattern symbol is shifted to the same location as its equivalent text symbol. In a singleton text matched with a singleton pattern, if there the subset match forces the fact that the text symbol equals the pattern symbol and we do not need to verify it. For the same reason, it is impossible for a pattern multiple to be matched with a text singleton when there is a subset match, since all pattern elements should be matched at least with the appropriate text symbols. □

Corollary 1. *After convolving the text vector with the pattern vector, all the non zero probabilities which appeared as* **singletons** *are calculated for all locations.*

As a result of the corollary we can now zero the calculated probabilities of the text singletons and repeat the process using different shifts until all the non zero probabilities become zero. Our solution is the sum of the results. At the end of this process each location where there is a subset match holds the log of the probability of the pattern appearing at this location.

The question we still need to address is how many such rounds are necessary until all the non zeros probabilities appears as **singletons** at least once.

Lemma 3. $O(\log s)$ *rounds are guaranteed to complete the process in the worst case. The appropriate shift functions can be computed in time* $O(s \log^2 s)$*, where* s *is the number of text symbols with non-zero probabilities.*

Proof: The lemma can be proven using superimposed coding as in Cole and Hariharan [5]. A complete proof will be provided in the journal version. □

Time Complexity: The shift functions can be computed in time $O(s \log^2 s)$. For each shift function we perform a convolution which takes $O(s \log s)$ time. There are $O(\log s)$ such convolutions, thus the total time of this step is $O(s \log^2 s)$.

6 Hamming Distance in Weighted Matching over Infinite Alphabets

We present an algorithm for the case of *errors in the text.* The case of *errors in the pattern* is similar.

The main idea of the algorithm is to combine the algorithm devised for the Hamming distance over finite alphabet with the algorithm devised for the weighted matching over infinite alphabet. The difficulty in applying the shifting technique in the Hamming distance case is that now the property that two aligned singletons must originat from the same character no longer applies. We show two ways of overcoming this problem: one with running time dependent on the number of errors and one with running time independent of the number of errors. Both algorithms use the bounded divide-and-conquer technique. We divide the sorted list of probabilities into blocks. For each text location we calculate the sum of probabilities and the number of matches for each block. Subsequently we add the log probabilities from the largest down until the probability is smaller than the input. The number of errors (the Hamming distance) is the size of the pattern minus the number of matches.

We combine the two algorithms to achieve the minimal running time of the two solutions.

6.1 Algorithm 1

The first algorithm solves the problem in the shifting technique by checking for each block separately if there was a match. This is done for each block by replacing each empty position in the shifted text with don't care and matching the shifted pattern with the new text. If a match exist then the sum of probabilities for this block is correct. If there is no match then we need to use brute force. This means checking each character in this block against the character appearing in the aligned position in the pattern.

Algorithm Outline
1. Sort the probabilities in the weighted sequence
2. Divide the sorted list of probabilities into blocks of size $\sqrt{s}$
3. For each block calculate the sum of probabilities (using the shifting technique).
4. For each text position and each block: If there is a subset match use the result calculated in the previous stage, else use brute force.
5. For each text position add blocks probabilities from the largest down until the sum goes below the input threshold.
6. For each text position add probabilities within the last block until the sum goes below the input threshold.

end Algorithm Outline

Time: $O(kn\sqrt{s} + s^{3/2}\log^2 s)$

Where k is the average number of blocks per text position, where a match does not exist.

6.2 Algorithm 2

The second algorithm handles the problem in the shifting technique by dividing the symbols into *frequent* and *non-frequent* characters and dealing with each type of characters separately.

Definition 8. *Let P be a pattern of length m. A pattern symbols is* frequent *if it occurs in the pattern at least $m^{2/3}$ times, otherwise it is* rare.

For each block and each *frequent* character, the sum of probabilities is calculated using convolution. Also the number of matches is calculated using convolution where we replace the probabilities in the text with ones. Since there are at most $m^{1/3}$ *frequent* characters and $s^{1/3}$ blocks, the time complexity for the *frequent* characters is $O(s^{4/3}m^{1/3}\log s)$.
For the *non-frequent* characters, brute force is used to calculate the sum of probabilities and the number of matches. Since each *non-frequent* character can appear up to $m^{2/3}$ times in the pattern the time complexity for the *non-frequent* characters is $O(sm^{2/3}) < O(s^{4/3}m^{1/3})$.

Algorithm Outline
1. Sort the probabilities in the weighted sequence
2. Divide the sorted list of probabilities into blocks of size $s^{2/3}$
3. For each block calculate the sum of probabilities and count the number of matches of the *non-frequent* characters using brute force.
4. For each block calculate the sum of probabilities and count the number of matches of the *frequent* characters using convolution.
5. For each text position add blocks probabilities from the largest down until the sum goes below the input threshold.
6. For each text position add probabilities within the last block until the sum goes below the input threshold.
end Algorithm Outline

Time Complexity: $O(s^{4/3}m^{1/3}\log s)$.

6.3 Combining the Algorithms

In order to obtain the minimal running time of both algorithm we start with first algorithm without doing the brute force part and check the average number of blocks per text location where a match was not found. If this number is not too large then we will proceed with the first algorithm and use brute force to calculate the sum of probabilities for these blocks and eventually the Hamming distance. If the number is too large, then we will use the second algorithm.

7 Conclusion and Open Problems

This paper defined the concept of approximate weighted distances, both in terms of Hamming distance and edit distance. We also presented efficient algorithms for these definitions. The algorithms are efficient in the sense that there are no known faster algorithms for edit distance in solid strings (by definition 1) or for finding the masked Hamming distance for solid strings. Further research directions would be to find efficient algorithms for the k-mismatches or k-error problems in weighted sequences.

Acknowledgments. The first author was partially supported by NSF grant CCR-01-04494 and ISF grant 35/05. The last author was partially supported by GIF Young Scientists Program grant 2055-1168.6/2002.

References

1. A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.
2. A. Amir, N. Lewenstein, and M. Lewenstein. Pattern matching in hypertext. *J. of Algorithms*, 35:82–99, 2000.
3. M. Christodoulakis, C. S. Iliopoulos, L. Mouchard, and K. Tsichlas. Pattern matchnig on weighted sequences. In *Proceedings of the Algorithms and Computational Methods for Biochemical and Evolutionary Networks (CompBioNets)*. KCL Publications, 2004.
4. R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized $o(n \log^3 m)$ time. *Proc. 29th ACM STOC*, pages 66–75, 1997.
5. R. Cole and R. Hariharan. Verifying candidate matches in sparse and wildcard matching. In *Proc. 34st Annual Symposium on the Theory of Computing (STOC)*, pages 592–601, 2002.
6. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1992.
7. M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. ACM*, 41(2):205–213, 1994.
8. M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.
9. 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.
10. C. S. Iliopoulos, K. Perdikuri, E. Theodoridis, A. Tsakalidis, and K. Tsichlas. Motif extraction from weighted sequences. In A. Apostolico and M. Melucci, editors, *Proc. 11th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3246 of *LNCS*, pages 286–297. Springer, 2004.
11. C.S. Iliopoulos, C. Makris, I. Panagis, K. Perdikuri, E. Theodoridis, and A. Tsakalidis. Computing the repetiotions in a weighted sequence using weighted suffix trees. In *European Conference on Computational Biology (ECCB)*, pages 539–540, 2003.
12. S. R. Kosaraju. Efficient tree pattern matching. *Proc. 30th IEEE FOCS*, pages 178–183, 1989.
13. V. I. Levenshtein. Binary codes capable of correcting, deletions, insertions and reversals. *Soviet Phys. Dokl.*, 10:707–710, 1966.
14. 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.
15. J.C. Venter and Celera Genomics Corporation. The sequence of the human genome. *Science*, 291:1304–1351, 2001.